Harnessing Cross-Layer Design

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Dipl.-Inform. **Ismet Aktaş**

aus Velbert, Deutschland

Berichter:

Prof. Dr.-Ing.	Klaus Wehrle
Prof. Dr. rer. nat.	Jörg Widmer

Tag der mündlichen Prüfung: 24.11.2015

Abstract

The success of today's Internet can partly be attributed to the design of the layered protocol stack. This design organizes communication protocols, that establish the rules of communication between different communicating entities, in hierarchical layers. These layers are strictly separated and offer only limited interfaces among adjacent layers. Essentially, protocols at each layer have a very specific task and they need to fulfill this task independently. Although this self-contained design of protocols worked well in wired networks, several problems appeared with the emergence of wireless and mobile communication. A prominent example is TCP's performance drop in wireless networks as it misinterprets packet loss, due to poor link conditions, as congestion in the network. In principle, the missing knowledge of higher layers about volatile wireless conditions and in case of lower layers about higher layer requirements leads to misinterpretation and misbehavior causing suboptimal performance.

A promising concept that addresses the lack of information availability is the cross-layer design paradigm which in fact circumvent the rules of strict layer separation and allows the interaction across non-adjacent layers. Many specific solutions, i.e., problem-oriented and tailor-made implementations, have demonstrated the utility of this paradigm by highlighting adaptivity advantages and performance improvements of applications and protocols. But a typical consequence of the very specific focus of the tailor-made solutions was the violation of software engineering principles such as maintainability and extensibility which are the major driving factors for the success and proliferation of software in general.

As a result of this observation, a few static cross-layer architectures have been proposed that facilitate systematic design and the integration of several specific solutions. Unfortunately, in static cross-layer architectures the cross-layer coordination algorithms are deeply embedded into the operating system (OS) and are realized at compile-time. This static and deep integration into the OS has several drawbacks. First, the design of cross-layer coordination algorithms requires relevant expertise to understand and modify protocols residing in the OS. Second, the experimentation with cross-layer coordination algorithms is tedious since their modification requires a recompilation. Third, coordination algorithms are always active even if not needed. Finally, application developers who know best about their application requirements and constraints are prevented from specifying and providing their own set of cross-layer coordination algorithms.

In this thesis, we present Crawler, a flexible cross-layer architecture that allows the specification, realization, and adaptation (i.e., addition, removal and modification) of cross-layer coordination algorithms at runtime. Based on the detection of underlying environmental changes, Crawler allows to automatically load the adequate set of coordination algorithms. It alleviates the problem of complicated access to relevant application, protocol, and system information by enabling a declarative and abstract way to describe cross-layer coordination algorithms and by providing a unified interface to inject such abstractions into the system. The generic design of this unified interface further enables the extensive experimentation with diverse compositions of cross-layer coordination algorithms and their adaptions. Moreover,

the interface allows applications to provide own coordination algorithms, to share information with the system and system monitoring. In this context, we classify problems such as conflicts when adding multiple cross-layer coordination algorithms and support developers to tackle them. In general, we enable an unprecedented degree of flexibility and convenience to monitor, experiment and run several cross-layer coordination algorithms. To further support the developer while experimenting, we even allow to remotely add, remove, and modify cross-layer coordination algorithms and their monitoring. We demonstrate the usability of Crawler for monitoring and experimentation with cross-layer coordination algorithms in five diverse use cases from different areas of wireless networking such as manipulating TCP behavior, VoIP codec switching, jamming detection and reaction.

Kurzfassung

Der Erfolg des Internets ist unter anderem zurückzuführen auf den hierarchischen Entwurf des Protokollstapels. Dieser ist in voneinander unabhängige Schichten unterteilt, deren Interaktion auf direkt benachbarte Schichten beschränkt ist. Die Schichten bestehen aus Protokollen, welche die Regeln der Kommunikation zwischen verschiedenen Kommunikationsentitäten vorgeben. Jedes Protokoll hat eine spezifische Aufgabe und muss diese unabhängig von anderen Protokollen erfüllen. In kabelgebundenen Netzen hat sich dieser geschichtete Ansatz über Jahre hinweg sehr gut bewährt. In mobilen und drahtlosen Netzen jedoch, in der sich die Funkschnittstelle ständig und unvorhersehbar ändert, führt der geschichtete Ansatz häufig zu einer zu langsamen und suboptimalen Anpassbarkeit. Ein bekanntes Beispiel ist der Leistungsabfall von TCP in drahtlosen Netzen. Hierbei geht TCP bei Paketverlusten fälschlicherweise immer von einem Stau im Netz aus, obwohl Paketverluste auch aufgrund schlechter Funkverbindungsqualitäten auftreten können. Im Allgemeinen führt die fehlende Kenntnis bei höheren Schichten über unvorhersehbare Netzbedingungen, die in unteren Schichten auftreten, und bei unteren Schichten das mangelnde Wissen über die Anforderungen, welche von den höheren Schichten vorgegeben werden, zu Fehlinterpretationen und zum Fehlverhalten.

Als vielversprechender Lösungsansatz für die oben genannten Probleme hat sich die sogenannte Cross-Layer-Koordination (CLK) erwiesen, die eine schichtenübergreifende Interaktion und Koordination zwischen allen Protokollschichten erlaubt. Viele Ansätze, die sich das Paradigma der CLK zu nutze gemacht haben, demonstrierten bereits in der Vergangenheit das immense Potential der Leistungsverbesserung. Jedoch waren diese Ansätze unsystematisch implementiert und jeweils auf sehr spezielle Probleme fokussiert, die sich meist auf die Interaktion zwischen zwei Schichten beschränkten. Eine direkte Folge solch einer unsystematischen Implementierung ist die Verletzung von Software-Entwicklungsparadigmen wie die Wartbarkeit und die Erweiterbarkeit, welche aber die Hauptantriebsfaktoren für den Erfolg und die Verbreitung von Software und somit auch des heutigen Protokollstapels sind.

Als Ergebnis dieser Beobachtung wurden einige statische Cross-Layer-Architekturen vorgeschlagen, die einen allgemeineren Ansatz verfolgen und den systematischen Entwurf und die Integration von mehreren spezifischen Ansätzen unterstützen. Leider sind bei statischen Architekturen die CLKs tief in das Betriebssystem eingebettet und zum Kompilierzeitpunkt erstellt, welches zu diversen Nachteilen führt.

Beispielsweise erfordert der Entwurf einer CLK fundierte Systemkenntnisse, um Protokollabläufe, die innerhalb des Betriebssystems ablaufen, zu verstehen und zu modifizieren. Ein anderes Problem ist, dass das Experimentieren mit CLKs mühsam ist, da eine Änderung der Optimierung ein erneutes Kompilieren und einen Neustart des gesamten Systems erfordert. Außerdem sind CLKs immer aktiv, auch wenn diese zeitweise nicht benötigt werden. Des Weiteren können Anwendungs-Entwickler, die am besten über ihre Anwendungsanforderungen und Einschränkungen wissen, nicht ihre eigenen Optimierungen dem System zur Verfügung stellen.

In dieser Arbeit wird die flexible Cross-Layer-Architektur Crawler vorgestellt, die je nach Bedarf erlaubt, CLKs zur Laufzeit einzuspeisen, zu entfernen und zu

modifizieren. Insbesondere ermöglicht Crawler basierend auf der Erkennung von Umgebungsänderungen, vordefinierte CLKs automatisch ins System zu laden oder wieder zu entfernen. Um den Zugang zu relevanten Anwendungs-, Protokoll- und Systeminformationen zu erleichtern, bietet Crawler zum einen eine deklarative Konfigurationssprache an, um von Systemdetails zu abstrahieren und um CLKs einfach zu beschreiben, und zum anderen eine einheitliche Schnittstelle, um Optimierungen dem System zur Verfügung zu stellen. Ferner erlaubt der generische Entwurf dieser einheitlichen Schnittstelle das umfangreiche Experimentieren mit unterschiedlichen Zusammensetzungen von CLKs. Insbesondere können Anwendungen diese Schnittstelle verwenden, um Informationen mit dem System austauschen, das System zu beobachten oder um eigene Optimierungen dem System bereitzustellen. In diesem Zusammenhang klassifizieren wir auch Probleme wie beispielsweise Konflikte, die entstehen können, wenn mehre Optimierungen in das System eingespeist werden und schlagen Ansätze vor, um diese zu beseitigen. Folglich bietet Crawler ein noch nie dagewesenes Maß an Flexibilität und Komfort, um mehrere CLKs einem System zu Verfügung zu stellen, wenn erwünscht zu experimentieren oder auch das ganze System (inklusive Optimierungen) zu beobachten. Um das Experimentieren noch weiter zu vereinfachen, erlaubt Crawler sogar das netzwerkweite Hinzufügen, Entfernen oder Modifizieren von CLKs und deren Monitoring. Wir demonstrieren Crawler anhand von fünf verschiedenen prototypischen Beispielen aus unterschiedlichen Bereichen der drahtlosen Kommunikation: TCP-Optimierung, VoIP-Code-Wechsel, Jamming-Detection und Reaktion.

Danksagung

An dieser Stelle möchte ich den besonderen Menschen danken, die mich während der Zeit der Promotion begleitet haben. Als erstes danke ich meiner Familie, weil Sie mir immer wieder klargemacht hat, dass es wichtigere Dinge im Leben gibt, als die Promotion. Im Speziellen möchte ich meinem Vater Mehmet dafür danken, dass er mich immer wieder mit seinen Geschichten über sein Leben motiviert hat. Meiner Mutter Sevim danke ich, dass Sie mir vorgelebt hat, wie man immer wieder aufsteht und kämpft, auch wenn das Leben manchmal die Karten anders mischt als erwartet. Bei meinem Bruder Ilker weiß ich, dass er für mich da ist, wenn nötig, sogar Berge für mich versetzen würde, danke! Meiner Schwester Ilkay danke ich für ihre Fürsorglichkeit, Kraft und Liebe, die Sie immer wieder für Andere aufbringt, auch für mich. Wo du bist läuft es wie am Schnürchen. Du erreichst alles was du willst, daran habe ich keinen Zweifel. Meiner Schwester Yildiz danke ich für Ihre liebenswerte, sympathische, süße und höfliche Art. Du bist mit deiner Art sehr speziell und etwas Besonderes. Dich gibt es garantiert nicht zwei Mal. Es ist immer wieder schön mit dir zu reden und zu träumen. Ein besonderer Dank gilt meiner Frau Dilek. Du warst immer wieder für mich da. Mit dir kann ich durch dick und dünn gehen. Ich danke dir für die wundervolle vergangene Zeit und freue mich auf die Momente, die noch auf uns zukommen. Meine liebe Familie Ihr habt mich aufgemuntert und unterstützt. Ich hoffe, dass ich euch mit dieser Arbeit etwas zurückgeben kann.

Aus ganzem Herzen danke ich folgenden Freunden. Cem und Azime Mengi über euch beide könnte ich so viel Gutes schreiben, es würde mehrere Bücher füllen. Ihr zwei seid tolle und vorbildliche Menschen. Woher findet ihr die Kraft, um so oft für andere Menschen da zu sein? Ich bin sehr dankbar euch kennengelernt zu haben. Wir haben gemeinsam sehr viel erlebt und ich hoffe, dass wir noch mehr gemeinsam erleben werden. Ihr habt so viel Gutes vorgelebt, ihr wart immer für mich da, vielen dank! Ibrahim und Ebru Armac sind auch zwei sehr liebenswerte Menschen, die ohne mit der Wimper zu zucken, für andere Menschen da sind. Ihr habt mir immer euer Ohr geschenkt und mich unterstützt wo es nur ging. Ihr lebt leider jetzt ein wenig weiter weg, aber ihr seid immer in meinem Herzen. Auch nicht anders sind Murat und Yasemin Basaran. Murat ist zwar die personifizierte Schnecke, aber wenn ich sage „ich brauche dich“, dann ist er da wie Speedy Gonzales. Er ist eine Art Papa für uns alle, aber wir alle wissen, dass eigentlich Yasemin das Sagen hat. Yusuf Bayram ist, wie einst mein lieber Freund Maurice sagte, der Italiener unter den Türken. Mit seiner lässigen, spontanen aber temperamentvollen Art ist er für jeden Spaß zu haben. Danke für die schöne und lustige Zeit, auch dir mein Freund Maurice. Ich vermisse die gute alte Zeit. Baris Tutar ist die Sympathie in Person. Wenn jemand den Kopf frei kriegen möchte, dann sollte er sich eine Woche Baris

verschreiben lassen. Außerdem kann er unglaublich gut motivieren, aber am Ende muss man aufpassen, dass man noch auf dem Boden bleibt. Ich danke dir für die vielen Motivationseinheiten und die lustigen Momente in den letzten Jahren. In meiner freien Zeit haben meine Freunde dafür gesorgt, den nötigen Abstand von der Arbeit zu bekommen. Ich danke euch für die schöne gemeinsame Zeit, ich habe es genossen und denk gerne mit einem Lächeln zurück.

Ich hatte das Glück mit großartigen Studenten arbeiten zu können. Ich durfte viel von diesen Studenten lernen und hoffe, dass ich auch ihnen ein wenig weitergeben konnte. Ihr habt einen großen Beitrag zu dem Ergebnis dieser Arbeit geleistet. Mit Tobias sind wir einen langen Weg über Seminararbeit, Hiwi-Job, Bachelorarbeit bis hin zur Masterarbeit gemeinsamen marschiert. Egal welches Problem, auf Tobi war verlass. Es war eine Freude mit dir zu arbeiten. Jens Otten war mein erster Student, den ich betreut habe. Wir haben in den Anfängen unglaublich viel diskutiert und sehr intensiv miteinander gearbeitet. Ich habe es genossen. Er hat mit seiner Arbeit einen großartigen Ausgangspunkt für die Architektur geschaffen. Mit Dominik Dennisen und Caj-Julian Schnelke habe ich mit zwei sehr motivierten und begabten Studenten an dem Thema Jamming gearbeitet. Dank euch hat mir das Thema unglaublich viel Spaß gemacht und ich denke wir haben es gut gemacht. Ihr seid zwei super Typen. Des Weiteren möchte ich Christoph Habets, Gloria Abidin, Kevin Möllering und Nikolas Koem für die gute Zusammenarbeit danken.

Ein weiterer Grund, warum das Thema Jamming mir so viel Spaß gemacht hat, lag auch an Oscar Puñal. Mit Oscar konnten wir schon immer gut gemeinsam lachen, aber während der Zusammenarbeit war es sogar noch mehr. Es war aber auch eine sehr produktive Zusammenarbeit, welches ich als meine Lieblingspublikation zähle. Bei Ericsson geht zumindest das gemeinsame lachen weiter. Wer weiß, vielleicht arbeiten wir bald wieder zusammen. Lieber Florian, niemand schenkt jedem einzelnen Wort in einem Paper so viel Liebe wie du. Wenn ich schreibe, denke ich oft daran, wie du es wohl formulieren würdest. Du konntest jeden guten Satz in einen noch schöner verwandeln. Außerdem danke ich dir für die vielen guten Unterhaltungen über Gott und die Welt. Es war und bleibt immer schön mit dir zu diskutieren. Mein lieber Raimondas, wir haben viel gelacht und viel über die Zukunft geträumt. Ich danke dir für die vielen schönen und lustigen Momente. Du bist ein Meister der Unterhaltung. Wir haben uns oft aus Spaß provoziert und geneckt. Ich hätte dich gerne noch länger beobachtet und mit dir gelacht. Wer weiß, vielleicht lachen wir noch weiter gemeinsam, wenn wir die erste Million machen. Dir traue ich es als einer der wenigen zu. Mein lieber Hamad, du kannst das Wesentliche klar auf den Punkt bringen. Ich bin sehr froh und dankbar, dass ich mit dir arbeiten durfte und erfahren konnte wie du die Dinge angehst. Ich danke dir, dass du mich mit dieser Arbeit unterstützt hast. Du bis jemand, der hilft ohne eine Gegenleistung zu erwarten. Du bist mein heimlicher Held. Lieber Tobias, du bist ein super Motivator. Du weißt, wie man Dinge angeht und wie man Leute begeistert. Oft hast du auch mich motiviert. Du bist jemand der konstruktiv kritisiert, danke. Lieber Elias, viel Zeit musste vergehen bis ich dich verstanden hatte. Ich habe erst nachdem du den Lehrstuhl verlassen hast bemerkt, dass ich viele deiner Eigenarten mag und sehr sympathisch finde. Deine offene und neugierige Art gegenüber allem finde ich toll. Was macht die Kunz? Ich habe dich so oft geärgert. Du wirst mein Lieblingsopfer bleiben, aber ich habe noch nie jemanden gesehen, der sich so wenig beschwert und immer so höflich ist. Du bist mit jedem befreundet. Ich weiß nicht, was das für mich

bedeutet. Bin ich etwa doch nicht besonders? Na ja, zum Glück kann ich noch die Zeit mit dir bei Ericsson genießen. Jeden Morgen und Abend habe ich das Glück, dass du mir dein Ohr schenkst und wir wirres Zeug reden. Ich danke dir dafür, nicht immer nur über Informatik und Kommunikation zu reden, sondern es kurios und bizarr du praktizieren. Man könnte sagen es ist Kunz. Wenn man mal eine Idee hat oder eine braucht, dann muss man zu Jo Ágila Bitsch. Man kann natürlich googlen, man kann aber auch Jo fragen, er weiß einfach alles. Google hatte einst das Ziel das Internet auf mehreren Servern zu speichern, aber Jo hat es bereits seit Jahren im Kopf. Er kann dir genau sagen, ob es die Idee gibt oder er sagt dir wie es neuartig werden kann. Es war immer super spannend und ein tolles Erlebnis mit dir zu diskutieren. Kennt ihr das Lachen von Uta? Wenn sie gelacht hat, dann musstest du mitlachen, auch wenn du mal eine Etage darüber warst. Sie war die gute Stimmung des Lehrstuhls. Sie hat für die gute Laune gesorgt. Janosch dagegen war der Logistiker. Dank ihm haben wir überhaupt einiges organsiert bekommen. Janosch du bist Mr. Smiley, immer ein Lächeln im Gesicht. Da gibt es auch noch den Benjamin Schleinzer für gute Laune, aber den darf man sich erst ab FSK18 anhören. Der Gesetzgeber kennt ihn leider nicht, sonst müsste man bei ihm eigentlich FSK30 einführen. Was der für Witze kennt und dann noch so viele, da ist der Bauchmuskelkater vorprogrammiert. Ein großer Dank gilt unseren beiden Sekretärinnen Petra Zeidler und Ulrika May. Sie sind liebevolle Menschen. Es war mir eine große Freude mit euch zu arbeiten. Zwei weitere liebenswerte Powerdamen, die ich noch sehr vermissen werde, sind Mónica Lora und Sepideh Ebrahimi. Mit euch konnte man sich nicht nur sehr gut fachlich unterhalten, sondern auch über alles Andere. Man konnte immer zu euch und ihr hattet immer Zeit. Vielen dank ihr Zwei! Mit Rainer Krogull hatte ich immer sehr amüsante Gespräche. Wenn ich Pausen hatte, habe ich ihn immer ziemlich gut geärgert und er hat sich nie beschwert. Gemeinsam haben wir an vielen Ecken des neuen Gebäudes rumgewerkelt und dank ihm konnten wir viele Dinge für den Lehrstuhl realisieren. Des Weiteren möchte ich mich bei Jan Rüth, Martin Serror, Martin Henze, Christian Dombrowski, Oliver Hohlfeld, Dirk Thißen, Torsten Zimmermann, Hendrik vom Lehn, Jan Henrik Ziegeldorf, Kai Jacobs, Donald Parruca, Nicolai Viol, Marco Weyres, Otto Spaniol und Rene Hummen bedanken. In der letzten Phase dieser Arbeit hat mich mein neuer Kollege Junaid Ansari und mein Vorgesetzter Michael Meyer unterstützt. Junaid ist wie ein großer Bruder, der sich meine Beschwerden viel zu häufig anhörte und auch immer einen guten Ratschlag zu geben wusste. Ich danke Michael, dass er mir die notwendige Zeit und die Unterstützung gegeben hat, um diese Arbeit zu beenden. Außerdem danke ich meinem Betreuer Klaus Wehrle zur Möglichkeit der Promotion und Herrn Jörg Widmer für die Zweitbetreuung dieser Arbeit. Zuallerletzt möchte ich mich auch bei den Menschen bedanken, die es mir nicht so einfach gemacht haben im Leben. Durch sie weiß ich, dass ich nicht werden möchte wie sie.

Contents

1 Introduction **1**

1.1 Problem Analysis 2

1.1.1 Problem Statement 3

1.1.2 Research Questions 5

1.2 Contributions 6

1.2.1 Relationship between Research Questions and Contributions . 7

1.3 Outline 8

2 Background and Related Work **9**

2.1 Layered Design 9

2.2 Cross-Layered Design 11

2.2.1 Cross-Layer Design Definitions 12

2.2.2 Information Exchange Alternatives 13

2.2.3 Architecture Classifications 15

2.2.4 Cross-Layer Information Processing 16

2.2.4.1 Synchronous and Asynchronous Processing 17

2.2.4.2 User and Kernel Space Separation 17

2.3 Related Work 20

2.3.1 Specific Cross-Layer Solutions 21

2.3.2 Approaches of Varying Scope 21

2.3.3 Cross-Layer Architectures 24

2.3.3.1 Static Cross-Layer Architectures 25

2.3.3.2 Flexible Cross-Layer Architectures 30

3 A Generic and Flexible Cross-Layer Architecture 31

3.1 Motivation . . . 32

3.2 Problem Analysis . . . 32

3.3 Design Overview . . . 35

3.3.1 Goals . . . 36

3.3.2 Relationship of Research Questions and Goals . . . 37

3.3.3 Design Scope and Limitations . . . 38

3.4 Architectural Details . . . 39

3.4.1 Manageability . . . 39

3.4.1.1 Configuration . . . 40

3.4.1.2 Interpreter . . . 42

3.4.1.3 Repository . . . 42

3.4.2 Application Support . . . 43

3.4.3 Runtime Flexibility & Extensibility . . . 47

3.4.3.1 FU Wiring . . . 47

3.4.3.2 Stubs – Accessing Signaling Information . . . 49

3.4.4 Context Adaptability . . . 50

3.5 Implementation and Architectural Overhead . . . 53

3.5.1 Implementation . . . 53

3.5.2 Architecture Overhead . . . 54

3.6 Conclusion . . . 56

4 Practical Use Cases and Evaluation with CRAWLER 59

4.1 Use Case: Manipulating TCP's Congestion Window and Application Behavior . . . 60

4.1.1 Motivation . . . 60

4.1.2 Setup and Monitoring . . . 60

4.1.3 Cross-Layer Coordination Approach . . . 61

4.1.4 Realization with CRAWLER . . . 61

4.1.5 Validation . . . 62

4.1.6 Summary and Discussion . . . 62

4.2 Use Case: Switching TCP's Congestion Control Algorithm . . . 62

4.2.1 Motivation . . . 63

4.2.2 Setup and Monitoring . . . 63
4.2.3 Cross-Layer Coordination Approach . . . 64
4.2.4 Realization with CRAWLER . . . 65
4.2.5 Validation . . . 65
4.2.6 Summary and Discussion . . . 66
4.3 Use Case: VoIP Codec Switching . . . 67
4.3.1 Motivation . . . 68
4.3.2 Setup and Monitoring . . . 68
4.3.3 Cross-Layer Coordination Approach . . . 70
4.3.4 Realization with CRAWLER . . . 70
4.3.5 Validation . . . 72
4.3.6 Related Work . . . 74
4.3.7 Summary and Discussion . . . 74
4.4 Use Case: Dynamic Adaptation of Jamming Detection and Reaction Strategies . . . 75
4.4.1 Motivation . . . 75
4.4.2 Setup and Monitoring . . . 76
4.4.3 Cross-Layer Coordination Approach . . . 78
4.4.4 Realization with CRAWLER . . . 78
4.4.5 Validation . . . 82
4.4.6 Summary and Discussion . . . 84
4.5 Use Case: Machine Learning-based Jamming Detection . . . 84
4.5.1 Motivation . . . 85
4.5.2 Setup and Monitoring . . . 86
4.5.2.1 Sensitivity of Metrics to Jamming . . . 87
4.5.2.2 Threshold Identification Problem . . . 89
4.5.3 Cross-Layer Coordination Approach . . . 90
4.5.4 Realization with CRAWLER . . . 90
4.5.5 Validation . . . 92
4.5.5.1 Indoor Detection Accuracy . . . 92
4.5.5.2 Impact of Outdoor Mobility . . . 93
4.5.5.3 Cooperation Between Nodes . . . 94
4.5.6 Related Work . . . 95
4.5.7 Summary and Discussion . . . 96
4.6 Conclusion . . . 96

5 Coping with Multiple Cross-Layer Coordination Algorithms 99

5.1 Motivation . . . 100

5.2 Problem Analysis . . . 100

5.3 Cross-Layer Conflict Detection . . . 101

5.3.1 Classification of Cross-Layer Conflicts . . . 101

5.3.2 Detecting Direct Conflicts . . . 102

5.3.3 Detecting Indirect Conflicts . . . 104

5.3.4 Related Work . . . 105

5.3.5 Summary and Discussion . . . 105

5.4 Cross-Layer Redundancy Removal . . . 106

5.4.1 Generic Design . . . 107

5.4.1.1 Constraints . . . 107

5.4.1.2 Equality of Module Compositions . . . 108

5.4.2 Graph-based Iterative Merge Algorithm . . . 109

5.4.2.1 Input Equality . . . 109

5.4.2.2 Behavior Equality . . . 110

5.4.2.3 Merging Modules . . . 111

5.4.2.4 Runtime and Memory Consumption . . . 111

5.4.3 Runtime Adaptation . . . 112

5.4.3.1 Challenges when Adding/Removing Modules and Connections . . . 112

5.4.3.2 Splitting Affected Modules . . . 113

5.4.4 Specific Design for CRAWLER . . . 114

5.4.4.1 Handling Runtime Adaptation . . . 114

5.4.5 Evaluation and Validation . . . 116

5.4.6 Related Work . . . 118

5.4.7 Summary and Discussion . . . 119

5.5 Conclusion . . . 120

6 Evaluation Support for Cross-Layer Coordination 121

6.1 Motivation . . . 121

6.2 Problem Analysis . . . 122

6.3 Remote Cross-Layer Evaluation . . . 123

6.3.1 Design Overview . . . 125
6.3.2 Architectural Details . . . 126
6.3.2.1 Remote Automation . . . 127
6.3.2.2 Remote Configuration . . . 130
6.3.2.3 Remote Monitoring . . . 132
6.3.2.4 Graphical and Interactive Front-End . . . 136
6.3.3 Implementation . . . 138
6.3.4 Evaluation . . . 138
6.3.4.1 Evaluating Remote Automation . . . 140
6.3.4.2 Evaluating Remote Configuration and Monitoring . . . 140
6.3.5 Related Work . . . 142
6.3.6 Future Work . . . 145
6.3.7 Summary . . . 145
6.4 Network Emulation Tool – FANTASY . . . 146
6.4.1 Design Overview . . . 147
6.4.2 Architectural Details . . . 149
6.4.2.1 Host Configuration Unit (HCU) . . . 149
6.4.2.2 Guest Configuration Unit (GCU) . . . 152
6.4.3 Implementation . . . 154
6.4.4 Evaluation . . . 154
6.4.4.1 Demonstrating Areas of Application . . . 154
6.4.4.2 Demonstrating Scalability, Automation and Rapid Testing . . . 158
6.4.5 Related Work . . . 159
6.4.6 Future Work . . . 160
6.4.7 Summary . . . 161
6.5 Conclusion . . . 161

7 Summary and Conclusions **163**
7.1 Contributions . . . 164
7.2 Future Work . . . 168
7.2.1 Increasing the Toolbox of Reusable Functional Units and Stubs 168
7.2.2 Conflict Resolution . . . 168
7.2.3 Timing Constraints . . . 168
7.2.4 Potential Use-Cases . . . 170
7.2.5 Realization on Further Platforms . . . 170

Glossary **172**

Bibliography **175**

A Syntax of CRAWLER's Configuration Language **189**

B Available Stubs and FUs in CRAWLER **191**

B.1 Stubs 191

B.2 FUs 194

C Configuration of Machine Learning-based Jamming Detection **195**

1

Introduction

Traditional network protocol stacks are logically organized in layers. These layers are strictly separated and the interaction between them is restricted by concise interfaces, which in effect only allow few limited procedure calls and responses such as passing packets up and down the stack. Interaction between nonadjacent layers is prohibited. Thus, layers are designed to fulfill their functionality independently by only interacting with adjacent layers and not across layers. History shows that this works well in wired and static environments [Pen00].

However, today's applications and protocols for wireless and mobile systems have to deal with volatile environmental conditions such as interference, fading, and mobility. In order to adapt to the rapidly and frequently changing network conditions, a more sophisticated interaction between protocols than in a traditional layered architecture is beneficial. For example, TCP interprets packet loss as congestion in the network and decreases its sending rate. In a wired environment, with the usage of a traditional layered protocol stack, this behavior is appropriate and improves the overall network performance, but in a wireless environment this behavior leads to performance degradation [SRK03, Pen00]. One major reason is that the packet losses in the wireless medium are much higher due to interference, signal fading, mobility, etc. compared to packet losses in wired networks which usually caused by congestion in the network [BSAK95, DCY93]. The knowledge about the wireless medium and the procedures at lower layers would help TCP to avoid such misinterpretations, keep its sending rate and accordingly improve performance [BSK95, BSAK95, CRRP04, SRK03].

A promising research concept to deal with conditions such as the volatile wireless medium and mobility is cross-layer coordination [LSS06, SRK03, SM05], that is, the exchange of information across non-adjacent layers. Utilizing information from other protocols and system components can improve the performance and responsiveness of applications and protocols. For example, in mobile and wireless systems, even a single cross-layer coordination process at the MAC layer can achieve throughput increases of 20 times and latency reduction of 10 times over regular TCP [BSK95].

Despite this tremendous potential to enhance system performance and boosting a fair share of research investment in recent years, the cross-layer design pardigm has not been able to leverage its utility beyond few promising yet problem-oriented research efforts [SM05, Yu04, RI04a]. Among other reasons, this can partly be attributed to the solution-oriented, tailor-made, and naive utilization of this design paradigm. Since the emergence of this design paradigm, it lacks a generic and flexible system architecture that enables developers to specify and experiment with diverse cross-layer coordination processes. Before being able to design such an architecture, we have to analyze and fully understand the problem space which we will bring closer in the following section.

1.1 Problem Analysis

Existing approaches either offer a *specific cross-layer coordination process* or a *static architecture* that combines several coordination processes but neither supports dynamic adaptation (i.e., addition, removal and modification) of cross-layer coordination processes nor context adaptation, i.e., automatically loading the adequate set of coordination processes at runtime based on the detection of underlying environmental changes.

In particular, a specific cross-layer coordination process is a tailor-made solution to optimize a certain behavior of the system, for example, a wireless MAC implementation monitoring TCP acknowledgments to prevent overzealous congestion control behavior [BSK95].

In contrast, a static cross-layer architecture [WAR03, SKC05, CMTG04, RI04b] facilitates easy manipulation of protocol-stack parameters and combines several specific cross-layer coordination processes. In current architectures of this type, cross-layer coordination processes are composed offline (i.e., at compile time) and are deeply embedded within the operating system (OS). Moreover, static cross-layer architectures have three key limitations that motivate the ideas presented in this thesis.

First, the process of cross-layer coordination development is tedious due to two reasons: (i) The deep integration of protocols and system components into the OS requires expert knowledge and significant effort to realize the desired cross-layer coordination process [AAS+14] and (ii) the ability to flexibly (de)activate cross-layer coordination processes based on environmental and network condition changes requires their detection and an immediate automatic adaptation [SM05].

Second, when independently developed coordination processes are added into the system this can cause two possible performance degradations: (i) contradicting coordination goals, i.e., coordination conflicts [Wil08] and (ii) redundant processing resulting in unnecessary memory usage and a waste of CPU time [AHA+14]. Detecting cross-layer conflicts remains one of the major unresolved challenges in the cross-layer development domain [KKTC05, SM05, Wil08].

Third, testing and subsequent analysis of test results is difficult and inconvenient when cross-layer coordination processes are involved since: (i) the unpredictable and frequently changing nature of the wireless medium complicates the interpretation of results. While physical layer issues such as noise and interference are one reason,

there exist also further reasons such as competition regarding the shared wireless medium or mobility. (ii) Many testing scenarios require the preparation of many nodes including the installation of programs and cross-layer coordination makes the testing tedious. Moreover, while and after running a test, system variables which might be difficult to access need to be logged and manually collected for later evaluation likely with different preferences on each node.

Designing an easy-to-use and generic cross-layer architecture that is able to automatically adapt to permanent network changes requires to tackle all these major problems. In the next section, we discuss the identified problems in more detail and derive three major research questions that are addressed by this thesis.

1.1.1 Problem Statement

From the problem analysis presented in the previous section we derive four observations that hinder the successful utilization and proliferation of a cross-layer architecture to develop, evaluate, debug, and run cross-layer coordination processes for current and upcoming wireless and mobile systems.

Lack of a generic and flexible cross-layer architecture for system monitoring, cross-layer design and experimentation

Developers face three major problems when designing their cross-layer coordination processes.

First, the process of adding or removing a cross-layer coordination is cumbersome: a coordination processes needs to be patched with the architecture, and because the architecture is deeply embedded into the OS, recompiling the kernel and rebooting the system are typical consequences. Hence, the developer has to deal with many system internals before designing efficient cross-layer coordination processes. Ideally, a developer should be relieved from the burden of system details and only focus on the pure design of cross-layer coordination processes which will simplify and accelerate their development.

Second, due to the same reason of deep integration into the OS and the fact that the design of a cross-layer coordination process requires several steps of improvements until finalization, experimentation and development cycles are too long and inconvenient. Hence, in an ideal case a developer should have the opportunity to add, remove and fine-tune cross-layer coordination processes at runtime and the ability to monitor any parameter in the system to analyze the behavior of the coordination process. Both features will help developers to experiment with cross-layer coordination processes and to finalize them.

Third, because of the static nature of existing architectures, a coordination process will adapt the system behavior even if it is not needed. For example, an application or environment-specific coordination process is not required when that application is not running or the underlying network conditions have changed in a mobile network. Hence, this coordination process and its interaction with the network stack is superfluous and may even adversely effect other active applications. We strongly believe that this is against the original idea of the cross-layer design paradigm [AOSW10]

which underlines the need for dynamic adaptation of the system behavior (i.e., protocols, hardware components like sensors, and applications) based on application requirements, system state and the network conditions.

Lack of an available cross-layer architecture and use-cases

Many specific cross-layer solutions have been proposed that demonstrate the usefulness of cross-layer design [BSK95, BSAK95, CRRP04]. However, the practical use for most of them is questionable since they are not built with software engineering principles (such as modularity, maintainability, reusability, and usability) in mind. Including all or even some of them will likely result in unbridled cross-layer design (i.e., the so-called spaghetti design [KKTC05]) which could lead to unmanageable and complex interdependencies in the code. As a consequence of this observation a couple cross-layer architectures have been proposed that leverage software engineering advantages [CMTG04, RI04b]. Unfortunately, these architectures lack useful examples since they only provide simple showcases to demonstrate their mechanisms rather than real-world cross-layer coordination use-cases. Moreover, the suggested architectures and showcases are not open to the public, hindering the credibility and acceptance of the community. Accordingly, one major problem of the cross-layer design research field is the lack of an available cross-layer architecture that shows convincing and novel cross-layer coordination examples and which also demonstrates the improved development support, convenience and freedom of more flexibility in designing cross-layer coordination processes.

Lack of developer support to handle multiple coordination algorithms

The process of designing and finalizing a single specific cross-layer coordination process as such is already very cumbersome. But when adding multiple coordination algorithms into the system, interdependencies are created which may lead to a peculiar system behavior. While a single specific coordination process is designed to improve the system behavior, multiple cross-layer coordination processes may have contradicting effects. Moreover, even when a set of multiple cross-layer coordination processes might improve performance in a specific scenario, e.g., multiple energy coordination processes that trade improved energy consumption for lower throughput while running in the battery mode, the same set of coordination processes could lead to performance degradation in another scenario, e.g., unnecessary tradeoff between energy and throughput when plugged into power. Therefore, developer support while experimenting with a set of coordination processes is a necessity to create the right set of coordination algorithms for a particular scenario. Redundancy is another resulting problem that occurs when different developers may decide to add coordination algorithms into the system where parts of the coordination algorithms may have similar instructions which waste CPU time and memory. All in all, a developer lacks support when experimenting with multiple or even a single cross-layer coordination.

Lack of evaluation support

After the design and implementation of a cross-layer coordination idea a next step is its validation, i.e., testing that the coordination is working as intended and demonstrating its benefits. This task has two major problems. First, due to the volatile nature of the wireless medium such as interference or other factors such as programming faults, evaluating cross-layer coordination processes and identifying root

causes of (mis)behavior in wireless environments is tedious. Hence, to ensure that a cross-layer coordination process is working and improving the performance as intended and to exclude misbehavior, many test runs have to be conducted and very likely in many varying scenarios. Accordingly, having more control and transparency about these factors will simplify testing and improve developers to eliminate a range of potential problems. Second, many of the suggested cross-layer coordination processess target at wireless scenarios where several nodes are involved. Setting up such scenarios requires much effort. For instance, many programs including cross-layer coordination processes have to be manually started and coordinated among the test nodes. Finally, test results have to be manually gathered from all devices. When repeating these experimentations, for instance, to gain more credibility in volatile wireless scenarios, all the exhausting steps have to be reapplied. Therefore, developers would significantly benefit when such experiments are automatized and the collection of test results is supported, preferably very conveniently, without physical interaction, and centrally controlled.

In the following section, we highlight the research questions that arise with the presented problems and present our contributions to tackle these questions.

1.1.2 Research Questions

From the problems presented in the previous section we derive the following three research questions. In the remainder of this thesis we present answers to these questions.

Question Q1 - How to enable a generic and runtime-flexible cross-layer architecture that facilitates rapid system monitoring, cross-layer design and experimentation?

We present the design of Crawler, a generic and flexible cross-layer architecture that, based on a given abstract configuration, derives cross-layer coordination processes and (de)activates them when specified conditions are satisfied. Crawler's rich interface allows to inject coordination processes into the system which can also be used by applications to exchange variables with the system. The intended coordination processes are achieved through the use of modules that can be freely composed at runtime.

Question Q2 - How to handle problems caused by multiple cross-layer coordination processes?

We investigate and classify problems such as conflicting coordination goals and redundant cross-layer coordination processing which occur when multiple coordination processes are enabled in the system and provide mechanisms to handle such circumstances.

Question Q3 - How to improve the evaluation of cross-layer coordination algorithms?

We support developers by providing two different frameworks which enable control of influencing factors, effective testing, central system monitoring and analysis of cross-layer coordination algorithms running on several devices, even at runtime.

1.2 Contributions

We address the aforementioned questions with CRAWLER, a **cr**oss-layer **a**rchitecture for **w**ire**le**ss netwo**r**ks and its extensions. In particular, we present four complementary contributions which are listed in Table 1.1.

Contributions	Peer-reviewed international publications	Non-peer-reviewed preliminary work
C1) A generic and flexible cross-layer architecture to conveniently and rapidly realize runtime adaptive cross-layer coordination algorithms, their monitoring, and experimentation.	[AOSW10] [ASA+12] [AAS+14]	[Ott09] [Drü10] [Den09]
C2) Novel cross-layer coordination use-cases to demonstrate the versatility and convenience of CRAWLER in various networking fields to realize and monitor cross-layer coordination algorithms.	[ASW+12] [ASA+12] [AAS+14] [APS+14a]	[Sch11] [Den12] [Sch13] [Abi13]
C3) Handling support for multiple cross-layer coordination algorithms to deal with unintended contradicting effects and to solve parts of redundant processing caused by multiple running cross-layer coordination algorithms.	[AAS+14] [AHA+14]	[Koe11] [Möl11]
C4) Evaluation support for cross-layer coordination to simplify and accelerate the testing, monitoring, and validation of cross-layer coordination algorithms.	[AvLH+12] [APS+14b]	[Hab11] [Drü13]

Table 1.1 Contributions of this thesis.

In the following we briefly discuss each of these contributions.

Contribution C1 - A Generic and Flexible Cross-Layer Architecture

We designed CRAWLER, a generic and flexible cross-layer architecture that facilitates the monitoring of system information (protocols and system components) and the realization of cross-layer coordination algorithms at a high level of abstraction. To provide the abstraction, CRAWLER offers a rich interface which can be used by applications to exchange information with the system and to provide own coordination algorithms. CRAWLER also offers flexibility that is essential for adjusting and experimenting with different sets of cross-layer coordination algorithms and, furthermore, extensibility for involving all possible protocols and system components. In addition, CRAWLER allows detecting underlying environmental changes and to automatically (un)load the adequate set of cross-layer coordination algorithms.

Contribution C2 - Novel Cross-Layer Coordination Use-Cases

By using CRAWLER, we show how we realized novel specific cross-layer coordination use-cases from various fields of networking. Especially, we highlight CRAWLER's monitoring capability to detect peculiar application and protocol behavior under certain networking conditions. Based on observations gained with CRAWLER's monitoring capability, we formulate coordination ideas and show their convenient and rapid realization by using CRAWLER. Dependent on the use-case, we present the achieved flexibility or relative performance gain.

Contribution C3 - Handling Support for Multiple Cross-Layer Coordination Algorithms

Giving developers the freedom to add their own cross-layer coordination algorithms into the system without knowledge about existing cross-layer coordination algorithms in the system might lead to two problems: (i) Coordination conflicts, i.e., running multiple cross-layer coordination algorithms in parallel could lead to unintended contradicting effects resulting in severe performance degradations. CRAWLER supports developers by providing debugging support for monitoring cross-layer coordination algorithms and their effects. (ii) Redundancy, i.e., the overall resulting union of cross-layer coordination algorithms may become suboptimal as some of the added cross-layer coordination algorithms are already in the system and have redundant instructions utilizing more CPU and memory than necessary. CRAWLER enables automatic detection and resolution of such redundancies without developer interaction.

Contribution C4 - Evaluation Support for Cross-Layer Coordination

Testing cross-layer optimizations in wireless environments is tedious. On the one hand, setting up the experiment requires huge effort to install the relevant software involving the cross-layer coordination algorithms and to coordinate them among different devices. On the other hand, an experiment usually requires many test runs and effort since the volatile nature of the wireless medium makes the evaluation results hard to interpret. To tackle these problems, we provide two extensions to CRAWLER. First, we coupled CRAWLER with simulation (ns-3) resulting in a network emulation architecture that allows the fully automated setup and execution of an experiment in a controllable environment to improve the monitoring and analysis of cross-layer coordination algorithms; Second, we extended CRAWLER's interface to remotely (a) allow the automation of test runs on different devices, (b) add, remove and modify cross-layer coordination algorithms at runtime and (c) live monitor and log different parameters in the system.

These contributions were partially developed in supervision and cooperation with students in the context of their Bachelor, Master and Diploma thesis [Ott09, Drü10, Den09, Koe11, Hab11, Sch11, Möl11, Den12, Drü13, Sch13, Abi13]. Table 1.1 shows the correlation between the supervision and the contribution of this thesis. At this point, I would like to explicitly thank these students for their contributions to this thesis.

1.2.1 Relationship between Research Questions and Contributions

The four previously presented contributions provide answers to the three identified research questions. Figure 1.1 provides the relationship between the research questions and the contributions.

In particular, question Q1 targets at enabling a cross-layer architecture for convenient realization, monitoring and runtime adaptation of cross-layer coordination algorithms. On the one hand, developers should be supported to build their cross-layer coordination algorithms. On the other hand, developers should be supported

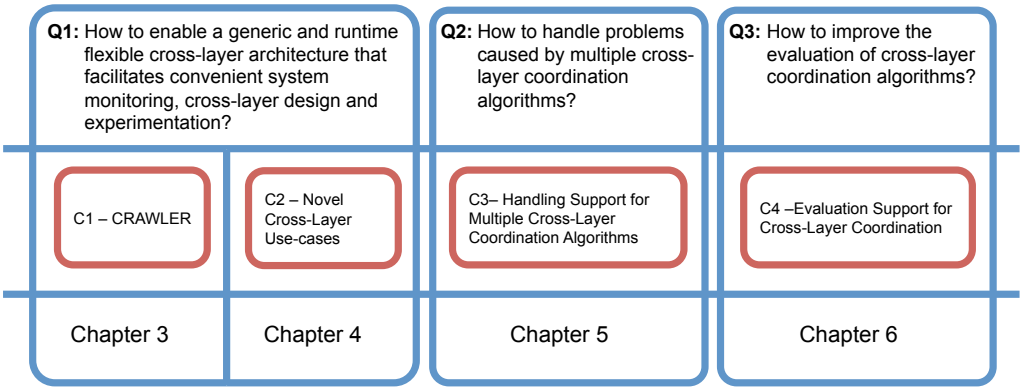


Figure 1.1 Overview presenting the relationship between research questions and contributions of this thesis.

to have a flexible and adaptive system to experiment and finalize their cross-layer coordination algorithms. This is tackled by our contribution CRAWLER (C1). Moreover, CRAWLER's features enabled us to experiment and design novel cross-layer coordination algorithms in divers fields of networking which highlight the versatility and practical use of CRAWLER and mark our second contribution, namely Novel Cross-Layer Use-Cases (C2).

The research questions Q2 and Q3 are answered on a one-to-one basis by the contributions C3 and C4, respectively. Contribution C3 particularly classifies and addresses issues that are involved when running multiple cross-layer coordination algorithms in the system. In contrast, contribution C4 targets the general case of supporting the developer during experimentation with cross-layer coordination algorithms. Thus, we focus on aspects such as improving the convenience, automation, monitoring and control over the experimentation.

1.3 Outline

The remainder of this thesis is organized as follows. In Chapter 2, we lay out the basis of the thesis by introducing the fundamental aspects of cross-layer design. We discuss problems and limitations when establishing cross-layer optimizations and formulate requirements for an architecture. Based on these requirements, we look deeper into existing cross-layer architectures and discuss their eligibility. Chapter 3 presents the design of our cross-layer architecture CRAWLER with all of its components and shows its performance. In the subsequent Chapter 4 we demonstrate the practical use of CRAWLER by means of four real-world use cases from diverse networking fields. Thus, we prove the real applicability of CRAWLER in order to explore novel solutions of cross-layer coordination algorithms. Afterwards, Chapter 5 presents our contributions to tackle problems involved with multiple cross-layer coordination algorithms. Chapter 6 targets at simplifying and improving the evaluation process of cross-layer coordination algorithms. Finally, Chapter 7 concludes the thesis by summarizing our contributions and discussing future work.

2

Background and Related Work

Cross-layer design is a is a widely used term in the research field of communication. The literature comprises a very diverse set of approaches including theoretical analysis, simulation-based and real-world implementations of single specific cross-layer coordination algorithms, or cross-layer architectures enabling to run several specific coordination algorithms. But before we interrelate these investigations to each other, we first introduce the relevant fundamentals that lay the foundation for this thesis. After providing a sound knowledge about the basics, we give a brief overview of state-of-the-art cross-layer architectures and derive the missing features that are relevant for providing a generic and flexible cross-layer architecture.

2.1 Layered Design

A major milestone that has contributed to the success of today's Internet is the design and realization of the layered protocol stack, that is, the TCP/IP protocol suite [For02]. It consist of hierarchically organized protocol layers where each layer has a specific task. The protocol fulfilling this task is restricted to only use few interfaces to adjacent layers. In particular, the interfaces only allow to use specific procedure calls such as passing packets up and down to an adjacent layer. One intention for a clear separation of the layers was to make protocols modular and self-contained, i.e., without the reliance on other protocols and applications. This has several architectural benefits such as (i) better maintainability of the whole protocol due to encapsulation and grouping of functionality, (ii) extensibility to add new protocols and algorithms without affecting remaining protocols in the system which allows to deal with the persistent evolution of functionality and requirements, and (iii) flexibility to adapt or exchange specific functionalities.

With the permanent evolution of the Internet partially driven by user demands and partially by technological development, many protocols at different layers appeared. Figure 2.1 gives an overview about the protocol stack structure of today's communication systems.

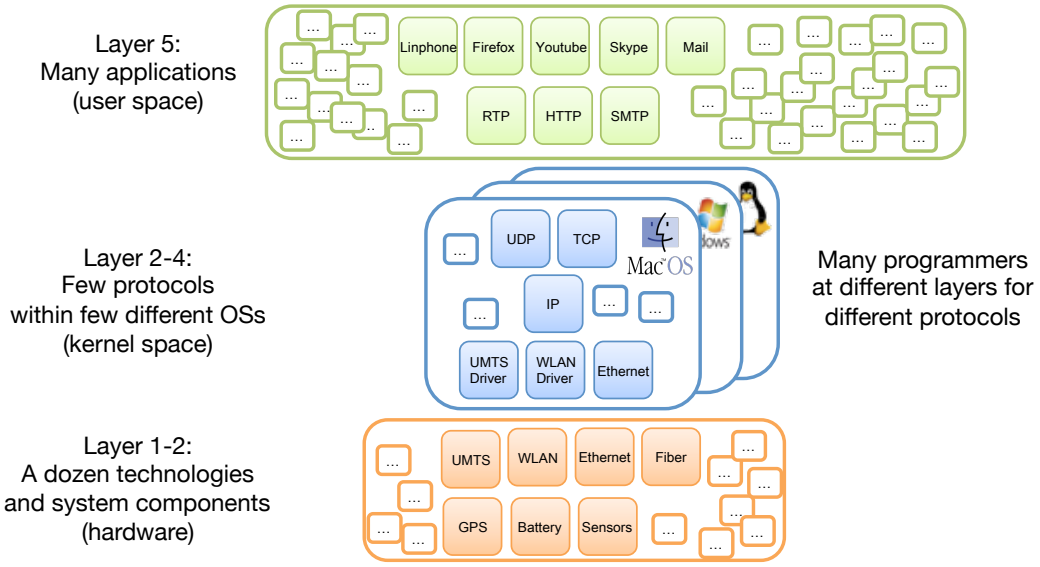


Figure 2.1 The layered TCP/IP protocol suite: A dozen technologies exist at layer one and two that are realized monolithically and in hardware, whereas in few cases layer two is also provided in software and resides in the kernel space of the operating system. Even lesser protocols exists in layer three and four which also reside in the kernel space. In contrast, in the range of hundreds and thousands applications exist running in user space.

The layers one and two which are referred to as physical and data link layer are typically built monolithically in hardware and combined into a single specific technology such as Ethernet for wired networks or WLAN or UMTS for wireless networks. Only a dozen technologies (realizing layer one and two) have been established since the emergence of the Internet. In contrast to the application layer where many applications such as browsers, messaging clients and VoIP clients are available. These applications utilize application layer protocols such as RTP, HTTP, FTP and SMTP to support their communication with their counterparts residing on remote hosts. In-between the application layer and the data link and physical layer, there are few network and transport layer protocols. Unfortunately, network and transport protocols do not offer much freedom in selection. Especially this holds true for the network layer where the majority of the users still use IPv4 although its improved version (IPv6) has been formally described in 1998 [KV13]. One major reason is that a change of the the network protocol involves the costly adoption of millions of routers in the core of the Internet.

An interesting fact, however, is that the TCP/IP protocol suite was originally not intended and designed to become the de-facto standard of the today's Internet, as at the time of its initial deployment nobody could image about its potential [MS03]. Therefore, the design of TCP/IP was never a one-size-fits-all implementation. It was continuously enhanced with the availability of new and enhanced protocols and applications to deal with various arising problems. Thus, the TCP/IP protocol suite underwent and still follows an evolutionary process.

To conclude, many protocols at different layers mainly for a couple of prominent operating systems have been implemented by different programmers. Although, many different and independent developers are involved, the layered and systematic design of the TCP/IP protocol suite ensured long-term sustainability and proliferation.

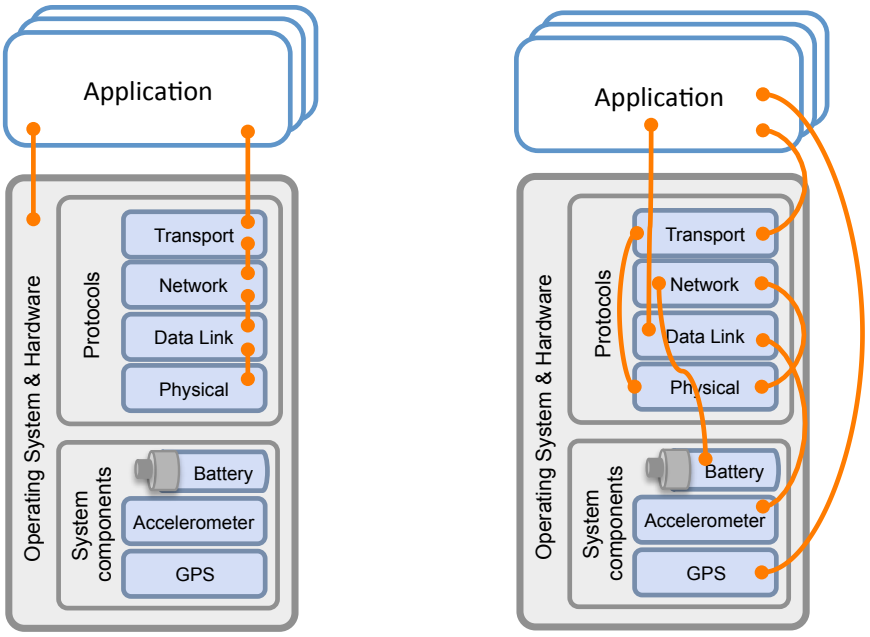
Nonetheless, there are some drawbacks of the layered design in wireless and mobile environments which we present in the following.

2.2 Cross-Layered Design

The layered protocol stack worked well in wired networks due to the static nature such as a fixed topology of devices, absence of mobility, not changing medium conditions, etc. But with the emergence of wireless and mobile communications several problems appeared [AXM04]. One problem, among others, can be attributed to the layered and self-contained nature of protocols. While this design paradigm offers software engineering advantages such as maintainability, at the same time it suffers from missing contextual knowledge. For example, the missing awareness of protocols about application needs, or in case of applications, the missing knowledge about network conditions [SM05] leads to a misinterpretation and misbehavior of the respective layer. In other words, one protocol layer has the information that other protocol layers need but the design paradigm prohibits the access. Due to this fact, protocols are limited in performing their task since they are forced to smartly use what their layer scope provides although the system could reveal more (context) information. Many works [SM05, Yu04, RI04a] have shown that using the so-called cross-layer design paradigm [LSS06, SRK03, SM05], which in fact breaks the conventional rules and allows the interaction across layers, improves performance.

The most cited problem in this regard is TCP's performance degradation in wireless networks which is a results of TCP's misinterpretation of packet losses as congestion even when packet losses occur on the wireless part of the communication [SRK03, Pen00]. Work [CRRP04, RI04a, SM05, FGA08] showed that using cross-layer information from lower layers helps to avoid such misinterpretations and keeps TCP sending rate which improves TCP performance in wireless environments [BSK95, BSAK95, CRRP04, SRK03]. However, this is only one example out of a wealth of many other cross-layer coordination examples from diverse fields of networking such as security [TS07, XWY06], handoff (mobility) [MA06, TYCH05, CC08], autonomic communication [RDN07b, Wód11], routing [IKSF04, YLA02, QK04], sensor networks [MVP06, AVA06, MLM+05], quality of service [KHZ+03, ZZ08, BPY09], and energy [KKT04, MHLS09, EBPC05]. Thus, without doubt the cross-layer design paradigm has demonstrated its potential to improve the system performance. But performance is not the only incentive to use the cross-layer design paradigm. For few research areas such as quality of service or cognitive radio it is in fact the only feasible concept, since in both cases, different layer information need to be provided to other layers for adequate adaptation. For example, in cognitive radio the selection of suitable wireless channels in the vicinity depends on the application (or user) requirements and available network conditions [MMJ99, TFDM07]. As a result, such a degree of adaptability necessitates versatile coordination and cooperation between all protocols which leads to the inevitable utilization of the cross-layer design paradigm.

Figure 2.2 sketches the allowed interactions for the traditional layered protocol stack in comparison to the potential interactions that the cross-layer design paradigm facilitates. The traditional layered protocol stack only allows applications to use few



(a) Traditional layered system.

(b) System following the cross-layer design paradigm.

Figure 2.2 Comparison of allowed interactions in traditional layered systems and the systems using the cross-layer design paradigm. The former strictly restricts the interactions to a degree where layers are only allowed to interact with adjacent layers and applications are only allowed to use APIs provided by the operating system. The later allows the interaction between any system component and protocol layers including applications.

specific interfaces, the so-called application programming interfaces (APIs). These APIs offer limited functionality. For example, an application is not able to interact with any protocol layer. Similarly with system components such as batteries and accelerometer which are also not easily accessible. Typically, the OS does not offer the interfaces for applications to directly access the information of interest. In contrast, the cross-layer design paradigm allows the interaction between any system component and protocol layers including applications.

2.2.1 Cross-Layer Design Definitions

The definition of cross-layer design[1] varies marginally in literature. For instance, [SRK03] gives the following definition which is very limited in its scope: "the physical and MAC layer knowledge of the wireless medium is shared with higher layers". In contrast, [VDS+05] provides a more holistic view: "an optimization with the objective to select a joint strategy across multiple OSI layers". Another very interesting definition is given in [SM05] as follows: "protocol design by the violation of a reference layered communication architecture is cross-layer design with respect to the

[1]Note that in the remainder of this work, we consider the terms cross-layer design, cross-layer optimization, cross-layer coordination algorithm and cross-layer interaction synonymously. Although we mainly use the term cross-layer coordination algorithm to emphasize the mechanism (algorithm) to realize the interaction across layers. Thus, we consider a cross-layer optimization as the result of a cross-layer coordination algorithm.

particular layered architecture". This definition leaves much space for interpretations, but includes the essential properties and draws a clear distinction between the traditional layered and cross-layer design. Although we agree with this definition, we want to add few comments especially regarding what cross-layer design is not.

Cross-layer design should not replace the layered architecture, it should be rather an evolutionary extension to preserve and reuse the well-established design methodologies used in the Internet. Ideally, cross-layer design should enhance the traditional layered architecture symbiotically by providing the ability to share information across all layers in addition to existing interfaces without losing the advantages that the layered architecture provides. In contrast to existing definitions, we also include (supplementary to the exchange between protocol layers) the exchange of information with system components such as batteries and sensors (i.e., accelerometer, gyroscope, and compass). The simple reason is that examples showed that protocols (e.g., position-based routing [MWH01]) and applications (e.g., indoor navigation using accelerometer and compass [LSVW11]) can benefit from system information and vice versa (e.g., energy efficient protocol design [MHLS09]).

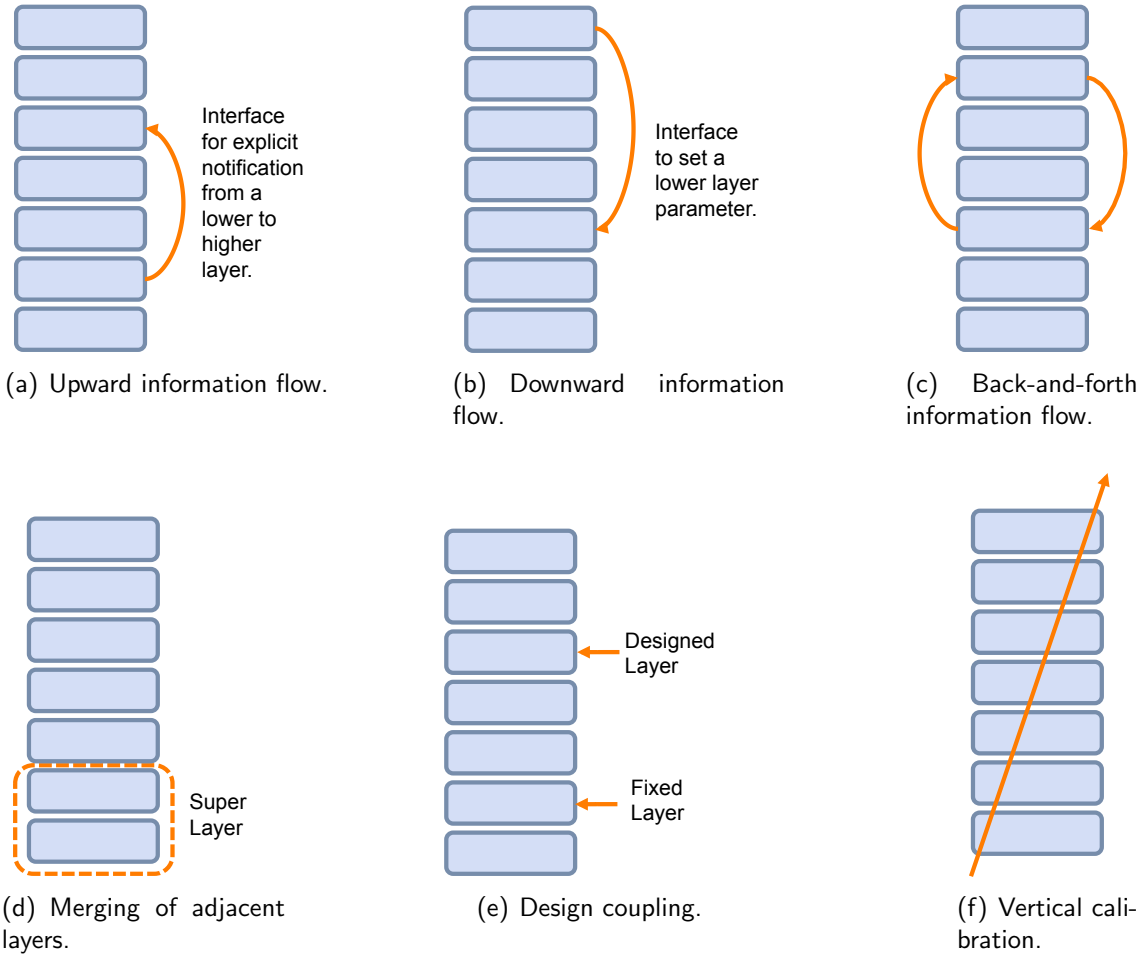
Furthermore, few papers consider cross-layer design as the exchange of information not only across protocols within a single system but also among nodes [WSNB06, RDN06]. For example, MAC layer information could be provided from one node to another node's TCP layer. From our viewpoint this is not cross-layer design. The exchange of information among nodes is rather the task of a protocol. Accordingly, we limit the scope of cross-layer design to only exchange information within a system. However, for network-wide optimizations we believe that a cooperative approach, thus, using a protocol, is the right way to go. How such a protocol could be designed is presented in Section 4.5.4 where we cooperatively detect the presence of a jammer by exchanging protocol information collected from several nodes.

To conclude, by breaking the rules inherited from traditional layered design, the cross-layer design paradigm has a great potential to improve performance in many fields of networking. But cross-layer design is not only a matter of performance, but also a very essential concept to provide a high degree of adaptability. Thus, it is not a a question of *whether* cross-layer design should be enabled but rather a question of *how* it should be established properly. In the following we turn our focus towards this question and begin with an overview about how information can be exchanged between protocols in general.

2.2.2 Information Exchange Alternatives

Based on investigated work on cross-layer coordination algorithms, Srivastava et al. categorized and suggested in [SM05] possible information exchange alternatives as shown in Figure 2.3. The authors also refer to these alternatives as layered architecture violations. Note that the categorized schemes are not exhaustive (i.e., there might exists different schemes) and they can be combined to realize more complex cross-layer interactions. For the categorization in Figure 2.3 Srivastava et al. used the seven layered ISO/OSI protocol stack for their sketch, but the categorization is similarly applicable to the five layered TCP/IP protocol suite.

However, Srivastava et al. differentiate between the following three types of exchange schemes:



(a) Upward information flow.

(b) Downward information flow.

(c) Back-and-forth information flow.

(d) Merging of adjacent layers.

(e) Design coupling.

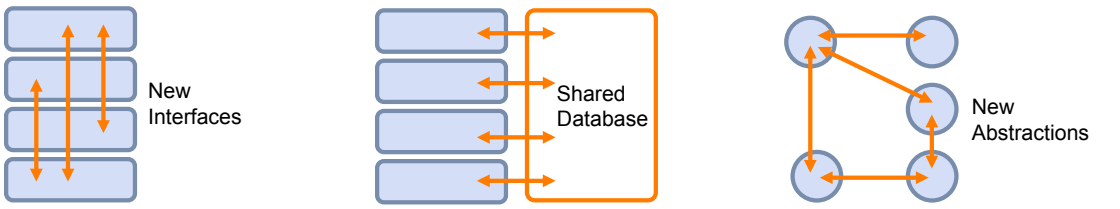
(f) Vertical calibration.

Figure 2.3 Cross-layer information exchange alternatives suggested by Srivastava et al. [SM05].

Creation of new interfaces: New interfaces are created with three different functionalities: (i) *upward information flow* (cf. Figure 2.3(a)) to provide lower layers hints about how application data should be treated, (ii) *downward information flow* (cf. Figure 2.3(b)) to notify higher layers about underlying network conditions, and (iii) *back-and-forth information flow* (cf. Figure 2.3(c)) to allow the collaboration between two different layers.

Merging of adjacent layers: In this scheme, two or several adjacent layers are merged resulting in a super layer that encapsulates all services and parameters as shown in Figure 2.3(d). Thus, this scheme does not require to provide new interfaces. The super layer can be interfaced by using the existing interfaces to the merged layers.

Design coupling without new interfaces: This scheme describes the coupling of layers at design time without creating additional interfaces. While designing one layer (e.g., designed layer in Figure 2.3(e)), the presence and specific mechanisms about another layer (e.g., fixed layer) are assumed and exploited. Thus, the independence of the designed layer is relinquished and the exchange of the fixed layer would directly influence the behavior of the designed layer.



(a) Direct communication between the different layers.

(b) A shared database.

(c) Clean slate: completely new abstractions.

Figure 2.4 Cross-layer architecture variations according to Srivastava et al. in [SM05].

Vertical calibration across layers: This scheme involves the adjustment of parameters that span across all layers as shown in Figure 2.3(f). It can be considered as a joint tuning of all layers to achieve better performance compared to independent adjustment of parameters in a specific layer which is the case when using the layered design. Srivastava et al. further differentiates between static and flexible vertical calibration. In the former case, parameters are adjusted at design time and left untouched afterwards. In the latter case, the adjustments are allowed at runtime which requires additional mechanisms to access and modify the values during protocol operation.

For particular examples of each violation, we refer the interested reader to [SM05]. In the following, we discuss further classifications of Srivastava et al. with respect to cross-layer architectures.

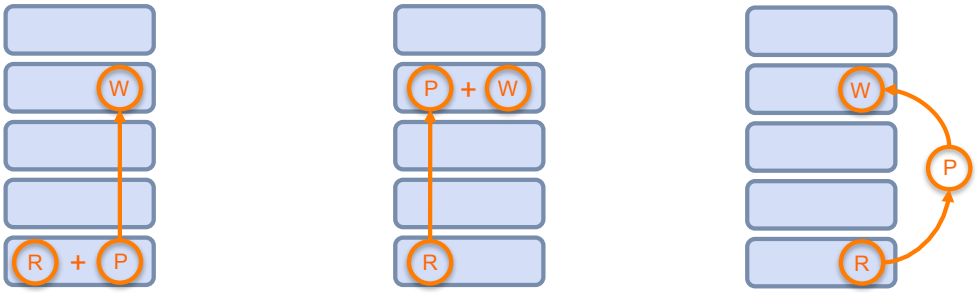
2.2.3 Architecture Classifications

Out of the investigated work, Srivastava et al. differentiate between three architecture types:

Direct communication between the different layers: In this architecture type each layer directly exchanges information with another layer as shown in Figure 2.4(a). One possible way is to create interfaces that can be called from another layer [WAR03]. Another approach such as in [WBLO99] attaches cross-layer information to packet headers that are accessed at another layer.

A shared database across the layers: In this category all layers provide their information to a variant of a database (plane) that can be accessed from all layers as shown in Figure 2.4(b). This layer can be considered as a new layer that allows several layers to store and retrieve data. Example architectures using a shared database are [SKC05, CMTG04, RI04b, GFTW06].

Completely new abstractions: The third case falls into the category of clean slate approaches which in fact completely reorganize and structure communication systems. For example, the proposal in [BFH03] organize protocol layers in so-called heaps as shown in Figure 2.5(c). Although such schemes enable totally new ways of information exchange and flexibility to organize functionality



(a) The access (cf. Ⓡ) and processing (cf. Ⓟ) of a cross-layer coordination is incorporated into a layer, based on the coordination algorithm another layer is manipulated (cf. Ⓦ).

(b) After accessing the information at one layer (cf. Ⓡ), it is provided to another layer where the information is processed (cf. Ⓟ) and the respective procedures are realized (cf. Ⓦ).

(c) The processing (cf. Ⓟ) of a cross-layer coordination is separated from the access (cf. Ⓡ) of information and the manipulation of a layer (cf. Ⓦ).

Figure 2.5 A cross-layer coordination consist of three steps: (i) accessing or reading information from a layer (indicated by Ⓡ), processing of the information, e.g., aggregation or computation of variables (indicated by Ⓟ), and (iii) manipulation of another layer by writing into a variable or triggering a function (indicated by Ⓦ). Each of the three steps can be performed at different places.

of protocols, we omit them in this thesis as we focus on practical solutions. Clean slate approaches require complete new implementations and also require changes to the core of the Internet which we do not want to change.

In the following sections, we will discuss the pros and cons of different architectures mainly belonging to the former two types.

2.2.4 Cross-Layer Information Processing

So far, a cross-layer coordination is considered as a single piece of algorithm. But technically, a cross-layer coordination algorithm consist of three steps. First, the access (read) of information available at a specific layer. Subsequently, this information is used for aggregation or computation (process). Finally, based on the outcome of the processing a pre-specified variable residing in a layer is modified (write) or a specific function is triggered.

However, each of the steps could reside at different places in the system as illustrated in Figure 2.5. In the first case, the access to information and its processing could be integrated into the layer being accessed. The processed information is afterwards provided to the layer which is intended for modification as shown in Figure 2.5(a). In the second case, the accessed information from a layer is directly provided to another one before processing. The processing is integrated into the layer being intended for modification as shown in Figure 2.5(b). Finally, the processing is completely separated from the access at one layer and the modification at another layer as shown in Figure 2.5(c). Later in Section 2.2.4.2 we will further elaborate the placement of processing as it is dependent on the target system and it might influence the system behavior and performance, but beforehand we present the different processing alternatives which have an impact on the system performance.

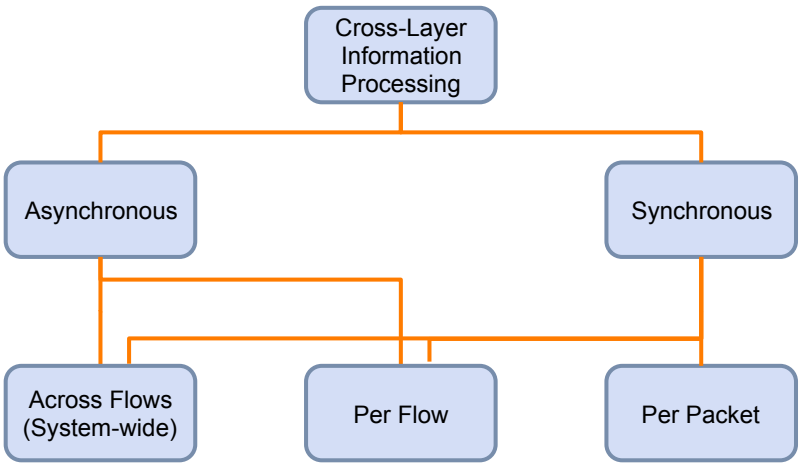


Figure 2.6 Synchronous versus asynchronous information processing according to Raisinghani in [Rai06].

2.2.4.1 Synchronous and Asynchronous Processing

Dependent on the interplay between access, processing, and manipulation, the performance of the system could be influenced. Raisinghani differentiates in [RI06, Rai06] between synchronous and asynchronous processing. In the former case, a protocol layer's processing is coupled with the processing of the cross-layer coordination, i.e., whenever a layer is accessed by an cross-layer coordination, it is not allowed to proceed its regular execution as long as the execution of the cross-layer coordination is finished. In the latter case, layers and cross-layer coordination algorithms are decoupled and executed in parallel.

Furthermore, cross-layer coordination algorithms are not only based on information from protocol layers, but also from their processed packets or packet flows. In this conjunction, Raisghani [Rai06] differentiates between the following three cases: (1) per packet, i.e., a cross-layer coordination accesses each passing packet, (2) per flow, i.e., a cross-layer coordination manipulates a certain flow (established connection), or (3) across flows, i.e., a system-wide adaptation for all flows. The relationship between synchronous and asynchronous packet processing and the three different cases are shown in Figure 2.6. The per-packet and across-flows cases can be performed synchronously and asynchronously, however, the per-packet case can only be performed synchronously since the adaptation needs to be performed as soon as the packet is being processed at a layer. Thus, when there is a need to treat each packet specifically, this introduces a per packet processing delay in the whole protocol stack.

2.2.4.2 User and Kernel Space Separation

Although Srivastava et al. give a well-elaborated overview about the interaction possibilities of cross-layer design proposals and how architectures can be designed, the presented classifications are rather abstract without considering specific properties of the operating system such as the separation of memory between user and kernel space. In previous sections we motivated the three essential steps of a cross-layer coordination and discussed how these three steps can be realized at different places

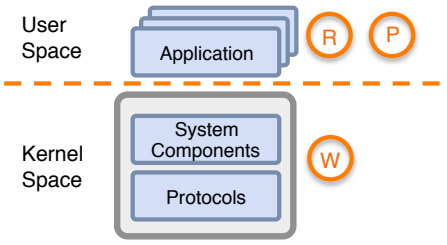
of the layers or the system. But dependent on the memory management, the realization can vary significantly. For example, in a operating system for a sensor node such as TinyOS, the access to protocol layers from the application layer and vice versa is rather simple as all layers including the applications run in a single dedicated environment from where the whole memory is accessible and thus all layers.

In contrast, in many other today's operating systems, the memory is subdivided into two differently privileged parts namely kernel and user space. This thesis focuses more on this complex case of user and kernel space separation. The separation mainly serves for robustness and security reasons. The kernel space contains the core of the operating system; amongst others, it also contains the protocol layers and controls system components. This part of the memory cannot be directly accessed by the processes such as from applications that run in the restricted user space. The access of the kernel space form the user space is limited to few so-called system calls. As a result of this restrictions, applications are constrained in influencing the operating system performance and stability.

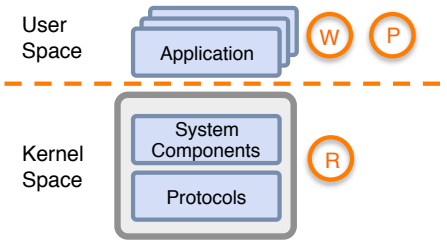
With respect to the cross-layer design paradigm, this separation between kernel and user space leads to several challenges for developers which are sketched in Figure 2.7. Dependent on the realization of the cross-layer coordination, the access to information could reside in the kernel space (case (b) – (d) in Figure 2.7) which contains protocols and system components or in the user space which contains the applications (case (a) and (e) in Figure 2.7). Similarly, the manipulation could also take place in kernel space (case (a),(c) and (e) in Figure 2.7) or in user space (case (b)–(f) in Figure 2.7).

However, after accessing the information at one layer and before the manipulation of another layer, the accessed information is processed, e.g., aggregated or computations are conducted. The developer could opt for realizing the processing in the user space (case (a) – (c) in Figure 2.7) or the kernel space (case (d) – (f) in Figure 2.7), whereby each decision offers different advantages and disadvantages. For instance, programming in kernel space is considered as being more complex for several reasons such as the confrontation with a bulk of undocumented code, programming errors, tedious memory management, less availability of libraries for functionality reuse, etc. Moreover, modification can also require a lot of effort due to compiling, patching, and system reboot. Thus, in cases where the processing may require adjustments, development of the cross-layer coordination may become tedious. In contrast, in user space implementation testing is much simpler since code or libraries are usually better documented and compilation does not require a system reboot. However, this advantage possibly comes along at a high cost of performance or responsiveness when much information has to be passed between kernel and user space which is referred to as a *context switch*. A context switch requires copying information from one part of the memory to another which is time- and CPU-consuming. High amounts of context switches and the size of copied data could lead to significant performance drops. Thus, keeping the size and the amount of exchanged information between kernel and user space as low as possible is reasonable.

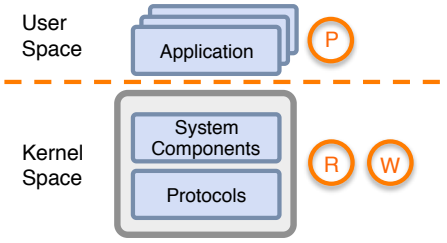
On the other hand, in case of updated values in one of the spaces (which have to be passed to another space) could reach the other space too late leading to false reactions. For example, due to delayed information passing, layer two is not initiating a handoff at the right time although the signal strength is significantly below a



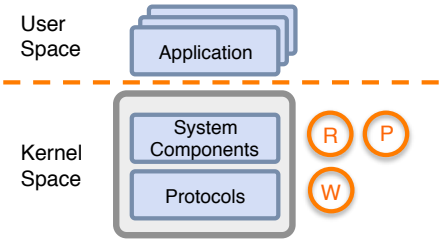
(a) **Processing in user space, downward information passing:** After application information is accessed, its processing is also conducted in user space for more adaptability. Based on the processing, functions need to be triggered in kernel space to manipulate protocol or system behavior. **Problem:** Could lead to many context switches or late manipulation.



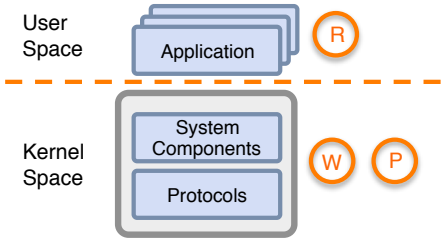
(b) **Processing in user space, upward information passing:** After system information is accessed, it needs to be passed to the user space for flexible processing which is then used to modify application behavior. **Problem:** Could lead to many context switches or late triggering of manipulation.



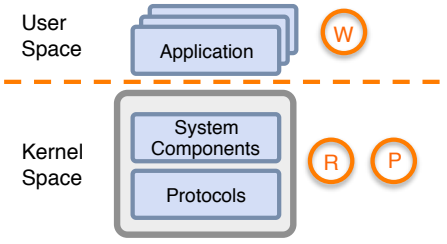
(c) **Processing in user space, application not involved:** Moving the processing to user space allows a developer more adaptability which also avoids patching and rebooting of the system. **Problem:** Could lead to many context switches that decrease system performance.



(d) **Each step in kernel and if application not involved:** As each step is completely running in the kernel, passing information to user space not necessary. **Problem:** kernel programming complex, programming faults lead to system crash, changes of processing likely requires patching and reboot of system.



(e) **Processing in kernel space, downward information passing:** application information residing in user space needs to be passed to kernel space for fast processing and protocol behavior manipulation. **Problem:** kernel programming complex, system crash possible, changes of processing likely requires patching and reboot of system.



(f) **Processing in kernel space, upward information passing:** After system information is accessed it is subsequently processed and then passed to the user space to modify application behavior. **Problem:** kernel programming complex, system crash possible, changes of processing likely requires patching and reboot of system.

Figure 2.7 Alternatives to realize cross-layer coordinations in architectures which are subject to user and kernel space separation. Reading or accessing layer information is indicated by Ⓡ, processing, aggregation or computation of information is indicated by Ⓟ, and writing, manipulation of a layer variable or function triggering is indicated by Ⓦ.

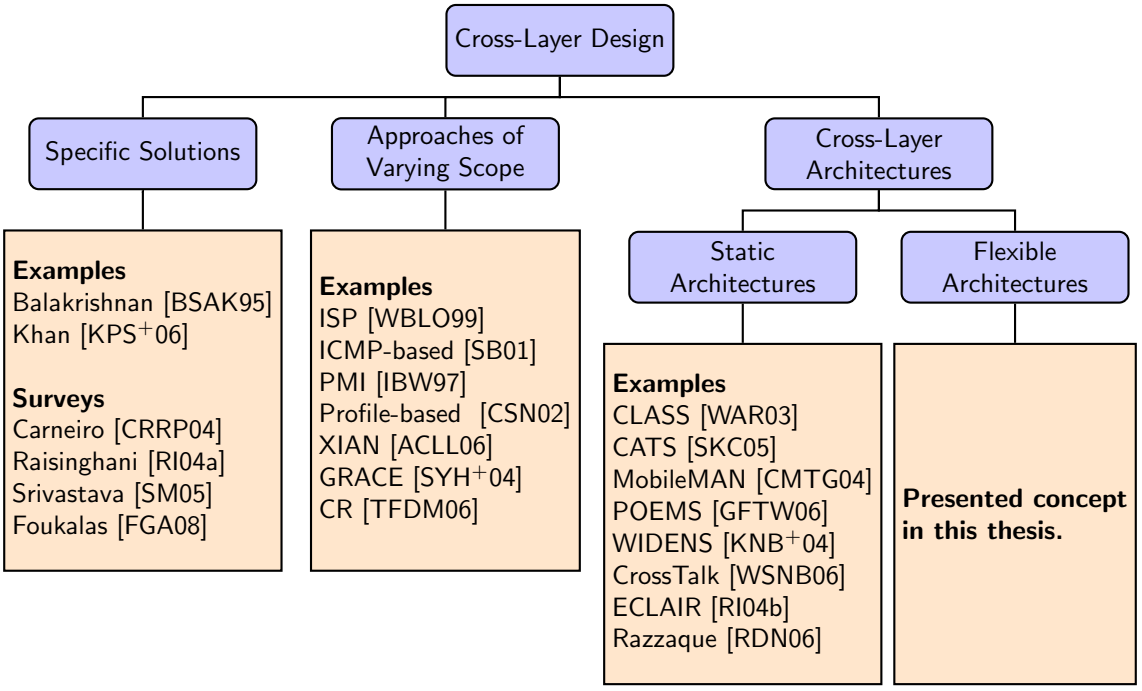


Figure 2.8 An overview about investigations in the field of cross-layer design.

predefined threshold. Or in the other direction, where layer two decides to initiate a handoff although the values are already stable again. Another problem regarding the information-passing between the kernel and user space concerns the interface. Typically, operating systems offer only few limited interfaces which do not easily allow to pass any kind of information. As a result, a developer needs deep expert knowledge and the will to put significant effort into realizing its information exchange scheme between the kernel and the user space.

To conclude, a developers faces several additional challenges when memory is separated into kernel and user space. Especially, the placement of the processing leads to a tradeoff between performance and flexibility. In this thesis, we show an approach where we dissolve this tradeoff and provide a solute where we keep the flexibility while offering an efficient realization of cross-layer coordination algorithms. But before presenting our approach, we discuss related work on cross-layer design and highlight the missing key features to successfully harness the cross-layer design paradigm.

2.3 Related Work

Many proposals utilizing the cross-layer design paradigm is available in the literature. As illustrated in Figure 2.8, we classify these proposals into the categories *specific cross-layer solutions*, *approaches of varying scope*, and *cross-layer architectures* which we describe in the following in more detail.

2.3.1 Specific Cross-Layer Solutions

A plethora of *specific cross-layer solutions* have been proposed that focus only on a certain behavior or performance improvement. These proposals are tailor-made and solution-oriented implementations which mainly target at realizing a single specific coordination idea. While in some cases the behavior of only a single layer is optimized by using information from other layers such as improving TCP by using link layer information [BSK95, BSAK95], in other cases two or more layers are are jointly optimized such as in [KPS+06] where quality of service adaptations are jointly optimized at multiple layers (application, data link, and physical layer) to improve video streaming quality. Nevertheless, from a system designer's perspective these solutions belong to the same problem category as none of these solutions follow software engineering principles. Therefore, these solutions are typically hard to maintain, to extend and to understand. Gaining a performance or behavior improvement by using the cross-layer design paradigm should not be at the cost of losing the well-established software engineering advantages that come along with the layered protocol stack design. Ideally, these two paradigms should be built in a synergetic fashion and advantages from both should be carried over. We omit the description of particular specific cross-layer solutions as it is does not provide more understanding from a system designer's point of view, but we refer the interested reader to the following surveys [CRRP04, RI04a, SM05, FGA08] that give several examples layer by layer.

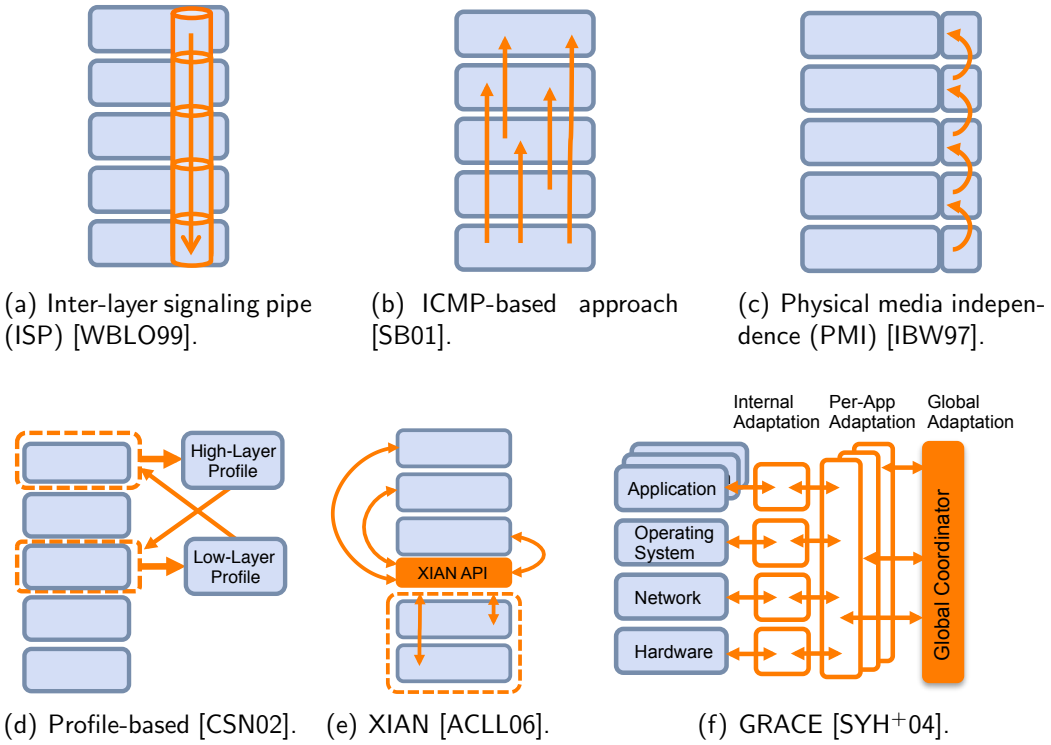
To conclude, specific cross-layer solutions are tailor-made implementations which mainly focus on a single specific coordination idea and typically not implemented in a generic way to realize further cross-layer coordination algorithms.

2.3.2 Approaches of Varying Scope

Another category of proposals that use the cross-layer design paradigm are the *approaches of varying scope*. These solutions are either too limited in their scope, e.g., only allow the signaling of information in one direction (from lower layers to upper layers but not vice versa), or have a very broad scope where cross-layer design is only one aspect and the focus is on specific scenarios or algorithms operating on the information gathered with cross-layer design.

For instance, **inter-layer signaling pipe (ISP)** [WBLO99] is a proposal with a limited scope which tries to reuse or exploit established system mechanisms. ISP utilizes packet headers to provide cross-layer feedback from upper layers to lower layers as indicated by the pipe in Figure 2.9(a). This approach is implemented in a simulator. Unfortunately the authors did not mention how applications, which in fact should not know about structures such as (network layer) packet headers, provide their information to lower layers. Another problem of this approach is the layer by layer processing which could be inefficient and lead to outdated information since each layer has to inspect and process packet headers. For example, the link state information could be not up-to-date anymore when reaching the higher layers and leading to inappropriate reactions.

Similarly, Sudame et al. [SB01] suggested to exploit **Internet control message protocol (ICMP)** messages to provide feedback from lower layers to upper layers



(a) Inter-layer signaling pipe (ISP) [WBLO99].
(b) ICMP-based approach [SB01].
(c) Physical media independence (PMI) [IBW97].
(d) Profile-based [CSN02].
(e) XIAN [ACLL06].
(f) GRACE [SYH+04].

Figure 2.9 Overview of semi-manufactured approaches.

as illustrated in Figure 2.9(b). Originally, the ICMP protocol resides in the network layer and was designed to gain feedback information from network devices such as routers that are on the path to a destination host. Note that ICMP messages, in their original version, reach only the network layer. But in contrast to the ISP approach, in the real-world implementation of the ICMP-based approach an appropriate handler is implemented to pass the encapsulated information in ICMP messages to a specified socket used by an application.

Another approach of passing information upwards is **physical media independence (PMI)** [IBW97]. In this approach each layer is extended by so-called adaptation modules which pass information to its adjacent upper adaptation module as sketched in Figure 2.9(c). Again, information is inspected and processed layer by layer leading to the aforementioned weaknesses.

A solution that is even more limited in scope and applicability is the **profile-based approach** [CSN02] that allows only the exchange between a middleware and the routing layer as shown in Figure2.9(d). The exchange of information between these two entities is performed in form of so-called system profiles. For example, the routing layer provides information such as node's location and movement pattern to the middleware, for instance, to predict future connectivity. On contrary, the middleware provides priority information of applications to the routing layer for scheduling purposes. In a nutshell, only two layers are involved and the variability of exchangeable parameters is limited.

XIAN [ACLL06] is another approach that is limited in its ability to support the interaction between all layers. XIAN is implement for the Linux operating system and offers an API to only WiFi-related information (in particular access to the MadWifi driver) but the API is accessible from all layers as illustrated in Figure2.9(e).

At the user space applications can use an API to XIAN and similarly protocols residing in the kernel space have also an available interface for interaction. Around 180 different WiFi-related information, which are in fact provided by the MadWifi driver, are accessible from all layers. Nonetheless, the scope to enable the access to parameters is limited to only layer one and two. Thus, the interaction between remaining layers such as between application and network layer is not possible with XIAN.

The research project **Global Resource Adaptation through CoopEration (GRACE)** [SYH+04] at the University of Illinois has a much broader scope. GRACE targets at a system-wide resource adaptation of applications, operating system, network and hardware (referred to as system layers) through cleanly defined interfaces as illustrated in Figure 2.9(f). To achieve this, GRACE uses a three level hierarchical adaptation scheme. The finest granularity is the internal adaptation that only adapts a single system layer or application. For instance, it is invoked per packet or after every hundred instructions. The next level is the per-application adaptation which considers one specific application and adapts all other system layers accordingly. It is invoked per job basis (not clearly defined). The coarsest granularity is the global adaptation which considers all applications and system layers. It could be triggered, for example, when an application joins or leaves the system. However, the focus of GRACE is the application-driven and system-wide optimization of few specific parameters such as energy usage, CPU and time and bandwidth. Therefore, it has very specific goals which limit its applicability. Moreover, although applications and the network can exchange information, signaling within the (system layer) network and thus across protocols is not feasible.

Another investigation ranging across the layers, amongst others (e.g., security and quality of service), is the **cognitive radio (CR)** research field. The term cognitive radio was originally introduced by Joe Mitola in [Mit00] as "wireless personal digital assistants (PDAs) and related networks that are sufficiently computationally intelligent about radio resources and related computer-to-computer communications (a) to detect user communications needs as a function of use context, and (b) to provide radio resources and wireless services most appropriate to those needs." After this fundamental work, the whole cognitive radio community was driven by tunable parameters in software defined radios and the goal to equip them with more "intelligence", particularly with abilities such as learning, self-awareness and adaptability. Inspired by this ability to build adaptive and cognitive radios, the scope has been continuously extended to cognitive radio networks which not only consider the network-wide optimization (and thus the cooperation between nodes) but also the optimization of each involved system and, hence, the cooperation across all protocols and applications. For instance, Ryan et al. [TFDM06] consider cross-layer design as an essential and inherent part of cognitive radio networks since it indirectly share information that is not available externally in the strict layered architecture. Ryan et al. also argue that another key feature of cognitive radio networks but not covered by cross-layer design is the ability of learning to improve future behavior. To achieve this, many concepts from artificial intelligence are utilized. However, proposals providing an architecture are the End-to-End Reconfigurability Project II (E^2R II) [BMS+06], the Value-Chain Research (CTVR) [SDN06], the m@ANGEL platform [DSB+06], and the cognitive resource manager (CRM) [MPRW06]. According to [TFDM06] these proposals are focused on particular applications (such as

4G or wireless), implementations such as cognitive mechanisms or associated APIs, or specific problems such as mobility or management. To summarize, cognitive radio networks differ from cross-layer design in their goals and scope [TFDM06]. In particular, while cross-layer design is a node centric instrument to access and optimize protocols and application (local scope), the observations of cognitive radio networks consider multiple nodes in the network and their optimization (network-wide scope) [TFDM06]. We concentrate in this thesis only on the part of cross-layer design and not on the cognitive process belonging to cognitive radio networks.

To conclude, approaches of varying scope either have a too limited or a too broad scope. In the latter case, cross-layer design is considered only as one integrated part to realize a bigger goal (e.g., network-wide optimization). In the former case, the approaches only allow signaling between few layers or in one direction. Accordingly, the realization of cross-layer interactions between any layers is not feasible. But this is an essential prerequisite of what we expect from a full-fledged cross-layer architecture. In the following, we present further requirements and based on them rate popular cross-layer architectures.

2.3.3 Cross-Layer Architectures

In recent years, a number of *cross-layer architectures* have been proposed that allow to run several (specific) cross-layer coordination algorithms and facilitate signaling across all layers and in both directions, i.e., any-to-any layer signaling. We distinguish between static and flexible architectures. In *static cross-layer architectures* a cross-layer coordination are realized offline (i.e., at compile time) and are deeply embedded within the operating system. In other words, the cross-layer coordination algorithms are hard-wired with the remaining system and always active even if not always necessary. For example, energy-saving cross-layer coordination algorithms might not be necessary when plugged into power. Such cross-layer coordination algorithms should be adaptable (add, remove, modify) at runtime.

This is tackled by *flexible cross-layer architectures* that allow the adaptation of running cross-layer coordination algorithms at runtime, for instance, when conditions are satisfied. Later we will give more details about flexible cross-layer architectures. But beforehand we want to review static architectures with respect to how well they support developers in case of user and kernel space separation. Particularly, we investigate how well the following set of requirements are satisfied by these architectures.

Applicability to real systems: Although simulation is a well-established methodology to validate a concept, it is often too abstract and does not take into account important effects that are caused by an operating system such as context switches, scheduling and buffering. Therefore, from a system developer's point of view we consider a conceptual design or even its validation in a simulator as insufficient since such architectures neglect user and kernel space separation challenges (as discussed in Section 2.2.4.2) and applicability to real-world scenarios which are essential for real use and proliferation.

Any-to-many layer signaling: The signaling of information should not only be enabled from one layer to any other, but also from one layer to many other layers.

For example, not only the transport layer is interested in link layer information to adapt its congestion control mechanism, but also a VoIP application could benefit from this information and switch its codec adequately. Thus, an architecture needs to support the sharing of cross-layer information with many other layers. Ideally, not only a polling scheme should be supported but also an event-based scheme to allow multiple layers the reaction to sudden events such a significant and sudden link quality changes, for instance, to initiate a layer two and three handover.

Maintainability: If cross-layer coordination algorithms are established arbitrarily and unsystematically, this will likely result in unbridled cross-layer design (i.e., the so-called spaghetti design [KKTC05]) which leads to unmanageable and complex interdependencies in the code. Thus, an architecture should support and steer the developer to systematically establish its cross-layer coordination algorithms to improve code maintainability and understanding. Moreover, to keep maintainability of the protocol stack, the cross-layer architecture should as less as possible modify the protocol stack or protocols, respectively.

Flexibility and extensibility: The architecture should offer the flexibility to support a wide range of cross-layer coordination algorithms. Therefore, a developer needs support in accessing the desired information in the whole protocol stack and the ability to modify the processing of the accessed information. Especially when adding novel protocols to the system, an architecture should support the developer in extending the set of accessors to the novel protocols. At the same time, adding new cross-layer coordination algorithms into the system should not force a developer to subsequent amendments of not directly involved protocols. For instance, when adding a cross-layer interaction that establishes signaling between layer two and four, this should not enforce the modification of any of the remaining layers.

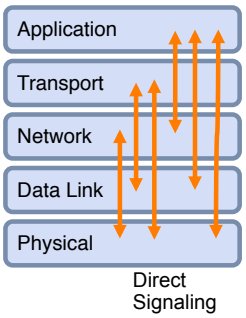
Based on these requirements, we review the proposed cross-layer architectures in the following.

2.3.3.1 Static Cross-Layer Architectures

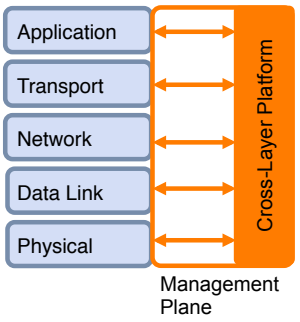
We present and rate the following popular static cross-layer architectures.

Cross-Layer Signaling Shortcuts (CLASS) [WAR03] enables direct signaling between all layers by message passing as illustrated in Figure 2.10(a). CLASS is only a conceptual design which distinguishes between the two messaging types internal and external. External messages are standard protocol packets. For internal messages, which are in fact the cross-layer interactions, the authors suggest the use of the ICMP-based approach [SB01]. But as discussed earlier, this scheme allows only the upward signaling and as the authors do not go into much detail, many open questions remain regarding the realization of the signaling. From the available information we derive that CLASS allows to run several cross-layer coordination algorithms. Furthermore, since the authors suggest the use of a destination address field, we assume that any-to-(m)any layer signaling, i.e., addressing several layers at once, is theoretically possible. Unfortunately, due to missing implementation details, we rate maintainability, flexibility and extensibility requirements as not supported.

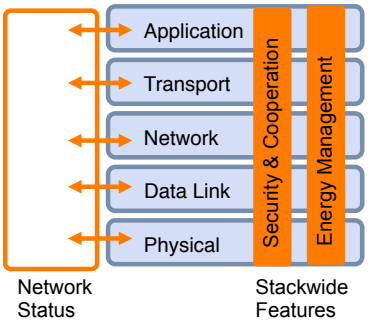
Cross-layer Approach To Self-healing (CATS) [SKC05] is a shared database-based approach that allows to establish cross-layer interactions by using a so-called



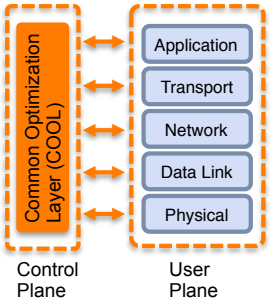
(a) **CLASS** [WAR03]: Direct signaling between all layers by message passing (ICMP messages).



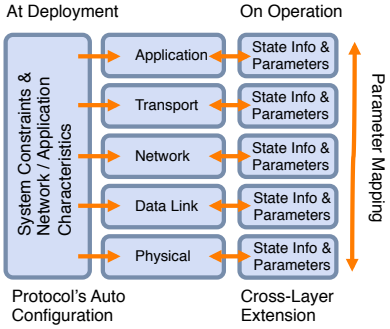
(b) **CATS** [SKC05]: The cross-layer platform stores the cross-layer information and makes it accessible from all layers.



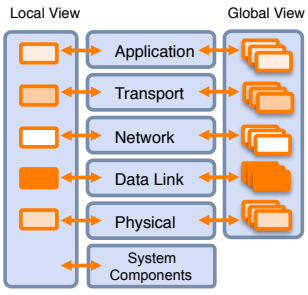
(c) **MobileMAN** [CMTG04]: Each layer can store protocol information in the network status and make it accessible to other layers. It also considers the coordination of security and energy management functionalities



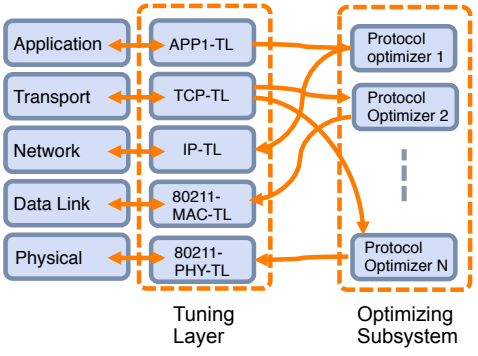
(d) **POEM** [GFTW06]: A shared-database approach that consists of a control plane containing the common optimization layer (COOL) that in fact allows the cross-layer interaction between the layers.



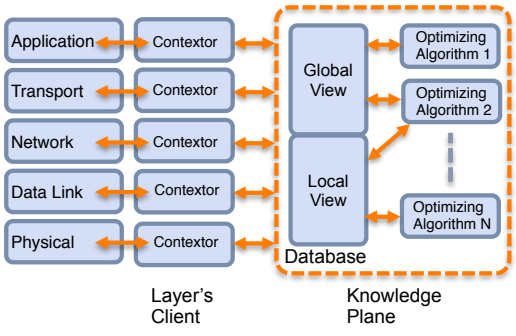
(e) **WIDENS** [KNB+04]: At deployment, it allows the reconfiguration of protocols. During operation, it extends each protocol with a cross-layer extension. Each extension contains the state information and parameter values of the layer which is only shared with adjacent layers.



(f) **CrossTalk** [WSNB06]: Consists of two data planes. Local view contains information from each layer and enables the actual cross-layer interactions. Global view represents the disseminated information of the local view with neighbors.



(g) **ECLAIR** [RI04b]: Specific tuning layer per layer that provides access to protocol data structures. The optimizing subsystem consist of several protocol optimizers that contains the algorithms and data structures to realize the actual cross-layer coordination algorithms. Protocol optimizers can call functions of different tuning layers for reading and modifying protocols.



(h) **Razzaque** [RDN06] et al.: Is a mixture of ECALIR and CrossTalk. Similar to ECLAIR's tuning layer, contexttors access protocol information which are accessed by the knowledge plane. The knowledge plane contains a database similar to CrossTalk, i.e., with a local and global view. The knowledge plane also includes the optimizing algorithms similar to ECLAIR.

Figure 2.10 Overview of static cross-layer architectures.

management plane as shown in Figure 2.10(b). The management plane collects information from all layers and provides them to the cross-layer platform which stores the information and makes it accessible from all layers. Depending on how the cross-layer platform offers the ability to distribute the cross-layer information to all layers, any-to-(m)any layer signaling might be achievable. Similar to CLASS, CATS is also a conceptual design without any implementation. From the available information, CATS seems to have a monolithic architecture, i.e., the management plane mechanisms are jointly designed with the protocol layers leading in the respective case to difficult maintainability. Since no information about interfaces between the management plane and the protocol stack are available, we rate flexibility and extensibility as not supported.

MobileMAN [CMTG04] is an EU funded project that targets to define and develop a metropolitan area and self-organizing wireless ad-hoc network. The proposed architecture is a shared-database approach where each layer can store protocol information in the so-called network status and make it accessible to other layers in a unified fashion as illustrated in Figure 2.10(c). MobileMAN also considers the coordination of security and energy management functionalities with the help of the network status since these functionalities require cross-layer interactions by nature [CMTG04]. In order to interact with the network status, protocols need to implement specific interfaces. These interfaces define the way how protocols are allowed to interact with the network status and accordingly with each other. In the project webpage an implementation of the architecture is not available. Implementation details are not given in [CMTG04], it is only mentioned that the implementation of the interfaces are placed besides normal protocol processing. MobileMAN requires a couple of protocol modifications to enable database interactions via the unified interfaces. The interfaces are designed such that the replacement of modified protocols with their original is possible, if necessary. This should also allow a decoupled integration of novel protocols from other protocol layers and thus supporting flexibility and extensibility. Details about how the interfaces could look like are not provided in detail making it hard to rate the degree of interdependencies between the protocols and the cross-layer coordination algorithms. As a result, we rate flexibility, extensibility and maintainability of the architecture as average.

Performance-Oriented referencE Model (POEM) [GFTW06] is a proposal towards a self-optimizing protocol stack for autonomic communication. The proposed architecture is a shared-database approach that consists of a control plane containing the common optimization layer (COOL) that in fact allows the cross-layer interaction between the layers as illustrated in Figure 2.10(d). The user plane contains the protocol stack. It is intended that the user plane works independent of the control plane. The interaction between the two planes are performed through so-called common operation interfaces. Unfortunately, the work neither provides details about how the interfaces could be implemented without effecting original protocol behavior nor describes how to design a cross-layer coordination in particular. Due to too the abstract presentation, it is hard to decide which of our requirements are satisfied.

WIreless DEployable Network System (WIDENS) [KNB+04] was an EU funded project targeting on easily deployable ad-hoc networks for public safety organizations. At deployment the proposed architecture allows the reconfiguration of

protocols to adjust their functionality to available system constraints and environments as shown in Figure 2.10(e). During operation, the architecture extends each protocol with a so-called cross-layer extension. Each layer's extension contains the state information and parameter values of the layer. The information available at the extension is mapped only to adjacent layers. In particular, each layer has to individually process and pass information to its adjacent layer. Thus, cross-layer interactions are controlled via adjacent layers. The authors argue that each layer can benefit and decide how to react upon receiving the cross-layer information. This might lead to delayed dissemination of cross-layer information since each layer has to inspect and process information and then prepare and pass it to the adjacent layer.

Furthermore, a developer has to specify what to do upon receiving such information. This implies that a developer knows which information is necessary and how to adapt a certain layer and this in turn makes the involvement of all layers and the layer by layer processing superfluous. However, it seems that new protocols are not easy to integrate into the existing protocol stack as it requires extensive changes to make it work with the remaining part. The tight coupling with the adjacent layers decreases maintainability, flexibility, and extensibility. Implementation details are not given in [KNB+04] and the project web page is also not available to verify the latest available state.

CrossTalk [WSNB06] is a shared-database approach that consists of two data planes, local and global view, as shown in Figure 2.10(f). While the local view is a shared database that consists of information from each layer and enables the actual cross-layer interactions, the global view represents the disseminated information of the local view with neighbor nodes for network-wide optimization. For the global view CrossTalk provides a data dissemination procedure which adds the local view information to outgoing packets. We believe that exchanging information with other nodes is a nice to have feature, but not a must-have since the exchange of information with other nodes in the network is not a real challenge and is in fact the core property of a protocol. CrossTalk is implemented for the network simulator NS-2 and a couple of examples are applied to demonstrate and validate its effectiveness. Implementation details about the interfaces between the layers and the local view are missing, making the direct applicability to a real system questionable. Due to the lack of information, we rate maintainability, flexibility and extensibility of the architecture as poor.

ECLAIR [RI04b] is the most advanced cross-layering architecture that is composed of a modular database (referred to as optimizing subsystem) containing several coordination algorithms and interfaces to access protocols (the so-called tuning layer) as shown in Figure 2.10(g). In particular, there exists a specific tuning layer per layer that provides the access to protocol data structures (with the goal of marginal protocol modification). This also facilitates extensibility of ECLAIR to support novel protocols, since only few modifications to the respective protocol and the implementation of its tuning layer are needed to establish cross-layer coordination algorithms. For portability, a tuning layer is further subdivided into a specific part towards the protocol and a generic part towards the optimizing subsystem. Since each operating system has a different implementation of protocols, the specific part is customized to the operating system needs and enables read and write

requests to protocol information (with less infiltration). The generic part facilitates protocol stack abstraction and platform independence and is accessible from the optimizing subsystem. The optimizing subsystem and the protocol stack work asynchronously. Hence, the protocol stack processing can not be blocked or slowed down by a cross-layer coordination. ECLAIR support portability only on a conceptual level since the architecture is only validated on a Linux OS. But from all known approaches ECLAIR is, to the best of our knowledge, the only approach that is implemented for and validated on a real system with user and kernel space separation. Unfortunately, the code is neither open source nor was available after request. However, the optimizing subsystem consist of several so-called protocol optimizers that contains the algorithms and data structures to realize the actual cross-layer coordination algorithms. The protocol optimizers can register at several tuning layers to get notifications whenever certain events occur. Similarly protocol optimizers can call functions of different tuning layers for reading and modifying protocols. Thus, any-to-(m)any layer signaling is possible. Interestingly, single protocol optimizers or even the whole optimizing subsystem can be enabled or disabled. This fact and the modularization of protocol optimizers facilities a high degree of maintainability. Unfortunately, this feature is only available at compile time. Therefore, activation or deactivation of protocol optimizers dependent on contextual changes at runtime is not feasible.

Razzaque et al. [**RDN06**] proposed a shared-database approach that is a mixture of ECLAIR and CrossTalk. Similar to ECLAIR, the information residing in layers are accessed by so-called contexttors which can be compared with the tuning layers in ECLAIR as shown in Figure 2.10(h). However, there is one interesting claim where the authors argue that there is no modification required to the existing protocol stack. This is probably due to the fact that the authors (in [RDN07a]) implemented their approach for the NS-2 network simulator where information is inherently easily accessible. But real protocols of popular operating systems typically reside in the kernel space and are by nature self-contained and difficult to access as discussed in Section 2.2.4.2. Therefore, the goal of a cross-layer architecture should be to modify protocols as little as possible since even modifying a single variable could already lead to significant behavior variations. However, the accessed information is provided to the so-called knowledge plane which contains a database similar to CrossTalk, i.e., with a local and global view. Moreover, the knowledge plane also contains the optimizing algorithms that similar to ECLAIR operate on the interfaces provided by the contexttor or tuning layer, respectively. Razzaque et al. describe the knowledge plane as an intelligent database that allows the manipulation and inference of data (including uncertain reasoning). In contrast to CrossTalk, the global information is not received by piggybacking information to outgoing packets, instead a gossiping service is built at the application layer to collect information from other nodes.

To conclude, we presented several static cross-layer architectures that facilitate the signaling between any layers and allow to run several specific cross-layer coordination algorithms. Most of them are only described on a conceptual level while few are validated with simulation. ECLAIR is the most advanced architecture as it is implemented for a real OS, offers a generic interface to facilitate platform independence, and uses a modular design where whole cross-layer coordination algorithms can be activated or deactivated. Unfortunately, this feature is only available at compile-

Architecture	CLASS [WAR03]	CATS [SKC05]	MobileMAN [CMTG04]	POEM [GFTW06]	WIDENS [KNB+04]	CrossTalk [WSNB06]	ECLAIR [RI04b]	Razzaque [RDN06]	CRAWLER
Static Architecture Requirements									
Possibility to run multiple coordination algorithms	✔	✔	✔	✔	✔	✔	✔	✔	✔
Any-to-any layer signaling	✔	✔	✔	✔	✔	✔	✔	✔	✔
Applicability to real systems	✗	✗	Ø	✗	✗	Ø	✔	Ø	✔
Any-to-many layer signaling	✗	Ø	✔	Ø	Ø	✔	✔	✔	✔
Maintainability	✗	✗	Ø	✗	✗	✗	✔	✔	✔
Flexibility and extensibility	✗	✗	Ø	✗	✗	✗	✔	✔	✔
Flexible Architecture Requirements									
System detail abstraction	✗	✗	✗	✗	✗	✗	✗	✗	✔
Runtime flexibility & extensibility	✗	✗	✗	✗	✗	✗	✗	✗	✔
Application support	✗	✗	✗	✗	✗	✗	✗	✗	✔
Context adaptation	✗	✗	✗	✗	✗	✗	✗	✗	✔
Handling support for multiple coordination algorithms	✗	✗	✗	✗	✗	✗	✗	✗	✔
Evaluation support	✗	✗	✗	✗	✗	✗	✗	✗	✔

Table 2.1 Comparison of cross-layer architectures: ✔ good, Ø mean, ✗ poor.

time since protocol optimizers can not be (ex)changed at runtime. Thus, it does not support the adaptation of cross-layer coordination algorithms at runtime. However, this feature, amongst others, is a prerequisite of a flexible cross-layer architecture which we describe next.

2.3.3.2 Flexible Cross-Layer Architectures

Although static architectures support developers in building and running several specific cross-layer coordination algorithms, they neither provide the ability to add, remove or modify cross-layer coordination algorithms at runtime nor allow the automated (de)activation of coordination algorithms based on the presence of a certain context such as chaining network (e.g., wireless versus wired) or system conditions (e.g., battery connection versus power supply).

In this thesis, we present the flexible cross-layer architecture CRAWLER and our main distinctive key features from existing work is that CRAWLER (i) allows the developers to specify cross-layer coordination ideas at a very high level of abstraction, (ii) enables runtime adaptability of cross-layer coordination algorithms depending upon the underlying network conditions, (iii) provides rich application support by enabling applications to interact with CRAWLER and provide their own coordination algorithms at runtime, (iv) provides the necessary support for developers to handle problems involved when adding multiple cross-layer coordination algorithms, and (v) supports developers in testing, monitoring and analyzing cross-layer coordination algorithms. To the best of our knowledge, these key features are not supported by any of the existing cross-layer architectures. An overview of the presented architectures and their support with respect to all presented requirements is given in Table 2.1.

In the following we present CRAWLER and discuss these listed key features in detail.

3

A Generic and Flexible Cross-Layer Architecture

After introducing the problem space, presenting a descriptive background and discussing related work in the previous chapters, we now turn our focus towards the major contribution of this dissertation. We begin with answering our first research question, i.e., how to enable a generic and flexible cross-layer architecture that facilitates convenient system monitoring, cross-layer design and experimentation?

So far, we have introduced the cross-layer design paradigm and discussed problems when using this design paradigm. In this conjunction, we have presented static architectures that partly addressed these problems. In static architectures the process of realizing cross-layer coordination algorithms is deeply integrated into the system and thus they neither provide the ability to add, remove or modify cross-layer coordination algorithms at runtime nor allow the automatic (de)activation of cross-layer coordination algorithms based on the availability of a certain context such as chaining network conditions. In this chapter we address these issues, amongst others, and present Crawler, a cross-layer architecture for wireless networks that (even for non-domain experts) enables convenient and versatile adaptation of protocols, system components, and applications at runtime, if desired triggered by contextual changes.

The remainder of this chapter is structured as follows. Section 3.1 motivates the lack of a generic and flexible cross-layer architecture for convenient realization, monitoring and experimentation of cross-layer coordination algorithms. For this purpose, in Section 3.2 we analyze problems that come along with designing cross-layer coordination algorithms to highlight the requirements for an ideal architecture. Subsequently, based on the problem analysis, we present the design goals that shape our proposed cross-layer architecture in Section 3.3. We then present the details of the architecture using a goal driven description in Section 3.4, followed by a presentation of the implementation and architecture overhead in Section 3.5. Finally, Section 3.6 concludes this chapter.

3.1 Motivation

Many specific cross-layer coordination algorithms have been suggested in the past. Unfortunately, all of these are tailor-made implementations to realize a specific cross-layer coordination without the focus on integrating further coordination algorithms into the system. Thus, these tailor-made implementations lack focus on software engineering principles such as maintainability and extensibility which is the crucial prerequisite for real use and proliferation of software. As a result of this observation, few static cross-layer architectures have been proposed that focused more on software engineering principles.

Although static architectures facilitate easy manipulation of protocol-stack parameters and allow cross-layer developers to run several specific cross-layer coordination algorithms, from an application and system develops point of view still further challenges remain. First, the process of realizing a cross-layer coordination process is still tedious as the information required for the cross-layer coordination processes and the coordination process itself are deeply embedded into the operating system (OS) and can only be realized at compile time. For experimentation reasons adding, removing and manipulating cross-layer coordination algorithms at runtime will highly accelerate and simplify the realization of desired coordination algorithms. Second, the realization of a cross-layer coordination process requires being a system expert as modifications of protocols require OS and protocol behavior understandings. Third, due to the static nature of existing architectures once a coordination algorithm is added into the system, it is always running and adapts the system behavior even if it is not necessary. For example, an energy efficient but performance suboptimal cross-layer coordination is not needed if plugged into power. Therefore, the statically added cross-layer coordination could become superfluous and may even adversely effect other active applications and protocols. Finally, application developers who know best about their application requirements and constraints can not specify and provide their own set of cross-layer coordination processes into the system.

From these challenges we derive the need for a runtime flexible and generic cross-layer architecture that enables the adaptation of system behavior (i.e., protocols, hardware components like sensors, and applications) based on application requirements, system state and network conditions. However, designing an runtime flexible and generic cross-layer architecture requires to tackle all these major problems. In the next section, we discuss the identified problems in more detail and derive the three research questions that we address in this chapter.

3.2 Problem Analysis

From the motivation we derive three limiting key factors that hinder cross-layer development, real use and accordingly proliferation of a cross-layer architecture. In the following we discuss the details of these limiting key factors.

Lack of an easy-to-use and systematic development support for cross-layer coordination algorithms

The process of designing a cross-layer coordination is cumbersome. It requires to consider three steps: (1) the access to protocol state information (stored in variables), (2) aggregation or computation and (3) manipulation of further parameters based on the previous step. Each of the steps may lead to many hindrances. For example, in each OS the access to a certain variable could be implemented variously and located differently within the code. This requires OS expertise to understand the OS peculiarities and programming language expertise to realize the intended access to the parameter. Moreover, the variable could be modified and used at different functions of the protocol which also require understanding of protocol behavior.

Another problem is where to realize the aggregation or rather computation of the accessed parameters. Typically, protocols and system parameters are placed in the kernel and implementing computations in the kernel is not a trivial task as it requires expert knowledge. On the other hand, a move of the computation to the user space of the operating system could lead to unnecessary context switches which can lead to performance drops.

Finally, after accessing the parameter and deciding where to implement the computation, patching and compiling is a common next step which is very time consuming and cumbersome when finding and fixing programming faults is necessary. To conclude, all of these steps require expert knowledge and significant effort from the developer. But we believe that by introducing an adequate abstraction, developers (especially these who are not system experts) will be released from the burden involved with these three steps mentioned above.

However, when we assume these steps are implemented arbitrarily for many coordination algorithms, then this will likely lead to violation of software engineering principles such as maintainability and reuse of code. Accordingly, on top of the previous requirements, a systematic realization of these three steps are necessary to ensure long-term benefits and sustainability.

In a nutshell, we believe that an architecture needs to support a systematic development of cross-layer coordination algorithms while making it utilizable, even by non-system experts.

Lack of a monitoring and experimentation platform for cross-layer coordination algorithms

Till finalizing cross-layer coordination algorithms many experimentation runs and adjustments of the cross-layer coordination algorithms are necessary. We believe that a developer needs support in the experimentation process by providing the ability to monitor many parameters within the system and to fine-tune cross-layer coordination algorithms. Such a capability requires the selection of relevant parameters at runtime to identify interesting (mis)behavior and to compare the relative differences after adjustment. Moreover, from an application developer point of view this further requires to have easy-to-use interfaces for monitoring and modification purposes as they are not system experts and accordingly should not be impaired with additional effort except designing their applications. In particular, developers should be able

to add, remove, or modify coordination algorithms whenever they want and also should have the ability to monitor the desired variable in the whole system.

Lack of a runtime context adaptive cross-Layer architecture

Specific cross-layer coordination algorithms are build with a certain scenario in mind. For example, many cross-layer coordination algorithms have been suggested to improve TCP in wireless environments. But when conditions change, for example, an Ethernet connection is available for the device, the TCP cross-layer coordination built for wireless environments is superfluous. Accordingly, such a case necessities a system where coordination algorithms can be activated and deactivated when necessary. Similarly, application designers know best about their application and their needs in terms of parameters from the system in order to build adaptive software. Accordingly, application designers need to share their variables and their coordination algorithms with the system and vice versa for a joint coordination. But applications start and terminate unpredictably, i.e., caused by user demands. Therefore, there is a need to detect such changes and the ability to feed the required coordination algorithms at runtime into the system and make application variables accessible. In other words, we need a system where cross-layer coordination algorithms can automatically be loaded and unloaded based on specified conditions.

From these observations we derive following research questions that we tackle in this chapter.

Question Q1 - How to enable convenient & systematic cross-layer development?
We designed an architecture that allows to express cross-layer coordination processes at a high level of abstraction or configuration. The configuration is automatically parsed and subsequently mapped to module compositions which ensures software engineering principles and thus support developers in designing their cross-layer coordination algorithms.

Question Q2 - How to enable a monitoring and experimentation architecture?
The generic design of the modules and their interfaces allow flexible compositions at runtime. At the abstraction level this flexibility enables to specify which, when and how modules should be composed to achieve the desired set of cross-layer coordination algorithms. To further support this, for experimentation and monitoring purposes we provide a generic interface to add, remove and modify cross-layer coordination algorithms.

Question Q3 - How to achieve a runtime context adaptive system?
We provide a generic interface for applications to provide their desired set of cross-layer coordination algorithms into the system. For this, only a configuration needs to be provided which includes the conditions, i.e., availability of certain context, to load or unload coordination algorithms. The architecture takes over the responsibility to automatically check for these conditions and to load the predefined set of coordination algorithms whenever the conditions are satisfied.

In the following, we present a design overview of our cross-layer architecture and our design goals or rather contributions that tackle the aforementioned problems.

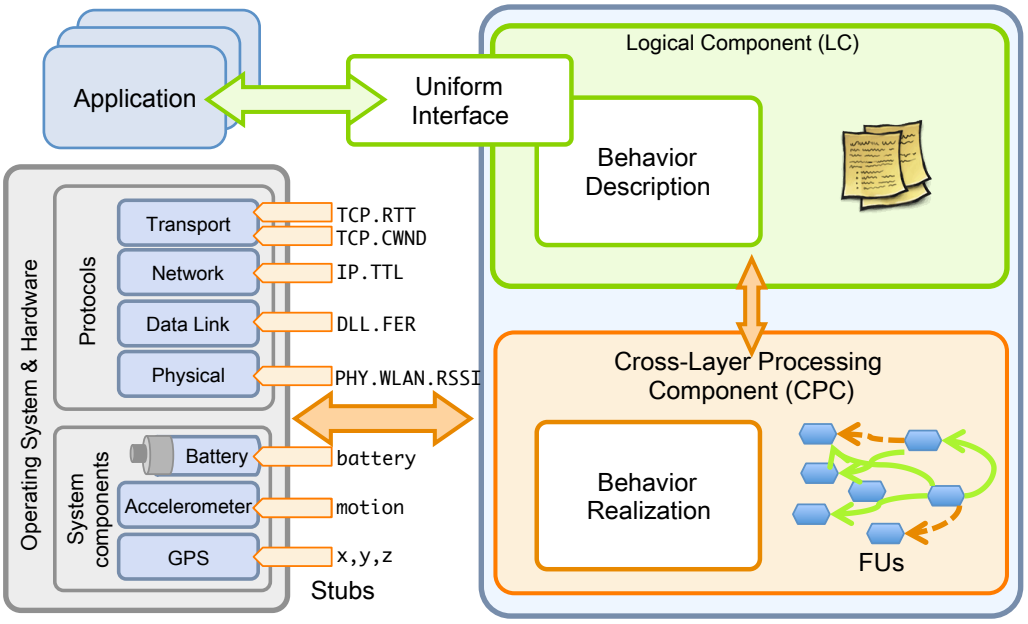


Figure 3.1 Concept view of CRAWLER. The logical component (LC) abstracts from the implementation of cross-layer coordination algorithms via an easily usable but powerful rule-based configuration language. The cross-layer processing component (CPC) realizes the coordination algorithms given by the LC which can be readjusted flexibly at runtime. The uniform interface allows applications to provide coordination algorithms and share variables with the system. Stubs provide access to protocol information and system components.

3.3 Design Overview

In this section we present a design overview of CRAWLER[1], a **cr**oss-layer **a**rchitecture for **w**irel**e**ss netwo**r**ks that even for non-domain experts enables convenient and versatile adaptation of protocols, system components, and applications at runtime. CRAWLER consists of two main components as shown in Figure 3.1:

The *logical component* (LC) allows cross-layer developers to express their monitoring and coordination needs in an abstract and declarative way. For this purpose, we have created a rule-based language customized to cross-layer design purposes. Using this language, developers can specify cross-layer signaling at a high level without needing to care about implementation details. Additionally, the LC offers a uniform interface that allows applications (i) to provide their own coordination algorithms on demand, and (ii) exchange information with the protocol stack, system components and other applications.

[1]The content of this and subsequent sections are partially based on the joint work with Muhammad Hamad Alizai, Florian Schmidt, Hanno Wirtz, and Klaus Wehrle published in "Harnessing Cross-Layer Design", Elsevir Ad-hoc Networks Journal, November 2013 [AAS+14]. The aforementioned journal paper in turn is based on the joint work with Florian Schmidt, Hammad Alizai, Tobias Drüner and Klaus Wehrle published in "CRAWLER: An Experimentation Platform for System Monitoring and Cross-Layer-Coordination", 13th International IEEE Symposium on a World of Wireless, Mobile, and Multimedia Networks, 2012 (WoWMoM'12) [ASA+12] and Tobias Drüner's bachelor thesis [Drü10]. Finally, subsequent sections are also partially based on the joint work with Jens Otten, Florian Schmidt and Klaus Wehrle published in "Towards a Flexible and Versatile Cross-Layer-Coordination Architecture", 29th International Conference on Computer Communications (INFOCOM 2010) [AOSW10], and Jens Otten's diploma thesis [Ott09].

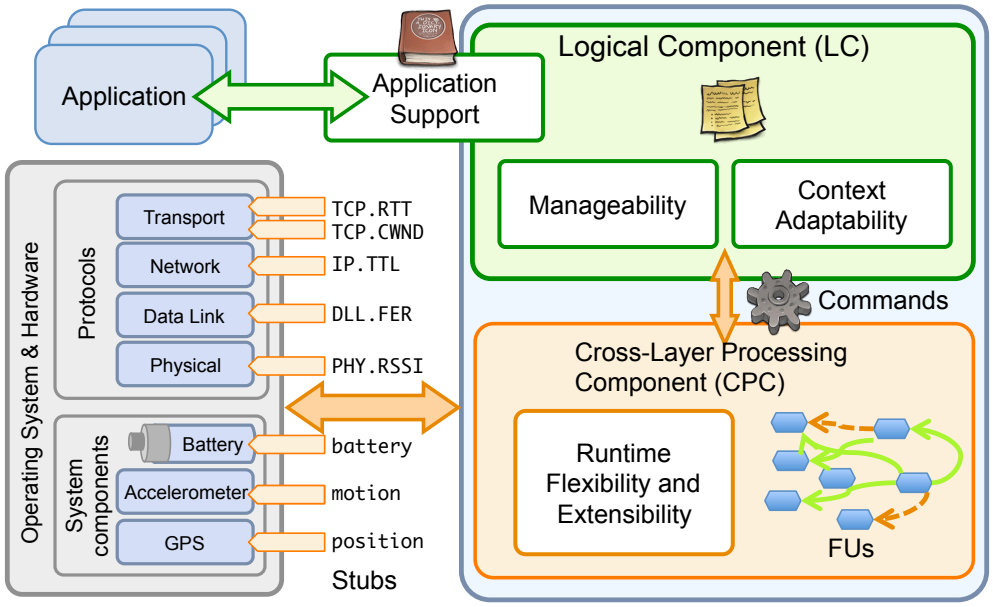


Figure 3.2 Goal-driven view of CRAWLER's components. The logical component (LC) contains the goals Application Support, Manageability and Context Adaptability. The cross-layer processing component (CPC) provides Runtime Flexibility and Extensibility.

The cross-layer coordination algorithms as specified in the LC are realized by the *cross-layer processing component* (CPC). Here, rules are mapped to compositions of self-written *functional units* (FUs). Finally, *stubs* provide read/write access to protocol information and sub-system states via a generic interface that abstracts from a specific implementation. Thus, additions and changes in coordination rules can be done at runtime using the LC. These changes are reported to the CPC, which adapts the FU compositions accordingly.

Before going into further details of the architecture, we present our design goals and briefly highlight the scope of our approach. Details about the architecture will be presented subsequently centered around our design goals.

3.3.1 Goals

Based on the observations stated in the motivation and the problem analysis above, we define four goals that shape our design as shown in Figure 3.2:

Goal 1 - Manageability: We believe that using cross-layer coordination algorithms should not impair the key software engineering properties, such as modularity, maintainability, and usability, of the layered protocol stack despite introducing dependencies across non-adjacent layers. Providing a systematic solution, for instance in form of a cross-layer architecture, should not impose additional requirements such as protocol dependencies when developing new protocols and system components. Moreover, we believe that cross-layer coordination algorithms should be easily maintainable and usable for application and system developers without requiring too much knowledge about system details and architectural requirements. Accordingly, developers of cross-layer coordination algorithms should primarily focus on designing the cross-layer interaction and

not be complicated and slowed down by system details. Nevertheless, we believe that since system developers are experts of their system, it is beneficial if they already provide access to the protocol and system variables to support cross-layer interactions.

Goal 2 - Application Support: Unlike existing approaches, the architecture should provide a unified interface for application developers to (i) specify and add their own monitoring and coordination needs into the system, and (ii) bundle these coordination algorithms with their applications, without needing to deal with OS level details. Moreover, it should simplify the process of accessing protocol and system information typically placed in the OS which only offers few limited interfaces. Ideally, application developers should not be impaired with manual inspection and adaptation of the very large OS code base.

Goal 3 - Runtime Flexibility and Extensibility: The architecture should offer flexibility that is essential for adjusting and experimenting with different sets of coordination algorithms, and further, the extensibility for involving all possible protocols and system components. In other words, for designing a coordination algorithm, the exchange of information between any number of layers and system components and the composition of any number of specific cross-layer coordination algorithms should be possible at runtime. To achieve this, the design of an architecture has to offer sufficient versatility to cope with the diversity and permanent evolution of protocols and application requirements.

Goal 4 - Context Adaptability: The architecture should offer the ability to (i) detect the underlying environmental changes, and (ii) respond to the changing application monitoring and coordination demands (e.g., when starting/terminating applications), by automatically loading the adequate set of coordination algorithms at runtime. For example, energy saving coordination algorithms may not be necessary if the device is plugged-in to a power supply. This necessitates detecting the right condition (i.e., plugged to power) and loading the right set of coordination algorithms (e.g., better performing but energy-consuming coordination algorithms).

In the following we briefly discuss the relationship of our research questions and goals.

3.3.2 Relationship of Research Questions and Goals

The four previously presented goals will provide answers to the three identified research questions. Figure 3.3 provides the relationship between our research questions and goals. While the three goals manageability (G1), runtime flexibility & extensibility (G3) and context adaptability (G4) answer the research questions on a one-to-one basis, the application support goal (G2) affects all research questions. The major reason is that the application support contribution is an interface that is used by the other contributions to realize their functionality.

In the following we discuss limitations or rather the design scope of the architecture that also shaped the design of our architecture. Subsequently, we present the details of the architecture following a goal-driven order as listed above.

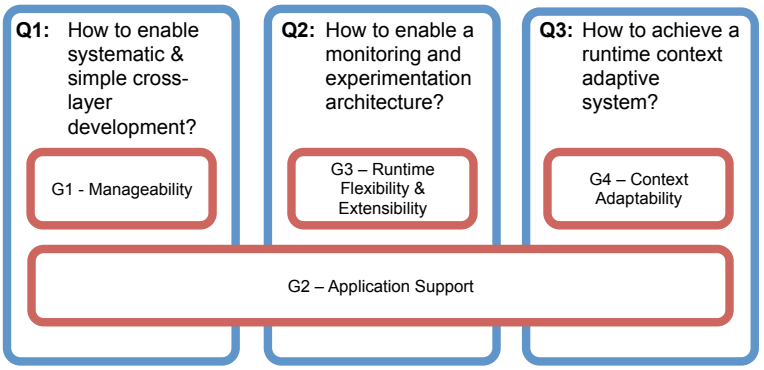


Figure 3.3 Overview presenting the relationship between research questions and contributions of this chapter.

3.3.3 Design Scope and Limitations

CRAWLER runs on end hosts and coordinates local information such as from the protocol stack and system components. CRAWLER itself does not establish information exchange among nodes in a network, such as in [WSNB06], because we believe that a monitoring and cross-layer experimentation architecture should not be responsible for establishing such information exchange mechanisms as it is rather the responsibility of a communication protocols. Nonetheless, a combination of such a protocol with CRAWLER could be used to share cross-layer information between nodes in a network. For example, in Section 4.4 we show a use case of CRAWLER where we shared a monitored parameter among neighboring nodes in an ad-hoc network in order to improve the detection of a jammer. Moreover, in Section 6.3 we extended CRAWLER's application support interface to remotely monitor and control (i.e., add, remove and modify) cross-layer coordination algorithms. Accordingly, we provided support to provide cross-layer information among nodes, however, from a design perspective we clearly separated the access to information and their exchange between nodes.

Another major design decision is to place the major part of the realization into kernel space of the OS. Since protocols and system information such as sensors lie within the kernel space of the OS, we wanted to keep their cross-layer coordination processing in the kernel space. The major reason is that information exchange between kernel and user space requires time which likely could lead to performance bottlenecks, for instance, if many coordination algorithms run in parallel. Nonetheless, if applications exchange information with the remaining part of the system, then context switches are not avoidable and we accordingly provide support for applications to exchange information (provide own information to and gather information from the kernel and thus share information with protocols and system components) which will be explained later in detail.

However, although we believe that the design of CRAWLER is very generic (i.e., in terms of supporting heterogeneous operating systems), we implemented and tested it specifically for Linux due to its open source nature. In particular, we have realized it for the Ubuntu and Vyatta distribution with kernel 2.6.32 for Ubuntu and 2.6.37 for Vyatta. The cross-layer realization component (CPC) of CRAWLER is implemented in C and is very generic. We have not used any third party library and believe that it could be easily adapted to any other OS (if they would be open

source). The implementation of the logical component is in C++ and considered similarly hard to adapt as any other application. The major problem here are the interfaces between the LC and CPC. Although our exchange format between these components can likely be reused, the interfaces among the OSs are very divers. We have used the generic netlink sockets as an interface for exchange information between both components. Other operating systems will require significant adaptations on both components and the accessors to information, i.e., our stubs, have to be reimplemented as protocols and drivers will significantly vary.

Furthermore, Crawler does not suggest cross-layer coordination algorithms. It is rather a framework for rapid design, realization, and monitoring in order to find out the right set of cross-layer coordination. Similarly this holds also for problems involved when running multiple cross-layer coordination algorithms. For instance, although Crawler also offers support for conflicting coordination algorithms (see Chapter 5.3), it nonetheless relies on cross-layer developers to provide the proper set of configurations and to ensure that the coordination algorithms are doing what they are aimed for or not conflicting with each other. All in all, Crawler allows to easily experiment with optimizations for different sets of use cases before being deployed on commercial systems.

In the following we present the details about the architecture following a goal-driven approach.

3.4 Architectural Details

We present a goal-driven description of Crawler by highlighting, with the help of simple examples, how our design achieves the four goals we laid out in Section 3.3.1. character

3.4.1 Manageability

The LC is the interface between developers and the CPC. Its major goal is to increase the usability and maintainability of cross-layer coordination processes for developers, allowing them to easily express their desired optimizations without paying too much attention to implementation details. For this purpose, the LC is divided into four subcomponents as shown in Figure 3.4. The *configuration* subcomponent allows a developer to express cross-layer coordination processes on an abstract level. It thus hides or rather abstracts for developers the implementation details of the cross-layer coordination processes within a particular operating system. The *interpreter* subcomponent is responsible for parsing and mapping this abstract description to so-called *commands*. These commands instruct the CPC on how to realize the given cross-layer description. In addition, these commands are stored in a *repository* subcomponent that maintains a view of the current realized cross-layer coordination processes in the CPC and stores all changes made to that state. The *application support* subcomponent allows applications to share their variables for cross-layer optimizations. Additionally, it allows applications to add their own monitoring and optimization needs. In the following we discuss the first three subcomponents which

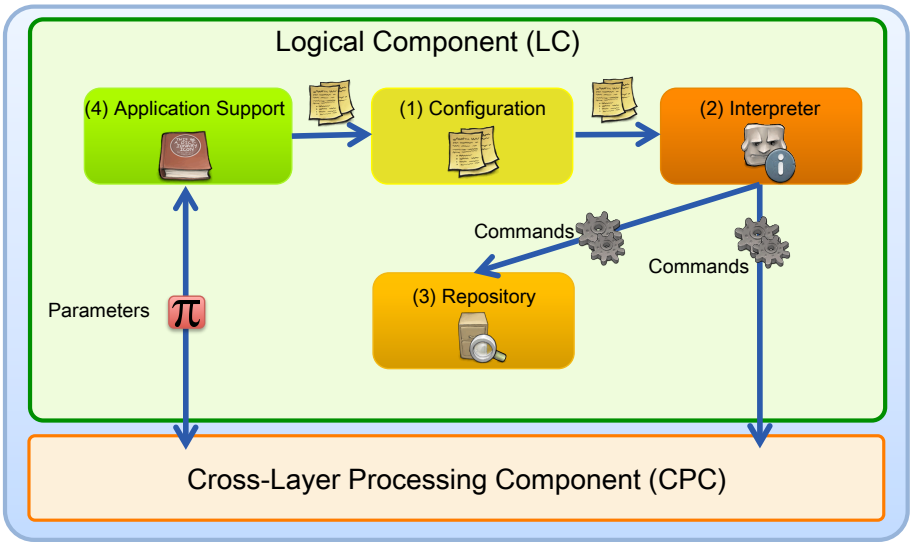


Figure 3.4 The LC comprises four subcomponents. (1) The configuration is an abstract description of a cross-layer optimization. (2) The interpreter parses the configuration. (3) The repository saves snapshots of configuration setups, allowing easy access to the current and past setups. (4) The application support component provides an interface to applications for communication with CRAWLER in order to provide own optimizations and access to parameters.

are intended to meet our design goal of *manageability*. We postpone discussion on the application support subcomponent to Section 3.4.2 to dedicatedly describe how this subcomponent realizes our second design goal of application support.

3.4.1.1 Configuration

The first step in CRAWLER's functionality is to allow the developers to specify their cross-layer optimizations. CRAWLER provides an easy to use but powerful rule-based language for specifying coordination processes in an abstract and declarative configuration. Each *rule* is a behavioral description of a part of a cross-layer interaction such as accessing protocol information and aggregation. Rules can be nested within other rules to form rule chains, i.e., to develop cross-layer optimizations. In Listing 3.1, we present an example configuration with rules that specify how to access and process protocol-stack information and when to notify it to the application. Each line in the configuration is a rule. Figure 3.5 shows a (slightly extended) graphical representation of this configuration. The figure is marked with numbers which correspond to the line numbers, i.e., rules, in the configuration.

The first rule `my_rssi` simply specifies which parameter, determined by a unique fully qualified name, should be accessed (see Section 3.4.3.2 for further details regarding the access mechanism). The second rule `my_history_of_rssi` collects the `History` (which saves a certain number, here `4`, of collected values) of RSSI (received signal strength indication) values, i.e., the last `4` RSSI values of the `wlan0` interface in this case. Similarly, the third rule `my_rssi_is_bad` determines if the average of these RSSI values is below a certain threshold, in this example `55`. Besides, the third rule also demonstrates the freedom of how rules can be nested with other rules. We have nested the `Less` and Avg rule without predefinition what we refer to as inline

```
1 my_rssi:get("phy.wlan0.rssi")
2 my_history_of_rssi:History(my_rssi, 4)
3 my_rssi_is_bad:Less(Avg(my_history_of_rssi), 55)
4 my_rssi_is_bad->my_appl_var1
5 my_rssi_is_bad->my_TCP_Freezer
6 my_appl_var1:set("application.app1.voip_var1", "bad")
7 my_TCP_Freezer:set("transport.tcp.cwnd", "0")
8 my_timer:Timer(200)
9 my_Timer->my_rssi_is_bad
```

Listing 3.1 A simple cross-layer signaling configuration in CRAWLER. This configuration file defines the setup illustrated in Figure 3.5.

```
1 my_Avg:Avg(my_history_of_rssi)
2 my_rssi_is_bad:Less(my_Avg, 55)
```

Listing 3.2 An alternative to the third rule in Listing 3.1 when not using inline identifiers. Although this increases code size, we suggest this notation for better reuse and runtime onfigurability.

identifiers. Inline identifiers can be used, for instance, to save code lines. An alternative could be a separate definition similarly as in the first and second rule which we show in Listing 3.2. Remark, we suggest the use of predefinition before nesting rules, as they provide also provide better reuse of configuration code and easier editing in case of runtime changes. For example, by using identifiers the separately defined rules such as `my_Avg` can explicitly be modified or exchanged by another rule such as with `my_Max` or `my_Min` as also indicated in Figure 3.5. We refer to Section 3.4.4 for more details.

So far, we have seen how computations and conditions can be specified using rules. However, sometimes it is desirable to react to events, such as a sudden drop in signal strength. This notification is denoted by an arrow such as in rules 4 and 5. The link quality condition of rule 3 is used to inform an application about the bad link quality (rule 6) and to reduce the sending congestion window of TCP connections to 0 (rule 7), i.e., to avoid triggering its congestion avoidance due to data corruption.

CRAWLER also allows developers to modify or add new rules during runtime. It recognizes these changes in the configuration and adapts the internal composition of cross-layer optimizations accordingly. For example, if we want to change the signal strength threshold, we only need to modify rule 3. We again defer further discussion on dynamic reconfiguration to Section 3.4.4.

Overall, the choice for a declarative and abstract language provides accessibility for developers who don't need to be cross-layer experts. Our language is customized to cross-layer needs in a way that all necessary functionalities for any kind of cross-layer interaction can be implemented. For the interested reader, we moved the details about the syntax of CRAWLER's declarative configuration language to Appendix A as it is not necessary to understand CRAWLER's abstraction concept.

However, as already mentioned, the configuration is only an abstract description of cross-layer interactions that need to be realized. In the following we will explain how the necessary information to realize the desired optimization is extracted from the configuration.

3.4.1.2 Interpreter

In the next step, such high level configurations of cross-layer interactions need to be transformed into the actual, resulting optimization. To this end, the interpreter subcomponent of the LC parses the configuration and maps rules to fine-grained instructions called *commands*. These commands hold instructions for the CPC on how to wire and parameterize different *functional units* FUs to compose a certain cross-layer coordination process. FUs are special stateful functions that keep their private variables between calls, and that have a uniform interface that allows to wire any FU with any other FU. Thus, the uniform interface enables the required flexibility to achieve the desired cross-layer optimization. Certainly, syntactically correct but useless compositions of FUs can be created. But as mentioned before CRAWLER does not protect from semantical misconfigurations. It is rather a support for developers to experiment and finalize their optimizations.

To give an example on how rules are mapped to commands that gives instructions on how to wire FUs, we pickup rule 2 in Listing 3.1. Here, rule 2 is parsed and the commands `createFU(History)`, `addParameter(my_rssi)` and `addParameter(4)` are gained which are then used to parameterize and wire the corresponding FU `History` (cf. 3 in Figure 3.5) with the `get-FU` (cf. 1 in Figure 3.5). The details about handling commands and the realization of cross-layer interactions are further explained later in Section 3.4.3.

However, the interpreter also employs filters [BHS07] to debug configurations. For example, filters can identify redundant rules, cycles in FU compositions or simply print commands for debugging purposes. For instance, in Section 5.4 we use filters to automatically detect redundant compositions in independently added optimizations. Another usage for filters is to print the current FU compositions in the system before or while optimizations are running. We have included various filters into CRAWLER to conduct different tasks while running.

To conclude, the interpreter maps configurations to commands that give instructions about how to create, parameterize and compose functional units (FUs). The composition of FUs lead to the specified cross-layer optimizations. Moreover, we used filters to scan CRAWLER for predefined conditions in order to conduct specific operations such as to create the adequate commands from configurations and to troubleshoot cross-layer optimizations at a very early stage of the development.

3.4.1.3 Repository

The repository keeps track of all the changes in a configuration. As the name suggests, it behaves similar to a revision control system: Each time the configuration changes, the commands (as created by the interpreter) are automatically committed as a new revision. As a result, several revisions of a configuration can be stored in a preprocessed state. The benefit of this is twofold: First, this assists CRAWLER in switching between different optimizations without needing to parse the rules again. In a running system, this allows more efficient switching between preprocessed sets of optimizations, e.g., if a certain context such as network connection type or certain power state is available. Second, while designing and testing new cross-layer optimizations, the repository allows the developers to roll back to a previous, well tested

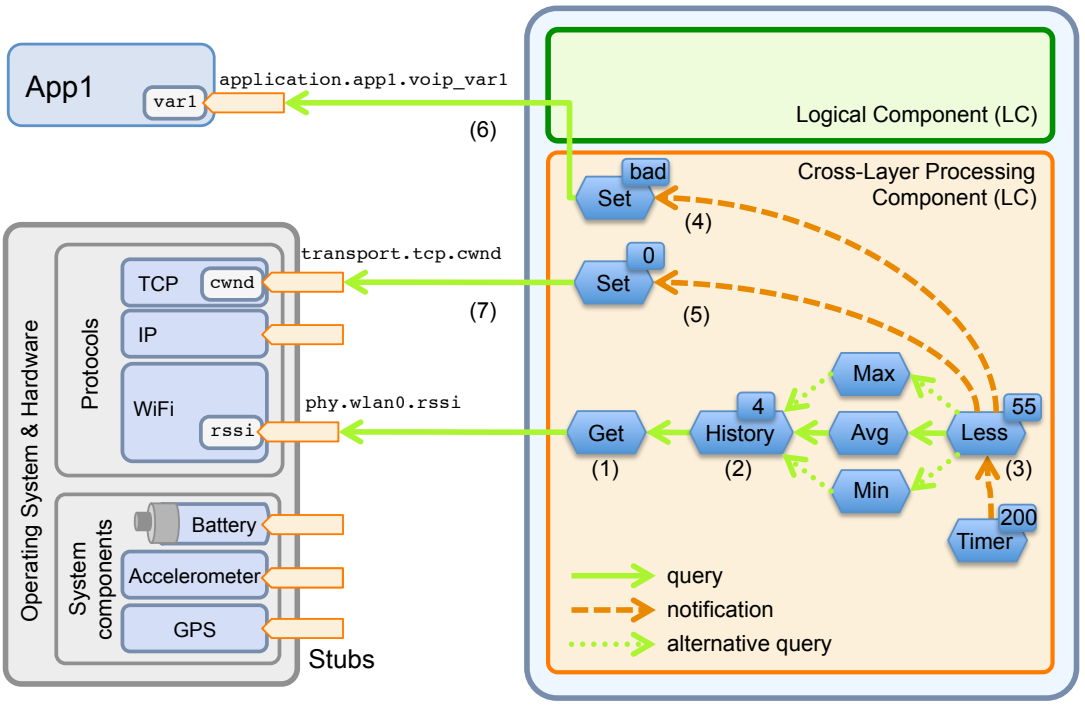


Figure 3.5 A simple cross-layer configuration in CRAWLER. We change the behavior of the TCP layer and an application based on signal strength.

and running optimization for debugging purposes. We used the repository very extensively in Section 5.4 where we continuously monitored the running optimization in the system to detect and remove redundant (parts of) cross-layer optimizations.

Summarizing, the declarative approach of specifying cross-layer interactions enhances the usability and maintainability of CRAWLER. None of the existing architectures simplify the specification of cross-layer optimizations to a degree where even developers who are not experts can describe cross-layer optimizations. Thus, because CRAWLER allows to specify cross-layer optimization at a high level of abstraction, it does not impose any system specific requirements on protocol and system developers. Hence, the collaboration of these three subcomponents of the LC fulfills our design goal of manageability.

3.4.2 Application Support

In the previous section, we discussed how a cross-layer developer specifies rules to describe cross-layer optimizations. However, to provide rich application support, we also need an interface between applications and CRAWLER. Such an interface allows developers to enable applications and the OS to work together to make informed joint adaptation decisions. For example, in a device, this could allow the OS to opt for a low-power mobile connection for background always-on services and switch over to a high-speed WiFi connection if the application requires a high-volume streaming connection. Similarly, an application could request a certain minimum and maximum required bandwidth and the OS could inform it about the bandwidth to be expected. The application can then choose a suitable transmission quality.

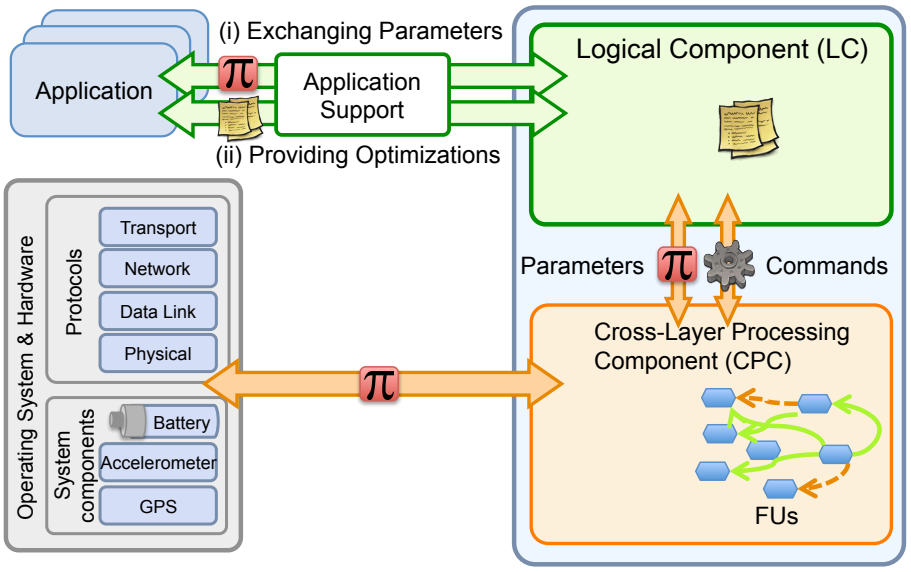


Figure 3.6 CRAWLER's application support provides rich interface for developers by providing two functionalities: (i) accessing system information and sharing variables and (ii) the ability to provide optimizations on an abstract level.

CRAWLER provides a rich interface for developers as shown in Figure 3.6. The interface enables the applications to specify their needs (i) by accessing system information such as from protocols and sensor and sharing their own information with the system and other applications, and (ii) by providing own cross-layer optimizations by providing a high-level abstract configuration, i.e., without needing to deal with implementation details of the OS or CRAWLER.

We first describe the former feature by demonstrating how simple it is for an application to share its variables and access system information. In a first step an application has to create a configuration for CRAWLER in order to realize the necessary cross-layer optimizations to access and share parameters. Let us consider the following example configuration given in Listing 3.3.

```
1 my_rssi_is_bad:Less(Avg(History(get("phy.wlan0.rssi"),10)),55)
2 my_app1_var1_ReadFromApp:get("application.app1.packetErrorRate")
3 my_plr_is_bad:Bigger(my_app_var1_readFromApp,4)
4 is_linkqualityBad:and(my_plr_is_bad,my_rssi_is_bad)
5 checkLink:if(is_linkqualityBad,1,0)
6 my_app1_var2_writeToApp:set("application.app1.linkQuality",checkLink)
```

Listing 3.3 This configuration shows an application accessing system information (`rssi`), sharing a variable with the system (`packetErrorRate`) and based on both values providing feedback to the same application by manipulating another variable (`linkQuality`).

This extract of a configuration presents an example of information exchange between an application and CRAWLER. In line 1 the RSSI of a WiFi device is accessed and the average is compared to a certain threshold, similar to the initial configuration example presented in Listing 3.1. In line 2 the application parameter `packetErrorRate` is accessed and provided to CRAWLER, for instance, to coordinate protocols and applications. For example, we have used the RSSI and the packet error rate (manipulated with netem) to switch the congestion control algorithm of TCP. We refer the interested reader to Section 4.2 where we present in detail the implementation

and evaluation of this specific use-case. Similarly, in this example, based on the RSSI and packet error rate (cf. rule 4 in Listing 3.3) we decide about the presence of a bad link quality. If rule 5 in Listing 3.3 is evaluated to "1" this indicates bad link conditions, "0" otherwise. Based on that evaluation we inform the application (cf. rule 6 in Listing 3.3) by setting its specific variable, here `application.app1.linkQuality`.

After providing such a configuration, applications need to register their corresponding variables in order to facilitate the signaling of states via a system-wide shared library. This only requires an application to include the library's header file `crawler.h`, provide callback functions to read or write to the application variables, and link against the library. The interaction between Crawler and applications is performed by the shared library itself. Listing 3.6 shows the few lines of code for an application in order to interact with Crawler.

```
1 #include "crawler.h"
2
3 int packetErrorRate, quality;
4
5 int getVariable(ValueD* result) {
6     if (result->name == PACKETERRORRATE)
7         result->setInt(packetErrorRate); //lib reads from variable
8     return 0;
9 }
10
11 int setVariable(ValueD* v) {
12     if ( v->name == QUALITY)
13         quality = v->getInt(); //lib writes to variable
14     return 0;
15 }
16
17 int main() {
18     /* (...) */
19     callback_ops co = {&setVariable,&getVariable,NULL};
20     initializeLibrary(co,APPNAME,strlen(APPNAME));
21
22     registerVariable(PACKETERRORRATE,strlen(PACKETERRORRATE),
23         CL_INT,VAR_READ, VALUEUPDATETIME);
24     registerVariable(QUALITY , strlen(QUALITY),
25         CL_INT,VAR_WRITE,VALUEUPDATETIME);
26     /* (...) */
27     addChains(configstr, strlen(configstr))
28     /* (...) */
29     return 0;
30 }
```

Listing 3.4 Registering an application and two functions via the shared library. Afterwards, using these two functions, Crawler is able (i) to read a specific variable, here `PACKETERRORRATE`, to allow the application to share its variable with the remaining part of the system, and (ii) to write to a specific variable, here `QUALITY`, in order to allow applications the gathering of information from the system.

First, the header file for the shared library is included in line 1 and the variables (i.e., `packetErrorRate` and `quality`) that should be accessed by the shared library are declared in line 3. Afterwards, for each variable that should be readable or writable by Crawler, a getter or setter respectively have to be implemented. Here, in lines 5-9 a getter is shown that is called by the shared library to read the packet error

rate from that corresponding application. Similarly, lines 11-15 show the setter that is called by the shared library to write to the application's variable `quality`. In the main method of the application, first, both callback functions are declared (cf. line 19). Subsequently, the application is registered with both callback functions (cf. line 20). Afterwards, all the specified variables that are accessible by CRAWLER are registered (cf. lines 22-25). In particular, the access type, i.e., readable or writeable, and their update intervals which further specifies the time between accesses are also defined.

At this time the architecture knows how to access the variables. But nonetheless the architecture requires instructions about which optimization it should execute. Therefore, as a next step the optimization or rather the configuration that we described above needs to be provided into the system. This can be simply done by the `addChains` method offered by the shared library (cf. line 27). From now on the cross-layer optimization (e.g., as specified in the configuration shown in Listing 3.3) is injected into the system and CRAWLER is able to access the variables shared by that particular application in specified intervals or triggered by the optimization, for example, based on satisfied conditions such as the bad link condition.

In the following, we present a helper application that demonstrates the simplicity and power of the shared library.

Helper Application for Monitoring

In order to demonstrate the simplicity and power of our interface, we implemented a show-case monitoring application that relieves the developer from any configuration efforts and makes the usage for monitoring purposes much more convenient. This show-case application does not only demonstrate how simple it is to use the shared library but also provides a ready-made system monitoring application to conveniently monitor a versatile set of variables within the OS (around 100 variables). For example, the monitoring application is called with the sending congestion window (`snd_cwnd`) of TCP as an argument as shown in the following listing.

```
1 $> ./monitorapp "transport.tcp.snd_cwnd"
```

Listing 3.5 The monitoring app conveniently allows to monitor divers parameters within the OS by a simple call in the console.

This simple call starts the application and constantly monitors and logs the desired variable in a file. By simple exchanging the unique variable name by another name from the set of variables that we already provide, a different variable in the system can be monitored. Later in Section 6.3 we show extension to our shared library and present three further helper applications.

So far, we have seen how applications can share and feed their specific optimizations into the system. However, the ability to add and terminate these optimizations during runtime is essential. This is because (i) application specific optimizations are only known to application programmers and may not be known at system start time, and (ii) these optimizations are only needed to be employed when the application is running. We will describe our support for application start / termination or rather context adaptability in Section 3.4.4. But before understanding how CRAWLER

adapts optimizations to certain context, we present how CRAWLER facilitates the modification of cross-layer optimizations at runtime which is the prerequisite of context adaptability support.

3.4.3 Runtime Flexibility & Extensibility

The flexibility of CRAWLER is associated with how a cross-layer optimization is composed and modified. CRAWLER provides a flexible wiring mechanism between FUs, the basic building blocks of an optimization, to enable developers to experiment with different compositions of an optimization. Similarly, extensibility deals with the underlying mechanism employed to access protocol-stack and system-component information. CRAWLER provides *stubs* as an extensible interface between cross-layer optimizations and the OS.

However, in the previous sections we have seen how rules are mapped to commands which in turn are instructions to CPC to wire FUs. In the following we present how we realized the wiring of FUs within the CPC.

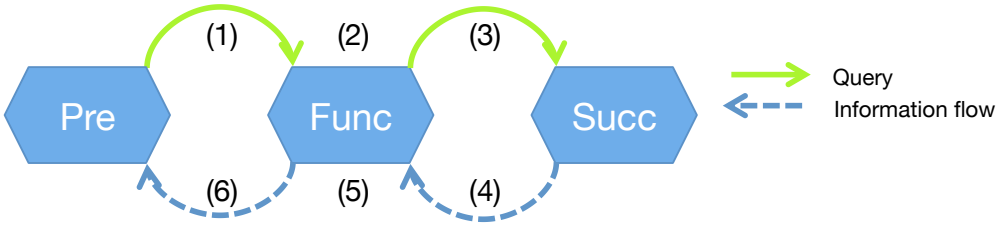
3.4.3.1 FU Wiring

FUs possess two properties which form the basis for dynamic reconfigurability and adaptability of cross-layer optimizations.

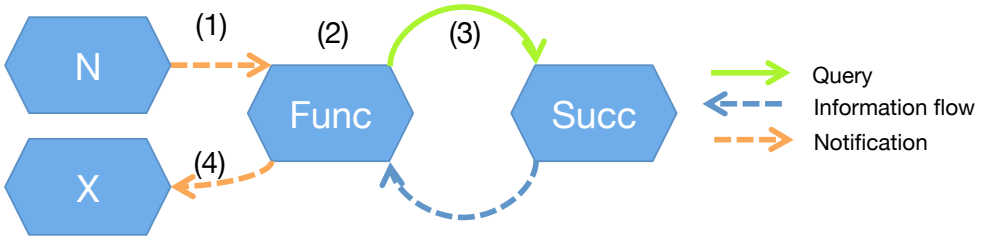
First, FUs are stateful functions that maintain a record of the data and provide results based on that record each time they are called. In contrast to stateless functions, whose output only depend on the input and the global state of the system, each FU keeps its private state (variables) as long as CRAWLER runs, much like an object in an object-oriented language. The output of an FU therefore depends on input, global system state, and private state of the FU. For example, every instance of `History` keeps its collected values between calls. As long as a configuration does not delete FUs but only changes their wiring, they will keep their current state and collected information.

Second, FUs share a unified interface so that they can be flexibly wired with each other. For example, by changing rule 3 in Listing 3.1, we can exchange the `Avg` FU in Figure 3.5 with `Min` or `Max` at runtime due to the uniform interface, and still use the collected data from `History`. This is because a change in the wiring does not re-instantiate all FUs. This opens up new possibilities especially for developers to utilize already established cross-layer optimizations. For example, an application can exploit already collected information such as localization information to provide future predictions based on past and recent data. However, the uniform interface of FUs also facilitates easy extension as newly designed FUs can easily be wired with the existing ones. CRAWLER supports two mechanisms to wire FUs, queries and events. Both types together cover the full range of information retrieval and aggregation to design any kind of cross-layer signaling.

Query-based Signaling: The query interface allows to explicitly request information. If the query interface of an FU is called, it returns the result to the inquiring FU. The query result of an FU may depend on the result of further FUs, leading



(a) Generic example of a **query-based signaling**: (1) Func-FU receives a query from predecessor Pre-FU, (2) Func checks its current state, (3) this might lead to cascading queries, here, Func queries successor Succ-FU, (4) Succ answers to Func, (5) based on Succs return value, Func computes its own return value, and (6) Func answers it result to predecessor Pre-FU.



(b) Generic example of a **event-based signaling**: (1) event from notifier (2) check current state (3) query (multiple) Succ FUs (4) notify interested FUs

Figure 3.7 FU wiring

to cascading queries. In case of providing the most up-to-date values, this is the intended behavior. In Section 3.5.2 we show benchmark test for the architecture that demonstrate that performance influences are practically not noticeable. However, to reduce the computational overhead, we suggest that each FU can cache its previously returned value and set a validity time for it. For example, on a query the `History`-FU returns immediately its collected and stored values instead of recollecting them. Thus, in case of a new incoming query, the FU can then decide to return the cached value or recompute a new one. Incorporating caching has to be implemented by FU developers as they know best about what to cache and how long these values are valid, i.e., the time to provide useful and reasonable results. However, Figure 3.7(a) describes the concept of our query-based signaling between three FUs. (1)The FU `Pre` queries the FU `Func`. (2) `Func` checks the validity of its return value. Here we assume that the validity time for its return value has expired, therefore (3) `Func` queries `Succ`. (4) Based on the return value of `Succ`, (5) `Func` computes its own new return value and updates its validity time correspondingly. Finally, (6) `Func` sends the result to the predecessor FU `Pre`. An example composition of FUs utilizing the query interface is also shown in Figure 3.5. `Less` queries `Avg` for an average of RSSI values provided by `History`.

Event-based Signaling: The query-based interface for compositions between FUs results in a polling architecture. To avoid unnecessary polling and to allow to react to sudden events, Crawler also supports an event-driven signaling that notifies interested FUs about the occurrence of an event, for example a significant change in a certain value measured by another FU. The notified FU can then act based on that triggering. In Listing 3.1, triggers that send notifications are defined in rules 4,5 and 9. Figure 3.7(b) depicts our concept of event-based signaling. (1) The `N-FU` notifies the FU `Func`. This can be, for instance, due to an elapsed timer or a measured change in a monitored value. (2) `Func` can now decide to act on that

notification, e.g., perform a specific calculation and / or (3) query further multiple FUs, here the `Succ`-FU is queried. Here again, we suggest that FU developers should decide how to react on a notification, this could be, for example, the same behavior similar to an incoming query or could completely differ. In CRAWLER we perform for all FU (so far) a query on an incoming notification. However, based on the configuration further FUs can also be notified, here (4) `X`-FU is notified. As an example in Figure 3.5, `Timer` acts as a trigger to periodically notify `History`, which then takes RSSI samples to save it and to provide it to other querying FUs. Furthermore, `Less` triggers two stubs (cf. Section 3.4.3.2) to set values in the TCP layer and the application.

Finally, to enhance the extensibility of the architecture, CRAWLER also maintains a *toolbox* that stores FUs. It helps in reusing generic FUs, such as `Timer` and `History`, or compose more complex FUs, such as a handoff estimation, by combining several small FUs.

Flexibility of CRAWLER is achieved with both signaling schemes which allow (i) the composition of FUs, the building blocks of optimizations, to realize the desired cross-layer optimization and (ii) their adaptability at runtime. In contrast, extensibility requires mechanisms to adapt to evolving nature of the OS, i.e, new or enhanced protocols and system-components. Therefore, this mechanisms should avoid dependencies between the cross-layer architecture and the OS. CRAWLER provides *stubs* as an extensible interface between cross-layer optimizations and the OS which we describe next.

3.4.3.2 Stubs – Accessing Signaling Information

Stubs provide read and write access to protocol and system information. They act as a glue element between the cross-layer optimizations and the OS. Stubs offer a common interface and a very fine-grained access to system information: Protocol and system variables have their own `get` and `set` stubs. Thus, to access the desired protocol or system variable, stubs need fully qualified, i.e., unique and hierarchical, names. For example, a stub's name begins with the name of the corresponding layer followed by the protocol name and the variable name, e.g. `transport.tcp.congestion_window`. As we differentiate between stub for reading and writing, the interpreter resolves the corresponding stub based on the prepended keyword get and set. Hence, reading that specific variable can be achieved by specifying in the configuration `get(transport.tcp.congestion_window)` and for writing `set(transport.tcp.congestion_window)` respectively (compare line 1 and 7 in Listing 3.1). In cases where writing values is not possible, e.g., sensors that provide read-only variables, stubs with only `get` functionality can be used such as the phy.wlan0.rssi" in Listing 3.1.

In CRAWLER's runtime the CPC automatically associates set and get FUs with each stub included in the architecture, as shown in Figure 3.5. However, protocol information often changes non-periodically and unpredictably as network conditions change. Because a stub is accessed by CRAWLER via FUs, these FUs can use the event-based signaling to notify other interested FUs about any change in protocol information. This increases the responsiveness of rules to changing conditions.

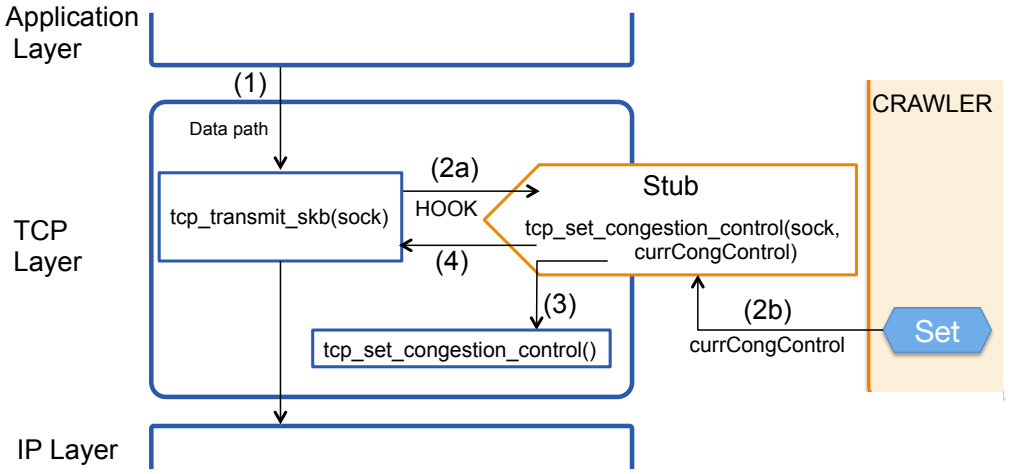


Figure 3.8 Stub for changing TCP's congestion control algorithm works as follows: (1) While processing a packet in TCP, the function `tcp_transmit_skb` is called. (2a) Here we hook into the processing and redirect the processing to the stub. (2b) In parallel (asynchronous), `CRAWLER` sets the adequate congestion control algorithm via a `Set`-FU. (3) Based on the given value, the congestion control algorithm is selected. (4) Finally, processing continuous.

Figure 3.8 shows an example of a stub that changes TCP's congestion control algorithm. The basic four steps to change TCP's congestion control algorithm are as follows: (1) After receiving and processing a packet from the upper layer, the function `tcp_transmit_skb` is called in TCP right before delivering the packet to IP. (2a) Here, we inject a hook that redirects the processing to the stub. (2b) The stub receives the current congestion control algorithm `currCongControl` from CRAWLER via a `Set`-FU in parallel (asynchronous). (3) If a change in the congestion control algorithm is requested, TCP's `tcp_set_congestion_control` algorithm is called for a certain socket. (4) Afterwards, the packet processing continues as normal. This stub is later used in the evaluation (see Section 4.2) to demonstrate a use case of CRAWLER.

Overall, stubs allow CRAWLER to monitor and coordinate a diverse set of protocols, system components and applications. Moreover, with a unified wiring interface between FUs, their different types of interconnection, and the ability to reuse and wire further FUs, a very high degree of extensibility and flexibility at runtime are achieved in CRAWLER.

3.4.4 Context Adaptability

Context adaptability is one of the key features of CRAWLER that allows to (de)activate cross-layer optimizations when necessary. For example, application support is not possible with a static set of rules that cannot adapt to application demands. Specifically, application specific rules might not be known at system start time; they have to be loaded when the application starts and removed when it terminates.

Examples presented so far have been rather static, i.e., the rules specified in the configuration in Listing 3.1 were available in a central configuration that is loaded automatically on CRAWLER's start and put into effect. But to load cross-layer

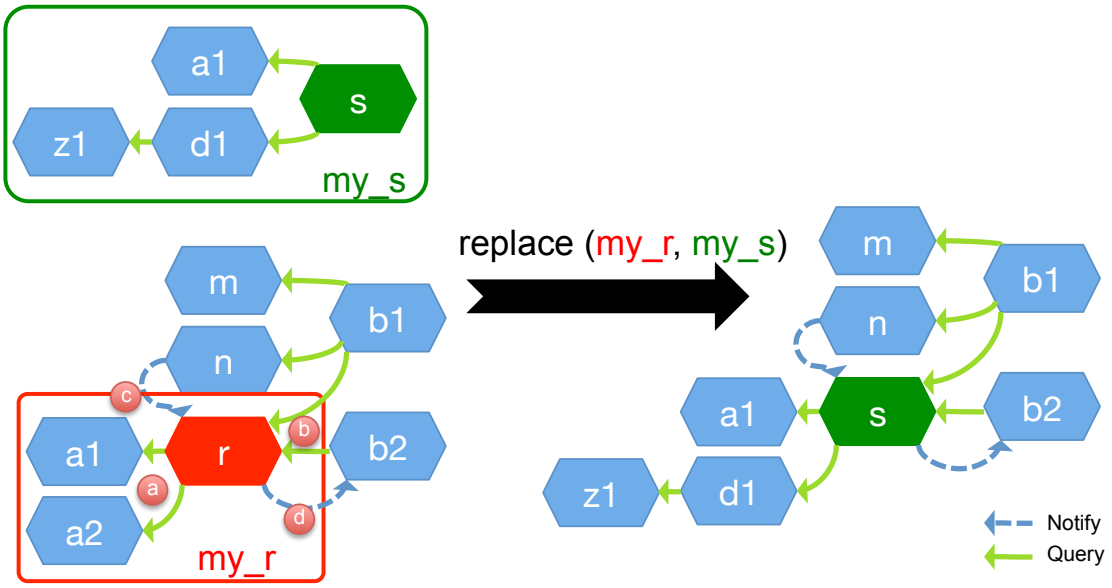


Figure 3.9 An example of the `replace` command where a running rule chain `my_r` is replaced with a rule chain `my_s`. The replacement of a rule is difficult, as it requires to rewire some of the connections of its representing FU, here `r-FU`. Moreover, connections into `r-FU` are also resolved and rewired to the FUs of the new rule `my_s`.

optimization when a certain context is available, we first need a mechanism for more control of cross-layer optimizations, i.e., to dynamically add, modify, and remove rules at runtime. To realize this CRAWLER provides the following three keywords that can be used in the configuration:

load(rule_name): The rule `rule_name` is loaded at runtime. This instruction is needed to add CRAWLER rules into the system, for instance, in a running system rules for monitoring a certain variable or several rules realizing a cross-layer optimizations. Dependencies between rules are automatically satisfied, for instance, if `rule_name` references another rules which are not loaded in the CPC they will be also loaded. For example, `load(my_rssi_is_bad)` in the configuration defined in Listing 3.1, will also automatically load `my_history_of_rssi`. The new rules (including rules having dependencies) are processed and composed into FU compositions as discussed in Section 3.4.1.2. Remark, the new FU compositions (described by our rules) can also be wired with already running FUs which we emphasized in Section 3.4.3.1.

unload(rule_name): The rule `rule_name` is unloaded at runtime. This instruction allows to remove rules from the system, e.g., if a developer wants to analyze and compare the system behavior without a specific optimization. The internal handling of unloading a rule is more complex than loading it since a naïve unloading can result in unreferenced FUs. To address this problem, CRAWLER associates a reference counter with each FU. As an example, `unload(my_rssi_is_bad)` will also unload the rule `my_history_of_rssi` unless it is used by another rule that is not listed in Listing 3.1. Sometimes there is a demand for keeping some rules. For example, if we consider again the configuration, the collected values by the `History` FU can be used later in time, e.g, an application may need the history of old localization information and corresponding RSSI values to predict future localization and / or connection quality values. Hence, if a developer is willing to prevent the automatic removal of an FU, this could be done by keeping a reference to the FU.

```
1 [manual]
2 rssiavg:avg(history(get("wlan0.qual.rssi"),10))
3 less1:less(rssiavg,60)
4 packetLossRate:get("app.switchCwnd.packetLossRate")
5 less2:less(4,packetLossRate)
6 changeCwnd:and(less1,less2)
7 cwndAlg:if(changeCwnd,"westwood","vegas")
8 initPort:set("tcp.activate.outgoingPacketsPort",5001)
9 setCwndAlg:set("tcp.cong_control_5001", cwndAlg)"
10
11 [contextEnter]
12 loadOpt:get("app.switchCwnd.loadOpt")
13 loadOpt->load(setCwndAlg)
14
15 [contextExit]
16 removeOpt:get("app.switchCwnd.removeOpt")
17 removeOpt->unload(ALL)
```

Listing 3.6 Configuration of an application-specified optimization: TCP's congestion control algorithm is changed based on packet loss rate (PLR) and RSSI values. If the PLR is high and the RSSI is low, TCP's congestion control algorithm is set from TCP CUBIC to TCP Westwood. If either of the conditions is not satisfied, the congestion control algorithm is set back to TCP CUBIC.

replace(rule_old, rule_new): The rule `rule_old` is replaced with `rule_new` at runtime. Internally the replacement of rules is a difficult task as some connections within the exchanged FU compositions have to be rewired. Figure 3.9 depicts an example where a rule chain `my_r` is replaced by the new rule `my_s`. As every rule, even if it is referenced multiple times in the configuration, only has a single correspondent FU representation, all connections and occurrences of the old rule needs to be resolved and replaced by the new rule. Thus, all the connections of `my_r` are re-wired to `my_s`. In particular following four types of connections indicated with (a)–(d) in Figure 3.9 have to be resolved: (a) outgoing query connections, (b) incoming query connections, (c) incoming notify connections, and (d) outgoing notify connections.

These three keywords trigger the functionality to add, modify, and remove rules at runtime. But before using these keywords to automatically load or unload the adequate set of cross-layer optimizations, it is necessary to provide the ability to describe and detect the underlying environmental changes.

Consequently, Crawler also provides mechanisms to automatically execute the rules associated with these three keywords based on context changes such as environmental conditions. For example, Listing 3.6 shows a configuration do demonstrate how application specified rules are automatically loaded or unloaded at runtime based on different conditions. Again using keywords, the configuration is divided into three sections. The `[manual]` section contains rules that are parsed by the Interpreter but are not directly applied in the CPC. `[contextEnter]` specifies which rules from the `[manual]` section should be loaded when a certain condition (also specified in the form of a rule) is met. Therefore, lines 12 and 13 specify that the rule `setCwndAlg` will be loaded when the application sets its variable `loadOpt` to true. Note that this configuration will be later used in the evaluation section to demonstrate the change of the congestion control algorithm of TCP. However, the section `[contextExit]` is used in the opposite way compared to `[contextEnter]`,

namely to unload rules when a certain condition is met. For example, in line 16 and 17 based on the application's variable `removeOpt` all rules are unloaded.

Summarizing, by supplying keywords to load, unload, and replace rules, CRAWLER achieves reconfigurability at runtime. Moreover, by using further keywords CRAWLER also allows to describe the conditions when a certain context is available and if the conditions are satisfied, CRAWLER automatically (un)loads a specified set of cross-layer rules.

So far, we have presented our four goals of CRAWLER that answer our research question of how to enable convenient realization, monitoring and experimentation of cross-layer optimizations. In the following we discuss implementation details of CRAWLER and show benchmarks to highlight its efficient realization.

3.5 Implementation and Architectural Overhead

In this section we discuss the implementation details of the architecture and evaluate the architectural overhead when running CRAWLER.

3.5.1 Implementation

We implemented CRAWLER[2] for Linux (kernel 2.6.32). The LC and all its subcomponents are implemented in C++. It runs as a daemon in user space. The CPC resides in kernel space and is implemented in C. Remember that the CPC realizes the cross-layer optimizations. Since system components and protocols which we wanted to coordinate reside in the kernel, we opted to move the CPC into kernel which reduces the number of expensive context switches between kernel and user space during runtime. The communication between LC and CPC takes place via flexible interfaces provided by generic netlink sockets [NAGL10].

To utilize CRAWLER as an application developer, applications can link against a shared library that contains all the functionality to interact with the LC. The interface itself uses unix domain sockets. Via these sockets, we transmit information about changing variable values. They are also used to send application-specific optimization rulesets, such as the one shown in Listing 3.6 and discussed in Section 3.4.4. Furthermore, multiple running applications can also utilize this interface to allow an exchange of variables among these applications and CRAWLER. Although we used unix domain sockets to realize the inter process communication, in our design we considered this scheme to be flexible exchangeable.

The wiring between FUs is implemented using a special data type that can contain characters, integers, boolean values, arrays, and a struct-like compounds of these types. Each FU can act accordingly to the received type, i.e., slightly different behavior depending on whether it receives a single value or an array of values. This has to be considered by the FU developer. So far, we have implemented about 20

[2]This article focuses on the main features of the CRAWLER architecture that support our design goals. The source code and documentation of the whole architecture can be accessed via `http://www.comsys.rwth-aachen.de/research/projects/crawler/`

FUs and 100 stubs, with the numbers growing with every new sample scenario. In Appendix B we give an overview of available stubs and FUs.

To evaluate the quality of our implementation, in the following we present benchmarks that verify different aspects of the architecture.

3.5.2 Architecture Overhead

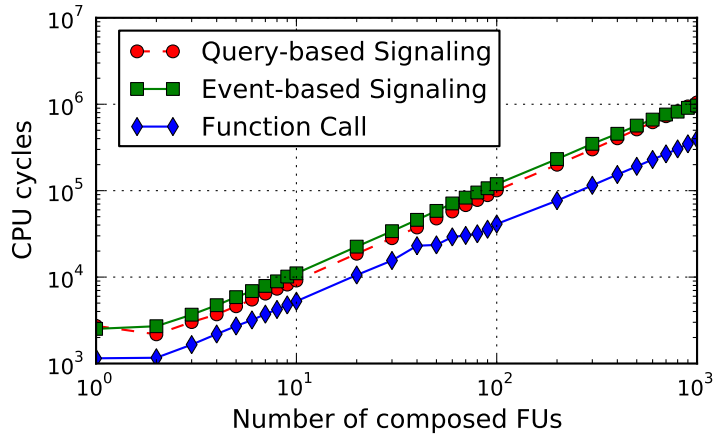
We now measure the runtime overhead of our architecture. During Crawler's runtime, the CPC, provides two main functionalities: (i) registering and wiring FUs and stubs, (ii) signaling between FUs and stubs to access protocol and component information. The registration of FUs and stubs is not time-critical since this only happens when a new optimization is loaded into the system. During the registration, each newly created FU and stub is checked to prevent duplicates. This has a runtime of $O(n) + O(m)$ where n and m are the number of already existing FUs and stubs, respectively.

Query-based and event-based signaling (cf. Section 3.4.3.1) play a vital role in determining the processing overhead of Crawler. To measure this, we use a simple benchmark of several wired `Forwarder` FUs. These do not contain any complex logic: they simply relay the query to the next FU. The idea here is to keep the complexity of the FUs as low as possible to measure the signaling overhead between FUs.

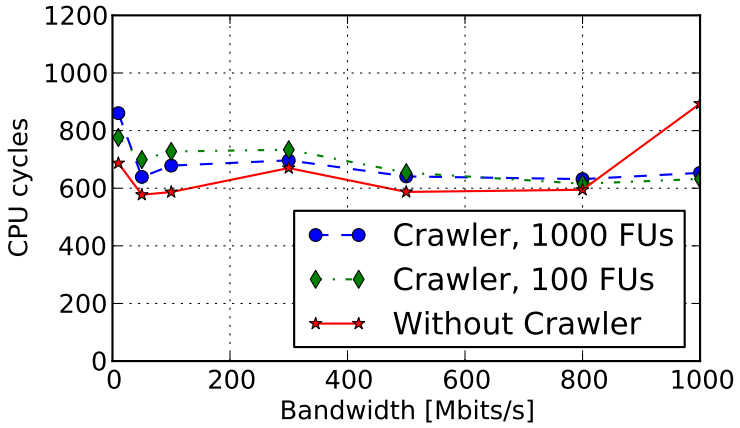
Figure 3.10(a) shows the results for both of the signaling mechanisms of Crawler when compared with a standard Linux function call (note the logarithmic scale on both axes). We created chains of `Forwarder` FUs of different lengths, from one to one thousand chained FUs. Afterwards, we measured the CPU cycles required to traverse all `Forwarder` FUs, repeating each benchmark 100 times. The results show that query-based and event-based signaling mechanisms introduce an overhead of a factor 2.1 and 2.8 when compared with native Linux function call, respectively. This can be mainly be attributed to locking mechanisms that protect the FU's states from concurrent access. We can clearly see that our two signaling schemes induce little overhead which increases linearly with the length of the chains. Thus, the overhead that our FUs and their composition create is negligable.

Furthermore, this processing overhead does also not increase the processing time of network packets. To show this, we connect two notebooks via a Gigabit Ethernet. The sender notebook runs the Crawler implementation with an optimization that changes each outgoing packet by manipulating the TTL field of the IP header. The second notebook ran an Iperf server [TQD+04] to create an endpoint for the first notebook that was running the Iperf client. We configured optimizations consisting of the following two rules: Rule 1 creates a chain of `Forwarder` FUs of different lengths. At the end of this FU chain, we added a simple FU that incremented an integer value. Rule 2 registers a netfilter hook in the IP output path that sets the TTL to that value. We successively increased the amount of UDP traffic via Iperf.

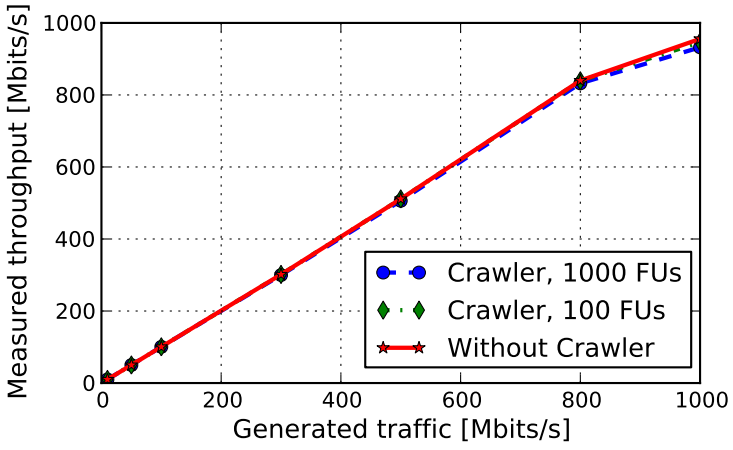
The results for our benchmark are shown in Figure 3.10(b) which clearly highlight that the length of rule chains does not contribute noticeably to the per-packet processing time. For comparison, we added a measurement of Iperf traffic generation



(a) Signaling overhead for query-based signaling and event-based signaling



(b) Packet processing duration within the protocol stack



(c) Measured throughput for different FU composition sizes

Figure 3.10 Performance measurements of CRAWLER. (a) The signaling overhead has a linear increase of CPU cycles with increasing amount of wired FUs. (b) As CRAWLER's rules run asynchronously, packet processing time is independent of the amount of wired FUs. (c) Likewise, throughput is not influenced.

without Crawler running at all which we refer to as "Without Crawler". The curves of our results for all three cases show a similar behavior with Crawler only utilizing marginal more CPU cycles, independent of the amount of composed FUs. The only exception occurs for generated bandwidth above 800 $Mbits/s$ where interestingly the number of CPU cycles is marginally higher for the case without running Crawler. We have conducted the benchmark several times which always lead to the same behavior. We assume that the case are differently affected by the memory management of the OS providing an marginal advantage for the case running Crawler. However, our results reflect the fact that Crawler works asynchronous to packet processing. That is, stubs accessing information in the system are not blocking the packet processing. In particular, the value of a variable that is being accessed by the stub is copied and Crawler works only on the copied version of the variable. In case of a setter stub it writes the values back to the original variable. Therefore, the packet processing is not affected. For details, we refer to Section 3.4.3.2.

This fact is also underlined in Figure 3.10(c) which depicts the throughput measurements for the same experiments. We see that even with a huge amount of wired FUs, the throughput is not significantly affected. Both of these results also underline, asynchronous packet processing is the right mechanism towards an efficient cross-layer architecture.

Overall, these results conclude that, while Crawler introduces processing overhead, this overhead does not deteriorate network performance in terms of throughput and packet processing time.

3.6 Conclusion

In this chapter we have presented Crawler, a cross-layer architecture for wireless networks that enables flexible and versatile adaptation of protocols, system components, and applications. We described the architecture following a goal-driven description which lead to the following four contributions of this thesis.

First, we presented Crawler's manageability contribution which enables convenient and systematic cross-layer development. In particular, we have introduced a declarative rule-based language to specify cross-layer optimizations. Specifically, the abstraction level of the declarative language does not impose any system specific requirements on protocol and system developers, consequently, even non-domain experts can specify cross-layer optimizations. After providing a configuration, the cross-layer coordination processes are mapped to module compositions which enables software engineering principles such as maintainability and extensibility.

Second, Crawler provides a rich interface for application developers which we refer to as application support. The interface allows application developers to benefit from two functionalities. On the one hand, it allows applications to access protocol and system component information and to share their own information with protocols, system components and other applications. On the other hand, it enables applications to provide their own cross-layer optimizations by feeding a high-level abstract configuration into the system.

Third, Crawler's degree of flexibility and extensibility contribution is the necessary basis for convenient and rapid experimentation with different set of protocols, system components and applications. In terms of flexibility, Crawler allows developers to add, remove and modify cross-layer optimizations at runtime which is achieved by the ability to map configurations to FU compositions. Here, the interface between FUs is designed in a generic way to enable the reuse and (re)composition of FUs at runtime. In terms of extensibility, Crawler allows to involve all possible protocols and system components and their interactions by using our stub concept which allows the actual read and write access to protocol and system information.

Fourth, by supplying keywords to load, unload, and replace optimizations when a certain context is available, Crawler's context adaptability contribution allows to automatically react to certain network conditions in a device's environment by adapting a specified set of optimization, i.e., executing a predefined set of rules depending upon satisfied conditions (determined by keywords) that are defined by the developers.

However, after introducing our four contributions, we applied different benchmarks to Crawler where we showed that Crawler does not deteriorate the network performance parameters such as throughput and packet processing time. In the following chapter, we further evaluate Crawler by demonstrating the utility and correctness of Crawler's implementation with help of use cases from diverse fields of networking.

4

Practical Use Cases and Evaluation with CRAWLER

This section focuses on how Crawler can be utilized for monitoring, experimentation, and cross-layer adaptation purposes in diverse networking areas. For this we show five different use cases from different research fields. We first focus on the classical and well-known cross-layer problem, namely TCP's congestion control. In particular, we show two cross-layer optimizations that adapt the behavior of TCP. Afterwards, we show how we use Crawler to realize a VoIP codec switching scheme to improve perceived user quality. Finally, we demonstrate Crawler's versatile monitoring and cross-layer adaptation features on two examples of jamming detection and reaction.

In particular, for all use cases we mainly highlight the following five aspects:

1. Motivation of the considered scenario and the need for cross-layer coordination.
2. Crawler's monitoring capabilities to observe varying behavior of applications and protocols under certain networking conditions in that particular scenario.
3. Proposal of a specific cross-layer coordination idea that takes advantage of earlier observations.
4. Realization of the cross-layer coordination idea with Crawler. We aim at highlighting the little effort, simplicity, and convenience to realize the solution by using Crawler.
5. Validation of proposed cross-layer coordination algorithms by demonstrating the proper reaction or adaptation to changing (network) conditions. Moreover, depending on the use case, also the evaluation of the relative improvement gained with our proposed cross-layer coordination algorithm.

For all use cases we highlight these five steps to demonstrate Crawler's versatility beginning with the use case for adapting TCP's congestion window.

4.1 Use Case: Manipulating TCP's Congestion Window and Application Behavior

The aim of the first use case[1] is to have a rather comprehensible example to understand the interplay between abstract behavior description and concrete realization in CRAWLER. For this we observe wireless conditions and based on that adapt the behavior of TCP and a particular application accordingly.

4.1.1 Motivation

A prominent example associated with cross-layer coordination is TCP's congestion control behavior [Yu04]. In particular, in wired networks TCP considers packet loss as an indication for congestion. In such a case, TCP decreases its sending rate to avoid more congestion in the network. This behavior performs well in wired networks, but unfortunately leads to performance drops in wireless networks due varying link characteristics. In highly dynamic environments, the resources to access the shared medium are limited and vary over time. In addition, higher loss probability is an important issue to handle. In literature many works have shown that the availability of cross-layer information, such as the information from the link layer, helps to improve TCP's performance [Yu04, RSI02, RNS13]. We wanted to follow a similar approach to improve TCP's congestion control algorithm in an IEEE 802.11g networks by using CRAWLER.

4.1.2 Setup and Monitoring

We first implemented two stubs to observe (i) the RSSI given by the Atheros WLAN card running the ath9k driver [ath] and (ii) the send congestion window "`snd_cwnd`" variable of TCP which is adapted by the sender depending on the number of bytes the sender can still send without requiring an acknowledgement from the receiver and the congestion state of the network. Afterwards, we set up two PCs where both are placed in a small office room but separated by a corridor (20 m range) in the computer science building E1 located at RWTH Aachen University. One PC was running CRAWLER and was equipped with a WLAN network card. The other PC was placed at the end of the corridor and connected to the router via Ethernet to avoid the impact of an additional wireless connection. Throughout our experiment, we ran iperf [TQD+04] to continuously create TCP traffic. Please note that we used the default settings for generating TCP traffic with Iperf.

We monitored both variables (i.e., RSSI and `snd_cwnd`). We observed that at certain RSSI levels the `snd_cwnd` decreased significantly and started to slowly increase again for high RSSI values. This type of growth is also known as the slow start algorithm of TCP [STE97]. The behavior is reflected in Figure 4.1 between 6 and 8 seconds. Here, even for short drops of the RSSI value, the `snd_cwnd` is also dropping and rising after a while.

[1] The content of this use case is partially based on the joint work with Jens Otten, Florian Schmidt and Klaus Wehrle published in "Towards a Flexible and Versatile Cross-Layer-Coordination Architecture", 29th International Conference on Computer Communications (INFOCOM 2010) [AOSW10].

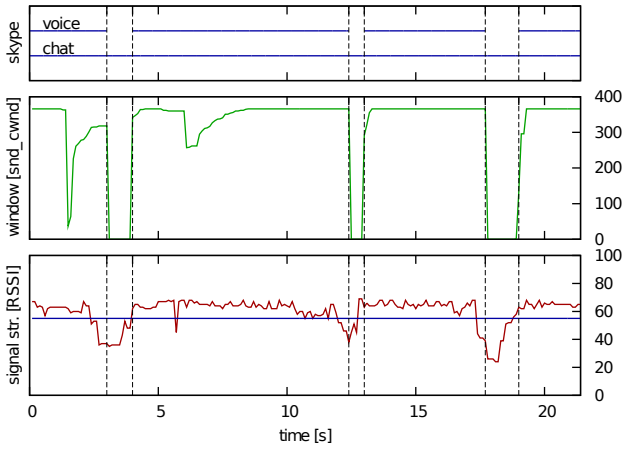


Figure 4.1 Results from our testing setup. From bottom up: (1) The measured RSSI and the threshold value of 55. (2) The congestion window, which (on three occasions) is set to 0 when the recent RSSI average is less than the threshold, and re-released when the average rises again. (3) On top, Skype chat is always available, VoIP is deactivated at low RSSI.

4.1.3 Cross-Layer Coordination Approach

Based on our observation, we defined a fixed threshold for the RSSI values to trigger two mechanisms if the value falls below that threshold: (i) storing the send congestion window `snd_cwnd` and (ii) informing a particular application. After the RSSI values recuperate, i.e., are for a while again above the predefined threshold, we again trigger two functionalities: (i) restore the stored value of `snd_cwnd` and accordingly avoid TCP's slow start mechanism and (ii) again inform an application.

The reason for informing an application is that we do not only wanted to control a protocol but also an application at the same time when certain conditions are satisfied. For this, we used the application Skype, which provides a scriptable API [Ltd08], and created a shell script to start and stop a UDP voice stream during the test, which acted as the application endpoint of our cross-layer setup. Note that the other PC was also running Skype and was connected to a WLAN access point (having Internet connectivity) via Ethernet and served as the other endpoint of our connection endpoint. The idea here is (1) to stop the VoIP conversation under poor wireless link conditions, i.e., RSSI is below a given threshold, (2) send a chat message to inform the callee about poor wireless link conditions and (3) start the VoIP call again after recovered wireless link conditions.

4.1.4 Realization with CRAWLER

We already described the configuration of this use case as an introductory example in Section 3.4.1.1. The used configuration is shown in Listing 3.1 and its graphical representation in Figure 3.5. We used a stub to read the RSSI value. The `History` FU that read the value was triggered every 100 ms by a `Timer`, and kept the last four values. `IfLess` monitored the values, and if the average of these dropped below a certain threshold (in our case 55), it triggered two notifications. The congestion window of ongoing TCP connections was set to zero (by setting the variable `snd_cwnd` in `tcp_sock` in the Linux kernel), and Skype was notified to stop its VoIP

stream, leaving only text chat. If the average increased above 55 again, the congestion window was reset to the previous value, and Skype was allowed to use VoIP again.

4.1.5 Validation

In Figure 4.1 three marked areas indicate the times where the notifications took place, roughly between three and four seconds, 12 and 13 seconds, and 17 and 18 seconds. The drop of the congestion windows to zero (caused by a drop of RSSI which we achieved by putting tinfoils on top of the antennas) and its recovering (after removing tinfoils again) are clearly visible shortly after 13 and 18 seconds respectively. Furthermore, the uppermost part of the figure visualizes how our signaling stopped and reactivated voice communication at the same time.

4.1.6 Summary and Discussion

This use case demonstrated how convenient and simple it is to monitor and realize a cross-layer coordination algorithms by using few lines of our rule-based configuration language. Based on our observations, we established a cross-layer coordination algorithm that based on a variable from the MAC layer (RSSI) adapted the behavior of two different layers, namely TCP and application layer. The major effort that was involved with this use case was the implementation of the stubs. However, we will see in the next use case the reuse of these stubs. Thus, the implementation effort for stubs is required only once for a developer, but afterwards they can be reused several times for different cross-layer coordination algorithms.

4.2 Use Case: Switching TCP's Congestion Control Algorithm

In this use case[2] we again demonstrate CRAWLER's versatility and convenience to discover behavior and performance differences of TCP. We analyze the performance of different TCP congestion control algorithms under different link conditions. Based on our observations, we propose a cross-layer coordination algorithm that based on available link conditions switches TCP's congestion control algorithm at runtime and without reestablishing the connection. This use case again demonstrates the power of CRAWLER's monitoring capability and CRAWLER's runtime flexibility feature to modify protocol behavior. Moreover, these use cases demonstrate our application support feature (see Section 3.4.2) to inject (at runtime) cross-layer coordination algorithms provided by an application into the system.

[2]The content of this use case is partially based on the joint work with Muhammad Hamad Alizai, Florian Schmidt, Hanno Wirtz, and Klaus Wehrle published in "Harnessing Cross-Layer Design", Elsevir Ad-hoc Networks Journal, November 2013 [AAS+14]. The aforementioned journal paper in turn is based on the joint work with Florian Schmidt, Hammad Alizai, Tobias Drüner and Klaus Wehrle published in "CRAWLER: An Experimentation Platform for System Monitoring and Cross-Layer-Coordination", 13th International IEEE Symposium on a World of Wireless, Mobile, and Multimedia Networks, 2012 (WoWMoM'12) [ASA+12].

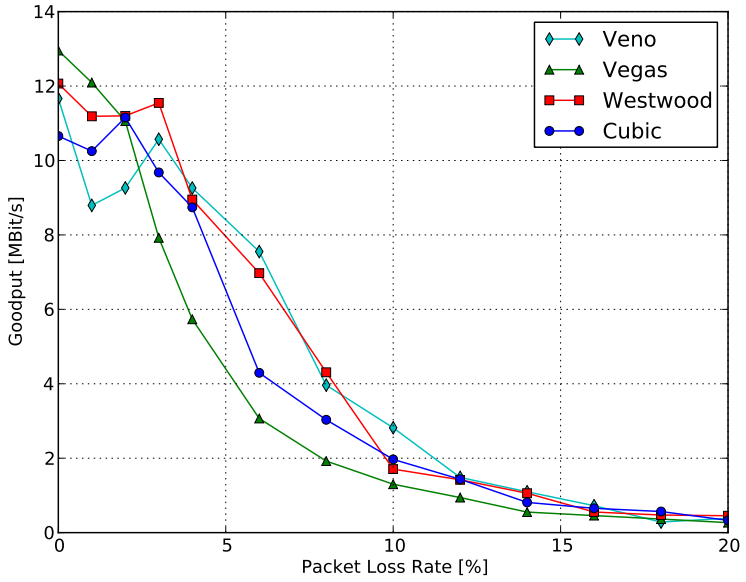


Figure 4.2 Performance comparison of different TCP congestion control algorithm for varying packet loss rates.

4.2.1 Motivation

Many congestion control algorithms have been proposed in the past for TCP, such as Vegas [BP95], Reno [MLAW99], Westwood [MCG+01], Veno [FL03], and CUBIC [HRX08]. These algorithms perform differently well depending on environmental conditions and fairness requirements (between different flows). However, we wanted to analyze how well these algorithms perform under different packet loss rates (PLR) in a WLAN network.

4.2.2 Setup and Monitoring

Our test setup consists of two PCs. One PC runs Crawler and is equipped with an 802.11g WLAN card. We use Iperf [TQD+04] to create TCP traffic, and netem [Hem05] to create different packet loss rates (PLRs) and to produce repeatable results in order to stress test our architecture. The other PC connects to an 802.11g WLAN access point and serves as the destination for Iperf traffic. Again, similar to the previous scenario, the two PCs are placed in a small office room but separated by a corridor (20 m range) in the computer science building E1 located at RWTH Aachen University.

However, we model different loss conditions and measure TCP goodput via Iperf. We monitor four different TCP congestion control algorithms: Vegas, Veno, Westwood and CUBIC. The results of our test runs are shown in Figure 4.2, where each mark in the graph corresponds to one test run. All test runs were conducted sequentially at daytime. While under low packet loss rates all congestion control algorithms perform similarly good, at around 4 % PLR and above Veno and Westwood performed significantly better.

As we observed many different neighboring WLAN networks which might have influenced our results, we have conducted several test runs and observed always that

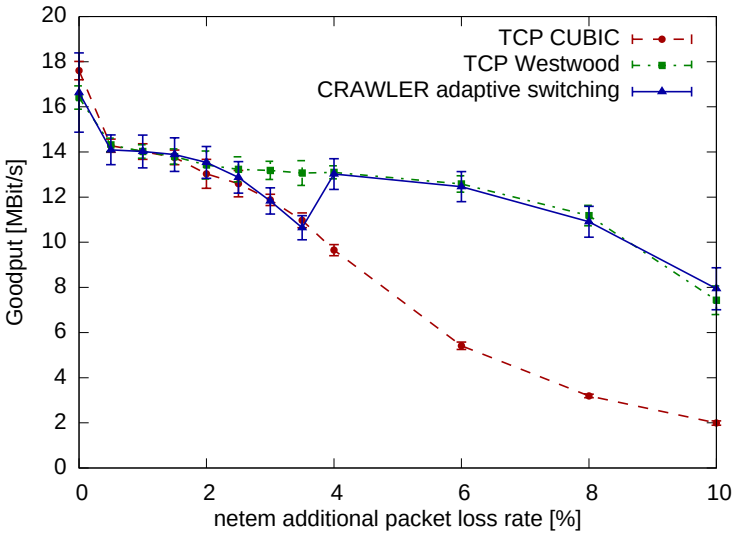


Figure 4.3 Adapting TCP's congestion control: The switch from CUBIC to Westwood is performed at a packet loss rate of 4 %. The error bars represent the 95 % confidence interval of ten repeated experimental runs. CRAWLER successfully adapts the TCP behavior.

Westwood and Veno performed better for higher packet loss rates (i.e., 4 % – 12 %). As the testing effort for all TCP congestion algorithm under different PLRs takes long time when aimed at collecting enough data for high confidence, we opted for the two congestion control algorithms CUBIC and Westwood to perform further analysis. We selected CUBIC since it is the standard congestion control algorithm in the Linux kernel since 2.6.19 due to its superior performance and fairness properties under different network conditions. Although we observed that Veno and Westwood preformed similarly good with high PLRs, we opted for Westwood as it is specifically developed for wireless communications (such as WLAN), and provides better throughput under challenging wireless conditions as it not very sensitive to random errors [CGM+02].

To obtain comparison results, we measured the performance of TCP CUBIC and Westwood in our specific setup with different PLRs. To also decrease the impact of neighboring WLAN networks, we measured at night. Due to these additional effects and the fact that packet losses during a transmission influences the congestion control behavior, we conducted ten test runs of two minutes for each data point. The results for both algorithms CUBIC and Westwood are shown in Figure 4.3. It can be seen that Westwood outperforms CUBIC for high packet loss rates. The variation in the results (specified by the 95 % confidence intervals) can be attributed to different environmental conditions observed during the campaign of ten repeated experimental runs in an indoor environment with several co-exiting WLANs deployments in the same frequency range.

4.2.3 Cross-Layer Coordination Approach

The goal of this coordination approach is to dynamically switch between the two different congestion control algorithms CUBIC [HRX08] and Westwood [MCG+01] depending on the underlying network conditions. Note that the switch should be conducted at runtime and without reinitializing the TCP connection. In particular,

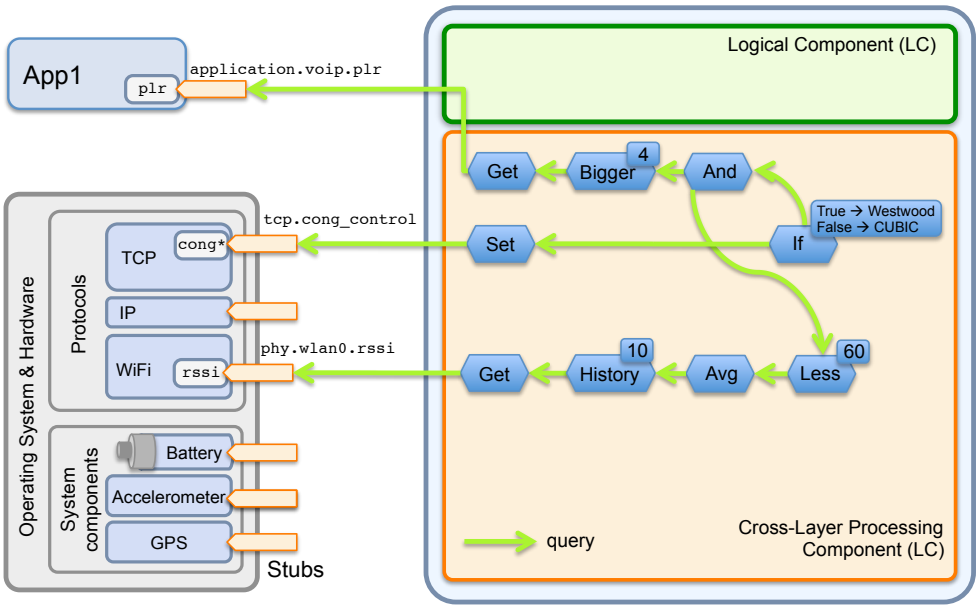


Figure 4.4 Graphical view of the TCP congestion control switching scheme: It changes TCP's congestion algorithm based on packet loss rate (PLR) and RSSI values. If the PLR is high and the RSSI is low, TCP's congestion control algorithm is set from TCP CUBIC to TCP Westwood. If either of the conditions is not satisfied, the congestion control algorithm is set back to TCP CUBIC.

it should be monitored whether the packet loss rate and the RSSI value fall below the specified thresholds. Based on that, TCP should switch from CUBIC to Westwood congestion control. A switch back to CUBIC should be initiated when the network conditions improve again, i.e., high PLR and RSSI.

4.2.4 Realization with CRAWLER

The complete configuration is presented in Listing 3.6 and its graphical representation is shown in Figure 4.4. The configuration specifies that the switch from CUBIC to Westwood shall be performed when the average RSSI of the last ten values falls below 60 dBm and the packet loss rate goes beyond 4 (which is a significant PLR for TCP). The choice of the thresholds was based on observations of several test runs. However, the configuration including the cross-layer coordination algorithm is provided by an application and can be activated at anytime through user interaction at the console by setting a parameter. The stub to change the congestion control algorithm at runtime is discussed in detail in Section 3.4.3.2 (see also Figure 3.8).

4.2.5 Validation

The effect of this coordination is shown in Figure 4.3 and indicated by CRAWLER adaptive switching. Note that a switch at a lower PLR of 3 % could also improve performance of the optimization. However, as our main goal is to show an exemplary optimization, the switch at 4 % already highlights its effects.

Figure 4.5 shows our results for a longer experimental run, and also highlights the possibility to load rules at runtime. For the first 60 seconds, we did not load the

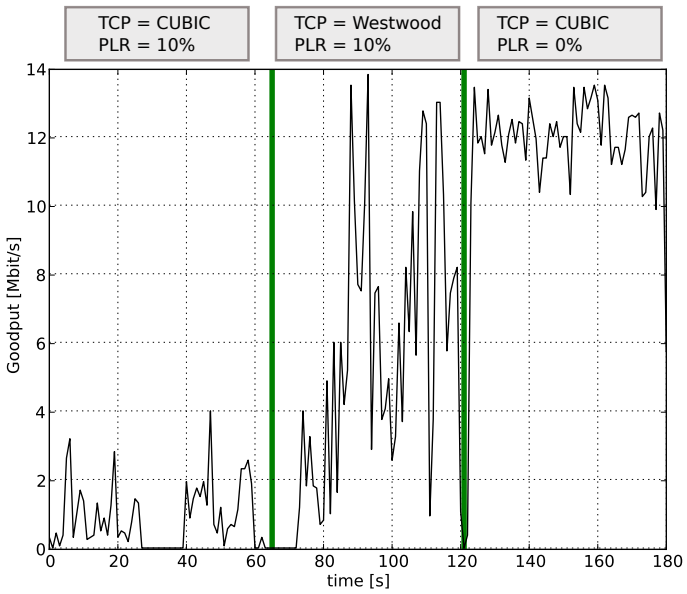


Figure 4.5 Goodput of a TCP transmission over time under varying environmental conditions and congestion control algorithms. The cross-layer coordination algorithm is loaded after 60 seconds which triggers the switch from CUBIC to Westwood. The switch back to CUBIC is triggered when the packet loss rate (PLR) falls below the application-specified threshold of 4 %.

cross-layer coordination algorithm into the CPC, as depicted by the low TCP goodput achieved during this time. The coordination algorithm is loaded at 60 seconds which triggers the switch from CUBIC to Westwood and subsequently improves the goodput. Similarly, at 120 seconds, when we adjusted the PLR with netem below the 4 % threshold, TCP switches back to CUBIC and thus achieves a consistently higher goodput.

In addition to our coordination algorithm, we also investigate if our on-the-fly algorithm change produces undesirable side effects. For example, we were interested in the behavior of TCP's congestion window (`cwnd`) across different congestion control algorithms which could have been affected by our cross-layer coordination algorithm. To monitor the behavior during the algorithm switch, we monitored the `cwnd` variable via CRAWLER's monitoring application (more details are given in Section 3.4.2 and Listing 3.5) by simply executing `monitorapp 'transport.tcp.cwnd'` in the console. In contrast, a manual setup would require changes to the kernel to introduce hooks and to create an interface to access the collected data. CRAWLER relieves the developer from these steps and expedites the testing and monitoring of variables and setups. With the monitoring application we have not observed specific side effects, but nonetheless were able to keep track of the variable behavior.

4.2.6 Summary and Discussion

In this use case we dynamically switched between parts of a protocol, namely between different TCP's congestion control algorithms, depending on wireless channel conditions. To achieve this, we used CRAWLER's monitoring capability which helped remarkably to discover the differences in performance for a set of TCP congestion control algorithms.

After discovering the relative differences under different link conditions, we were able to conveniently express the desired cross-layer coordination algorithm. A 15-line configuration can be used to adapt TCP's congestion control without the need of re-initializing the end-to-end connection, which highlights CRAWLER's manageability feature (see Section 3.4.1). We also showed that a cross-layer coordination algorithm can easily be added and removed at runtime, even an application can inject coordination algorithms emphasizing both CRAWLER's runtime flexibility (see Section 3.4.3) and application support (see Section 3.4.2) feature. Note that we can configure the rules to switch between any congestion control algorithm. Similarly, we can also modify the thresholds for when to switch congestion control algorithms. Additionally, we can define when to feed the coordination algorithm into the system, for instance, on application start or triggered by a user interaction demonstrating CRAWLER's context adaptability feature (see Section 3.4.4).

In addition to our proposed cross-layer coordination algorithm, we utilized the convenient monitoring application. Please note that with this monitoring application, as presented in Section 3.4.2, further parameters such as the congestion control window (that we also used in the previous use case) can easily be observed with a simple instruction in the console.

4.3 Use Case: VoIP Codec Switching

In this use case[3] we propose a cross-layer coordination targeting only at improving the application performance defined by the Quality of Experience (QoE) of a user. The proposed cross-layer coordination improves the user-perceived quality of Voice-over-IP (VoIP) by automatically switching the codec during a phone call. This automatic switching is based on the observed network parameters packet loss, jitter, and bandwidth. These parameters strongly impact the user-perceived quality of a VoIP call [KP09]. To collect all relevant network parameters, it was necessary to use the application WBest [wbe] which in turn required a coordination with Linphone (inter process communication). By using CRAWLER we show how simple it is to realize coordination among different applications.

Similarly as in previous examples, after providing some base knowledge about the problem space, we use the power of CRAWLER's monitoring capability to compare the speech quality of a set of standard VoIP codecs given different network conditions. Subsequently, based on our observations, we propose an adaptive end-to-end based codec switching scheme that fully conforms to the Session Initiation Protocol (SIP) standard [RSC+02]. Afterwards, we show how we easily configured CRAWLER to achieve the desired cross-layer coordination. Finally, before discussing related work, our evaluation with a real-world prototype based on Linphone demonstrates that our codec switching scheme adapts well to changing network conditions, improving overall speech quality.

[3] The content of this use case is partially based on the joint work with Florian Schmidt, Elias Weingärtner, Caj-Julian Schnelke and Klaus Wehrle published in "An Adaptive Codec Switching Scheme for SIP-based VoIP", 12th International Conference on Next Generation Wired/Wireless Networking (New2An'12) [ASW+12] and Caj-Julian Schnelke's bachelor thesis [Sch11].

4.3.1 Motivation

The constantly changing dynamics of wireless and mobile environments are a great challenge for VoIP communications. Current VoIP software has only limited capabilities to deal with these dynamics. They typically support a number of codecs that differ in the optimal speech quality depending on prevalent network parameters jitter, packet loss rate and bandwidth. However, these VoIP clients typically negotiate one single codec for the entire duration of the call. While these codecs might have the capability of self-adaptation to a limited degree (e.g., by adjusting their internal codec parameters to increase or decrease redundancy dependent on marginal network parameter changes), existing VoIP clients abide by their initial codec choice. Hence, they often apply a codec that is not well suited for the present network situation although better codec choices would be available. To our knowledge, none of the existing VoIP clients implement an adaptive strategy to switch the session's speech codec for changing network conditions.

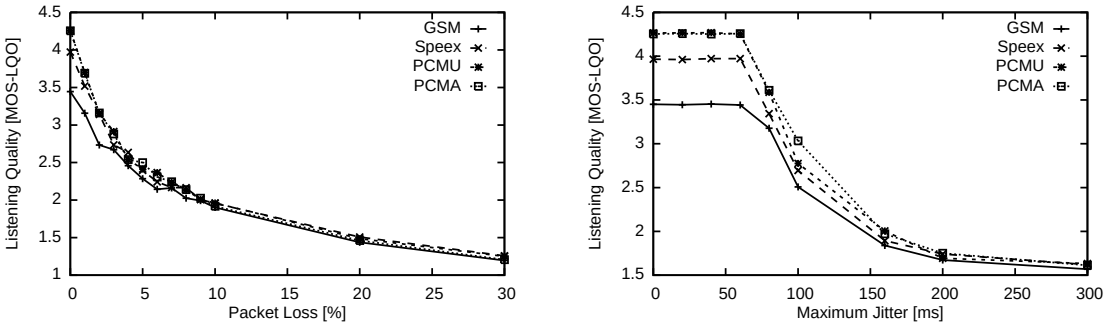
In the following, we present our test setup and analyze the influence of different network conditions such as varying packet loss, jitter, and available bandwidth on the speech quality of four different open codecs.

4.3.2 Setup and Monitoring

In order to objectively evaluate perceived speech quality, we rely on the PESQ tool [pes, ITU01] that rates perceived quality with a MOS-LQO (mean opinion score – listening quality objective) score ranging from 1 (bad quality) to 5 (excellent quality). As reference file, we used the ITU-T test file u_af1s03.wav (female voice speaking two sentences with a short pause between them) [tes]. For VoIP communication, we employed the VoIP client Linphone [lin] as it is open source, SIP conform and already offers a fair set of commonly used speech codecs.

Our test setup consists of two notebooks running Linux Ubuntu 10.04 and a router in between. The two notebooks are connected to the router via a 100 Mbit/s Ethernet connection. A wired connection was chosen to avoid uncontrollable wireless interference that can impact PESQ-MOS results. The whole test was automated to achieve repeatability and to conduct manifold tests for credible results. We used the following four codecs: (1) Speex (8 kHz version), (2) GSM-FR, and G.711, also known as Pulse Code Modulation (PCM) with its two variants (3) PCMU and (4) PCMA.

We regard (1) packet loss, (2) jitter, and (3) available bandwidth as the factors that define the current network condition. We opted for these parameters as they are known for having a strong impact on VoIP communications [KP09]. All tests followed the same order. We used netem [net] to insert jitter or packet loss into the connection, and employed traffic shaping via the token bucket filter [tbf] to reduce the available bandwidth. For each combination of codec and a certain packet loss/jitter/bandwidth, we repeated the experiment one hundred times. One notebook, the sender, initiates the call and transmits the ITU-T test file via Linphone. The other notebook, the receiver, answered the call and recorded the audio output.



(a) Influence of packet loss on perceived speech quality for several voice codecs.

(b) Influence of delay jitter on perceived speech quality for several voice codecs.

Figure 4.6 All codecs are similarly impacted, no benefit achievable by switching codecs.

Figures 4.6(a), 4.6(b), and 4.7 show our results for the codecs GSM, Speex, PCMU (PCM with μ-law encoding), and PCMA (PCM with A-law encoding) under different network conditions manipulated with the help of netem [net] and token bucket filter [tbf] traffic shaper.

Figure 4.6(a) shows the results for varying **packet loss** rates that are manipulated with the help of netem. The results for the different codecs are very close to each other with no codec having a real advantage over another. As the values above 10 % packet loss are already too bad for a real communication, codec switching does not provide any benefit.

Jitter was also manipulated with netem. The performance was tested with a packet delay following a normal distribution and a maximum variation from ± 20 ms to ±100 ms in 20 ms steps and additionally for ±160 ms, ± 200 ms, and ± 300 ms. The chosen delay variations ultimately led to packet reordering. Figure 4.6(b) shows that the limit of the jitter buffer of all codecs is reached roughly between 60 ms and 80 ms. Beyond this point, the speech quality degrades sharply for all codecs, and their MOS-LQO ratings converge.

In order to limit the **bandwidth**, we used the token bucket filter. Each audio codec was tested with upstream bandwidths from 10 kbit/s up to 100 kbit/s in 10 kbit/s steps. The results can be seen in Figure 4.7. The divergence between the value reached in 10 kbit/s and the 20 kbit/s test cases of the PCMU and the PCMA codecs may be due to the extremely low quality output. At these low bandwidth values output is so garbled due to data loss that quality assessment fails to properly evaluate the speech. From 20 kbit/s to 30 kbit/s the GSM codec performs slightly better than the Speex codec. Up from 40 kbit/s until 90 kbit/s, the Speex codec outperforms all other investigated codecs. After 90 kbit/s the PCMU and the PCMA codecs perform better than the other. Further tests are not necessary as neither the Speex codec nor the GSM codec are expected to outperform the PCMU or the PCMA codecs at higher bandwidths.

The tests demonstrate that with the available set of codecs, bandwidth is the best characteristic to select a codec, since it shows the highest impact on the quality. In all remaining cases the codecs behave similarly well, resulting in no need for a codec change. The experiments also show that the GSM codec can be dropped as

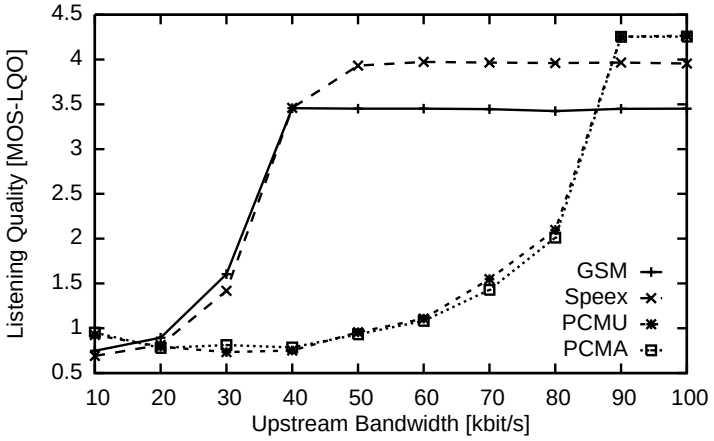


Figure 4.7 Influence of available bandwidth on perceived speech quality for several voice codecs. Codec performance clearly depends on bandwidth, so codec switching depending on available bandwidth is sensible.

a useful audio codec because it never outperforms the other codecs in any of the experimented network conditions in a way that makes it reasonable to use.

4.3.3 Cross-Layer Coordination Approach

Even though PCMA and PCMU always perform marginally better than Speex in the packet loss and jitter tests, we have to consider the low MOS-LQO value of the two PCM codecs at upstream bandwidths smaller than 90 kbit/s. Here the selection of Speex as a codec provides a better MOS-LQO value.

As a result, we draw the conclusion that switching between Speex and PCM[4] on available bandwidth is reasonable. In particular, we should use Speex for every bidirectional bandwidth between 0 kbit/s and 180 kbit/s and PCM above 180 kbit/s. Note that, since we only investigated a limited set of codecs, this proposed switching decision is rather simple. The switching scheme proposed in the following will, however, also work with more sophisticated switching decisions and more codecs.

4.3.4 Realization with CRAWLER

We used the VoIP client Linphone which uses SIP to establish, modify and terminate sessions where two or several more participants are involved [RSC+02]. After a session is established, the Real-Time Transport Protocol (RTP) is used to transport VoIP data to the participant. RTP is usually used in conjunction with its helper protocol Real-Time Control Protocol (RTCP) that provides periodic feedback about the reception quality. However, to implement a codec switching scheme using CRAWLER, we used WBest [wbe] to measure bandwidth during a VoIP session since Linphone does not offer such measurements. CRAWLER gets the IP addresses of a call from Linphone and provides it to Wbest's bandwidth measurements (cf. line 1 of WBest configuration in Listing 4.1). Similarly, it gets bandwidth measurements

[4] PCMA or the PCMU codec curves are very similar in all cases, therefore from now on we use only the term PCM to indicate both

from Wbest, calculates a sliding window based average, and provides it to Linphone to initiate an appropriate codec switching (cf. Linphone configuration in Listing 4.2). Note that the Linphone configuration shows a threshold of 175 for codec switching which we selected marginally lesser than the threshold 180 in order to avoid unnecessary ping pong effects. These two configurations highlight how conveniently we achieved a cooperation or inter process communication among applications.

```
1 theIPget:get("app.linphoneapp.theip")
2 theIPset:set("app.wbestappsnd.theip",theIPget)
3 makeMeasureget:get("app.linphoneapp.makemeasure")
4 makeMeasureset:set("app.wbestappsnd.makemeasure",makeMeasureget)
5
6 theIPget->theIPset;
7 makeMeasureget->makeMeasureset;
```

Listing 4.1 CRAWLER configuration used by WBest to access the required IP address to perform bandwidth measurements.

```
1 appnv:get("app.wbestappsnd.newvalue")
2 appbwidthget:get("app.wbestappsnd.bwidth")
3 bwhistory:history(appbwidthget,3)
4 avgbwidth:avg(bwhistory)
5 appwhich:if(less(get(app.linphoneapp.makemeasure),1),175,avgbwidth)
6 appbwidthset:set("app.linphoneapp.bwidth",appwhich)
7
8 appnv->bwhistory;
9 appnv->avgbwidth;
10 appnv->appbwidthset;
```

Listing 4.2 CRAWLER configuration used by Linphone to obtain the bandwidth measurements provided by WBest and to decide when to perform a switch between codecs.

In Figure 4.8 we graphically show the decision graph for our codec selection scheme. In a first step we have to ensure that we use the best performing codec for the current bandwidth when we start the call. Since Linphone does not offer a bandwidth measuring functionality, we used WBest [wbe]. Although this causes longer initiation time (around 1 second) for a call, we believe that such a short time is not annoying for a user if it is at the beginning of a call. If the available bandwidth exceeds our threshold ($180\,kbit/s$), we send out a SIP-INVITE message with PCM as a codec. If it is smaller than our threshold, we offer only the low bandwidth consuming codec Speex.

Observing a decrease in the bandwidth is fairly easy. If the codec needs more bandwidth than what is available, this ultimately leads to packet loss. RTCP reports already provide information about packet loss. Packet loss information is directly accessible from Linphone. If the reported packet loss increases above our threshold of 10 %, we switch from the high bandwidth consuming codec to the low bandwidth consuming one.

On the other hand, an increase of available bandwidth is harder to discover. A packet loss of 0 % does not necessarily mean that we have enough bandwidth to switch from a codec with low bandwidth consumption to one with a high bandwidth consumption. So once again we have to use WBest to obtain the currently available bandwidth. We measure the available bandwidth every 3 seconds as every measurement with WBest generates overhead. This is a good trade-off between reaction

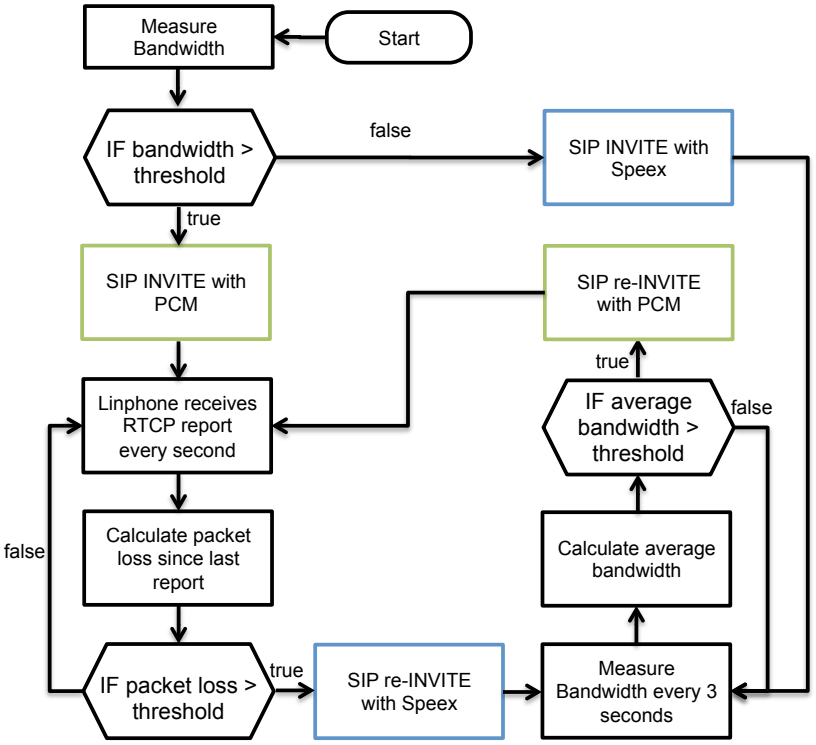


Figure 4.8 Flowchart of the adaptive codec switching scheme.

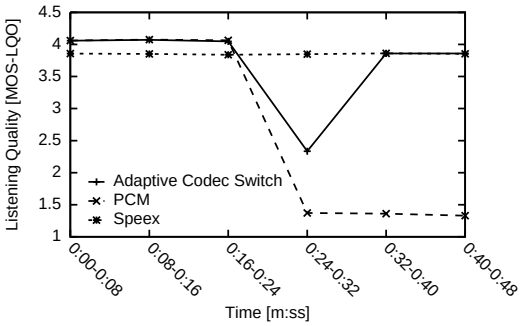
speed and overhead. When using PCM we do not run WBest because there is no benefit from finding out whether even more bandwidth is available. To ensure that a bursty short-term increase in bandwidth does not lead to a premature codec change, we calculate the bandwidth over a sliding window history of three measurements. We switch from Speex to PCM conservatively because switching to PCM at a bandwidth below 180 $kbit/s$ leads to considerable degradation of the quality and should be avoided. To coordinate the change with the callee, we used the SIP re-INVITE message defined in the SIP standard. This message facilitates adding, removing, or modifying a session.

4.3.5 Validation

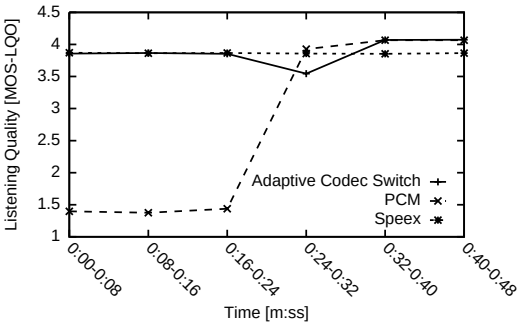
We investigate the speech quality achieved by our codec selection scheme. We compare our cross-layer coordination for codec switching with a static use of either Speex or PCM. To highlight the effects of the switch in either direction, we present two evaluation cases: a bandwidth increase and a bandwidth decrease situation.

In the bandwidth decreasing case, the bandwidth is limited to 200 $kbit/s$ initially, and is further reduced to 65 $kbit/s$ after 24 seconds. Conversely, in the bandwidth increasing case, the available bandwidth is increased from 65 $kbit/s$ to 200 $kbit/s$ after 24 seconds. We use the same ITU-T test file for all experiments. We loop that test file six times to create a 48-second test file out of the 8-second sample.

Note that each case was conducted 20 times with each approach (Speex, PCM, and our codec switch scheme). The total MOS-LQO value for the whole test period of 48 seconds measured by the PESQ tool does not provide a fair comparison method, since the longer the test file is the lower is the influence of the switching gap from one codec to another. Therefore, we decided to divide each record to 8 second



(a) MOS-LQO values for decreasing bandwidth. At 24s, bandwidth was reduced from 200 kbit/s to 65 kbit/s, and our scheme switched from PCM to Speex.



(b) MOS-LQO values for increasing bandwidth. At 24s, bandwidth was increased from 65 kbit/s to 200 kbit/s, and our scheme switched from Speex to PCM.

Figure 4.9 MOS-LQO comparison for our codec switching scheme and pure PCM and Speex codec in case of changing bandwidth conditions.

chunks and compare those chunks with our original 8 second test file. This has the further advantage that it also shows the perceived quality over the duration of the experiment, instead of only one aggregated value.

The results for the **decreasing bandwidth** case are shown in Figure 4.9(a). As expected, the speech quality of the Speex codec stays constant throughout the test, because the bandwidth limitation to $65\,kbit/s$ is still above Speex's requirements. On the other hand, PCM shows a strong degradation of quality after the bandwidth reduction. Please note how CRAWLER correctly chooses PCM as the initial codec. The effect of our codec switching scheme can be clearly seen. The temporary degradation between 24 and 32 seconds can be attributed to two factors. (1) The bandwidth decrease is detected due to frame losses, which reduces the listening quality for the time span before the switch takes place. (2) Linphone's current implementation reacts to a re-INVITE codec switch with a small playback gap of about $200\,ms$, which also decreases the perceived quality.

Similarly, Figure 4.9(b) shows the results for the **increasing bandwidth** case ranging from $65\,kbit/s$ to $200\,kbit/s$ after 24 seconds. Again, Speex's speech quality stays constant. PCM benefits from the increased available bandwidth, which leads to a strong quality improvements after the 24-second mark. Our codec-switching scheme correctly decides on the better codec to use at any given point in time by choosing Speex as initial codec, and switching to PCM after the bandwidth change at 24 seconds. The slight degradation between 24 and 32 seconds can be attributed to Linphone's playback gap, which temporarily decreases the perceived quality.

To conclude our evaluation, our test shows that our codec switching scheme selects the specified codec properly at the beginning and during the communication. In addition, we can see how it improves listening quality compared to a static codec choice, except for the short time of the switch itself. Note that we do not expect to change codecs very frequently, so these evaluation results overemphasize the temporary quality loss during the switch; in a real setup with long conversations and only occasional codec switches, the overall quality improvement will strongly outweigh the short degradations.

4.3.6 Related Work

Related work on adaptive codec switching typically focuses on adaptation to degrading network conditions, and only discusses adaptation to improving conditions briefly or not at all. Furthermore, to the best of our knowledge, none of these approaches provides a solution for choosing the optimal codec at the beginning of the call. In [CDMM00] the authors suggest to use packet delay as an indicator to select codecs. Their reasoning is to detect network congestion that way, and to preemptively switch codecs before prohibitively high packet loss occurs. However, their purely analytical approach does not focus on speech quality as a metric; therefore it is not clear whether such a switching approach actually would improve quality. In [NHS05] packet loss is used as an indicator to switch the codec under degrading as well as improving conditions. However, how and when to infer from low packet loss that additional bandwidth is available is not discussed. The adaptation to improving conditions is not evaluated either, so this question remains open. Similarly, the authors of [YCLT08] use packet loss as an indicator for both degrading and improving channel conditions. Furthermore, they propose a handover scheme between different types of networks (e.g., WiFi and WiMAX) that also takes signal strength into account. Again, no evaluation for adaptation to improving network conditions is presented. Both [NHS05] and [YCLT08] employ a SIP re-invite technique similar to ours. The authors of [MK07] propose an adaptation that combines the goals of quality and security. They continuously monitor the MOS via a no-reference scoring algorithm and then decide on which codec to use, whether to introduce additional forward error correction, and how much security overhead they can introduce without compromising quality. However, the authors do not fully address the increasing bandwidth case. Moreover, their design has neither been implemented nor tested, so it is unclear how well their approach would work in reality.

4.3.7 Summary and Discussion

This use case demonstrated a cross-layer coordination algorithm provided by an application to improve the user perceived quality of VoIP by automatically switching the codec during a phone call. To achieve this, we used CRAWLER's monitoring feature to first analyze the speech quality of several standard VoIP codecs for different network conditions. The results of this analysis showed that bandwidth is the most relevant metric for perceived quality.

Based on empirical evidence, we designed an adaptive codec switching scheme that fully conforms to the SIP standard and integrated it into the Linphone [lin] VoIP client. For the realization we used CRAWLER to coordinate the two applications Linphone and WBest. Depending on available bandwidth, our adaptive codec switching scheme performs three tasks: (1) choice of the currently best performing codec before the actual communication starts, (2) change to a low bandwidth consuming low quality codec when the packet loss increases, and (3) change to a high bandwidth consuming high quality codec when the bandwidth increases.

Our evaluation shows that our solution enhances the perceived listening quality compared to a static codec choice at the beginning of the call. It improves listening quality compared to a static codec choice, except for the short time required to

perform the switching operation. However, in a real setup with long conversations and only occasional codec switches, we expect the overall voice quality improvement to strongly outweigh these short degradations.

4.4 Use Case: Dynamic Adaptation of Jamming Detection and Reaction Strategies

With this use case[5] we present a powerful and convenient jamming detection and reaction framework to dynamically (un)load own jamming detection and reaction strategies which underlines CRAWLER's runtime flexibility and extensibility feature. We used this framework to dynamically coordinate a jamming detection and reaction mechanism targeting at different layers. In particular, at runtime we feed two different cross-layer coordination algorithms into the system provided by an application. We propose a "naïve" detection strategy for 802.11 systems. Our evaluation demonstrates, in a real-word outdoor and mobile scenario, how we successfully detect a jammer and initiate a reaction strategy.

4.4.1 Motivation

The shared nature of the wireless medium enables a special kind of security attacks, the so-called jamming attacks which target at disturbing the communication sometimes even to a degree where communication is not feasible anymore. Past and recent research shows the effectiveness of jamming in the context of 802.11 and 802.15.4 systems [BKL+08, PnAG12, XTZW05], as well as in the context of cellular networks [HHT+02, SSC11]. With an increasing demand for (time-critical) machine-to-machine applications and safety-critical applications in vehicular networks, the importance of jamming-detection and according reaction is expected to increase in the future.

However, to effectively cope with the problem of jamming without costly and special hardware such as spectrum analyzers, monitoring potential indicators in the system that react to jamming is a necessity. Such potential indicators can be obtained at different layers (e.g., packet delivery rate at the application layer and medium access delay at the MAC layer, among others). Therefore, the use of a cross-layer architecture such as CRAWLER can simplify the task of collecting necessary metrics and, hence, of jamming detection [Stå00].

Although various metrics for different radio access technologies have been investigated, the field of 802.11 is not much explored. This has historical reasons, since typically when secure and reliable communication was required, radio access technologies running on dedicated hardware and bands were used [Stå00]. Nowadays, 802.11 is widely accepted and is even considered as a relevant technology in military scenarios [KSHH04, Shy06]. The broad acceptance of this technology can be attributed, amongst others, to the simplicity of the technology and wide range of usage scenarios. But this fact makes the technology also very attractive for jamming attacks. This motivates the development of a reliable jamming detection approach.

[5] The content of this use case is partially based on the joint work with Dominik Denissen and published in his master thesis [Den12].

The majority of the proposed strategies [XTZW05, LKP07, SDv10] to detect jamming are customized for a specific jammer in a certain scenario. Consequently, it is difficult to use those approaches together even though this could help in obtaining a unified system that can detect the presence of a jammer in an adequate manner [PIK11]. For example, selection of the detection strategy based on the scenario and the jammer type. CRAWLER provides a good basis to combine such strategies as it allows (i) the monitoring of several protocol and system component information (see Appendix B), (ii) the convenient design of detection and reaction strategies by using CRAWLER's abstract configuration language, and (iii) the ability to add, remove and experiment (at runtime) with a set of jamming detection and reaction strategies for different scenarios and jammer types.

In the following we highlight these features by adapting a popular jamming-detection approach originally designed for IEEE 802.15.4 to the requirements of IEEE 802.11.

4.4.2 Setup and Monitoring

We aimed at reimplementing the strategy suggested by Xu et al. [XTZW05] as it is the most widely used approach in the field of jamming detection for sensor networks. But we wanted to rebuild this strategy for 802.11 by taking advantage of CRAWLER's features of accessing and monitoring parameters in the system. The strategy of Xu et al. follows a cooperative approach using the packet delivery ratio (PDR) and the radio signal strength (RSS). All sensor nodes measure their PDR and (cooperatively) exchange this metric with their one-hop neighboring sensor nodes. Existing traffic contributes to the PDR and RSS measurements. The detection strategy for this approach is shown in Listing 4.3.

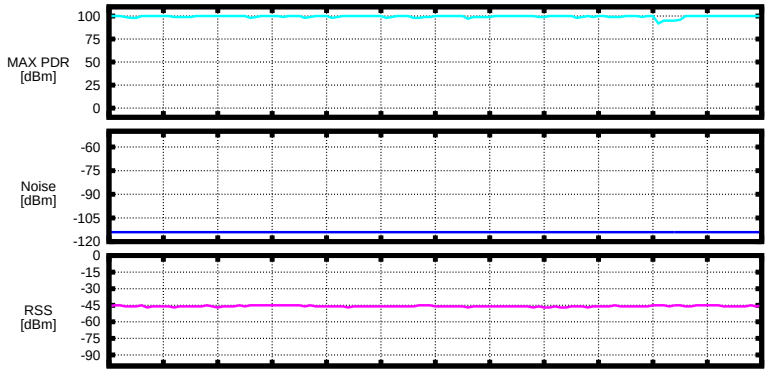
```
1 MaxPDR = max{PDR(N): N in Neighbors};
2 IF (MaxPDR<THRESHOLD1) THEN
3   SS = SampleSignalStrength();
4   IF (SS>THRESHOLD2)  THEN
5     Jammer detected;
```

Listing 4.3 Jamming detection strategy proposed by Xu et al. [XTZW05] using signal strength consistency checks in 802.15.4 networks.

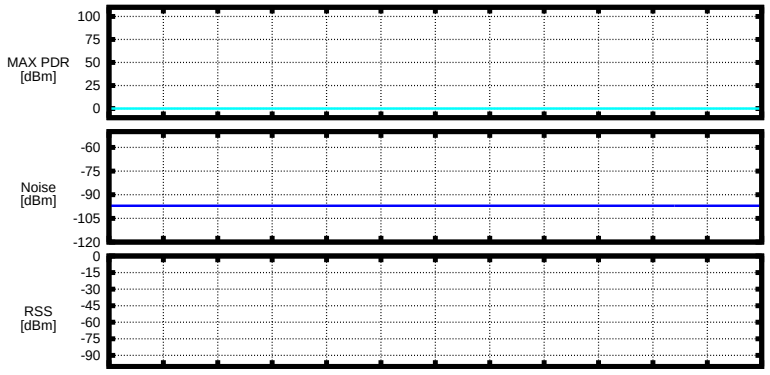
In a first step, the maximum PDR is selected from the PDR exchange messages that are continuously exchanged between all neighboring nodes. In a next step, if the maximum PDR falls below a predefined threshold, a signal strength measurement is initiated . If the signal measurement exceeds a predefined threshold, the strategy decides the presence of a jammer. The RSS consistency check helps to differentiate between a low PDR caused by a large propagation distance, e.g., due to mobility, or by the presence of jamming.

Although this approach is very practical for sensor networks, we experienced that it is not practical when directly applied to 802.11 due to how signal strength measurements are obtained. For 802.11 we observed cases where in the presence of a jammer RSS (or radio signal strength indicator, RSSI[6]) measurements are not available. In the following we describe a test setup which highlights the problems and their causes.

[6] Note that RSS is not provided by consumer network cards and the RSSI measurements are calculated differently in each brand.



(a) Jammer absent.



(b) Jammer present.

Figure 4.10 Impact of the pilot jammer on the metrics RSS, PDR, and noise. The jammer is located next to the node and prevents successful packet transmissions which leads to missing RSS calculations in IEEE 802.11 network interface cards.

Our test setup consists of two Linux PCs equipped with an 802.11g/n Atheros WLAN card running the ath9k driver. The two PCs are placed in a small office room with a distance of one meter in the UMIC building located at RWTH Aachen University. We let the two PCs build an ad-hoc network and continuously exchange messages on channel 11 within the 2.4GHz band. We used the jammer as implemented and described in [PnAG12] which we placed next to the nodes. The jammer is realized on the Wireless Open-Access Research Platform (WARP) board [12], which provides an 802.11-like OFDM physical layer featuring a 10MHz bandwidth and an output power of 18 dBm in the 2.4GHz band. In particular we used the OFDM pilot jammer which targets to jam the four pilot subcarriers of OFDM instead of the whole bandwidth (for details about the effectiveness we refer to [PnAG12]). We have also implemented an ad-hoc PDR exchange messaging scheme where all nodes exchange their currently measured PDR via broadcast with all their neighboring nodes.

Figure 4.10(a) shows the result under normal operation, i.e., when the jammer is not active. In contrast, Figure 4.10(b) shows the impact of the jammer on our metrics. After activating the jammer, the PDR dropped from 100 to 0%, RSS measurements were not available and noise increased from -115 to -100 dBm. The reasons for missing RSS values is that RSS measurements in 802.11 are coupled with packet reception. That is, RSS is calculated over the preamble of a packet and if a packet is not detected due to too low received signal strength or high interference, calculated RSS measurements are discarded.

4.4.3 Cross-Layer Coordination Approach

Based on our observation, our first idea was to decouple the RSSI measurements from packet reception. We used the ath9k driver and tried to modify the packet coupled nature, but unfortunately we ended up in the firmware which does not provide the required interfaces. The firmware of the NIC provides only a RSSI value based on valid packet preambles. If the NIC considers the transmission as a valid packet, the driver first normalizes the RSS measurements and subsequently incorporates the noise floor to obtain the RSSI. In other words, using RSS as a metric is not reasonable as it is not always available. Fortunately, we observed that noise is strongly affected in presence of a jammer as also shown in the Figure 4.10(b) and is not dependent on packet receptions. Therefore, we adapted Xu's approach to use noise instead of RSS for consistency checks leading to the jamming detection strategy shown in Listing 4.4.

```
1 MaxPDR = max{PDR(N): N in Neighbors};
2 IF (MaxPDR<THRESHOLD1) THEN
3   NN = SampleNoise();
4   IF (NN>THRESHOLD2)  THEN
5     Jammer detected;
```

Listing 4.4 Our naïve jamming detection strategy for 802.11 systems which uses PDR and noise as consistency check.

So far, we designed a detection strategy that aims at detecting a jammer, but with Crawler it is also similarly easy to design reaction strategies. After detecting a jammer for this particular scenario, we informed the headquarter (in the scope of a NATO project) via an additional (emulated satellite) link about the presence of a jammer. Moreover, in the project packets with different security classes were routed differently. Therefore, we labeled the packets at IP layer accordingly. This required a cross-layer coordination between the application that was responsible for informing the headquarter and the network layer to label only the traffic of that specific application.

In the following we present how we realized the jamming detection and reaction strategy using Crawler.

4.4.4 Realization with CRAWLER

We aimed at realizing a flexible architecture to add and remove jamming detection and reaction strategies at runtime. One major goal was the separation of detection and reaction mechanisms. In other words, we wanted to offer flexibility and control at runtime to load a user-defined set of detection and reaction strategies. Although Crawler allows to feed cross-layer coordination algorithms from an application into the system, the structured and convenient handling of these strategies needs to be provided by the application.

To provide an easy way to handle the set of detection and reaction strategies, we used the "strategy" design pattern [GHJV95]. This design pattern satisfies the mentioned requirements. It facilitates to (un)load jamming detection or reaction strategies during runtime. In addition, to have a notification of reaction strategies by detection

strategies, we used the "observer" design pattern [GHJV95]. Thus, when detection strategies decide about the presence of a jammer, preregistered reaction strategies are informed.

However, after implementation, the detection and reaction strategies can simply be called from the console. Listing 4.5 shows the general syntax for executing jamming detection and their corresponding reaction strategies.

```
1 $>./mainJammingDetection [+/-/~]s: StrategieName [parameter]
2     [+/-/~]o: StrategieName ReactionName [parameter]
```

Listing 4.5 General syntax for the execution of jamming detection and their corresponding reaction strategies from the console.

The first line enables the start of the main jamming application with a defined set of different jamming strategies and their parameterization. The second line enables to load reaction strategies which are assigned to certain strategy names specified in the first line. A particular example that we have used later in our evaluation is shown in Listing 4.6.

```
1 $>./mainJammingDetection +s: naiveDetection -i wlan0
2    -c 172.16.0.6 -s 172.16.0.255 -t 100
3    +o: naiveDetection perfSonarSoapClient -s 172.16.0.4:8080
```

Listing 4.6 Particular call for starting our naïve jamming detection and the reaction strategy to inform the headquarter.

In the first line we load our naïve jamming detection strategy followed by its parameterization (e.e.g, IP addresses configurations and information exchange intervals are needed for the PDR exchange between nodes). In the third line we load the reaction strategy (followed with its parameterization to inform the headquarter with a specific IP address) and assign it to our naïve jamming detection strategy.

In the following we give details about the detection and reaction strategy, beginning with the detection strategy.

Our **detection strategy** required the metrics noise and PDR. Noise was directly received from the Atheros wireless network interface card with ath9k driver. For the PDR we used the definition as given by Xu et. al. [XTZW05], that is, the number of packets received that pass cyclic redundancy check (CRC) divided by the number of all packets (or preambles) received. Figure 4.11 shows the graphical representation of our detection strategy.

From the total number of received packets we subtracted the packets from the point where we started to measure. Similarly we applied to packets having CRC errors. Afterwards, we subtracted both values from each other to obtain the number of packets passing the CRC errors. Finally, this value was divided by the number of received packets from the point starting to measure in order to obtain the PDR. The PDR is delivered to our message exchange application that broadcasts this value to other nodes. From all received PDR values the maximum is taken for detecting the presence of a jammer. Notice that discussions about drawbacks of this PDR calculation approach will be discussed later. We show the full configuration in Crawler's abstract description language in Listing 4.7.

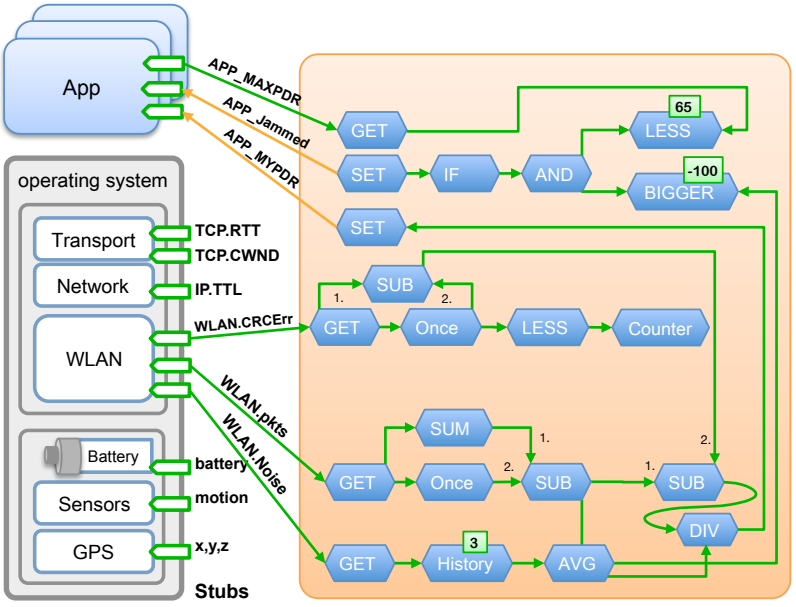


Figure 4.11 Graphical representation of our naïve jamming detection. For clarity reasons we simplified the graphical view. Arrows represent the information flow.

```
1 [init]
2 Timer:pollingtimer(25)
3
4 NOISE_AVG:avg(NOISE_HISTORY)
5 NOISE_HISTORY:history(NOISE,3)
6 NOISE:get(wlan0.cfg80211.survey)
7
8 MAX_PDR:get(app.consistencyDetection.maxPdr)
9 PDR:percentdiv(sub(RX_PKTS_INIT,RX_DROPPED_INIT),RX_PKTS_INIT)
10 RX_PKTS_INIT:sub(sum(RX_PKTS,1), DO_ONCE2)
11 DO_ONCE2:once(RX_PKTS)
12 RX_PKTS:get(wlan0.ath9k.rx.pkts_all)
13 RX_DROPPED_INIT:sub(RX_DROPPED, DO_ONCE1)
14 DO_ONCE1:once(RX_DROPPED)
15 RX_DROPPED:get(wlan0.ath9k.rx.crc_err)
16
17 TIMEOUT:less(COUNTER,1)
18 COUNTER:ringcounter()
19
20 JAMMED:set(app.consistencyDetection.jammed,EVALUATE)
21 EVALUATE:if(CONSISTENCY_CHECK,1,0)
22 CONSISTENCY_CHECK:and(CHECK_PDR,CHECK_NOISE)
23 CHECK_PDR:less(MAX_PDR,65)
24 CHECK_NOISE:bigger(NOISE_AVG,-100)
25 SET_PDR:set(app.consistencyDetection.pdr,PDR)
26
27 TIMER->NOISE_HISTORY;
28 TIMER->JAMMED;
29 TIMER->SET_PDR;
30 TIMER->COUNTER;
31 TIMEOUT->DO_ONCE1;
```

Listing 4.7 Configuration of the naïve jamming detection strategy.

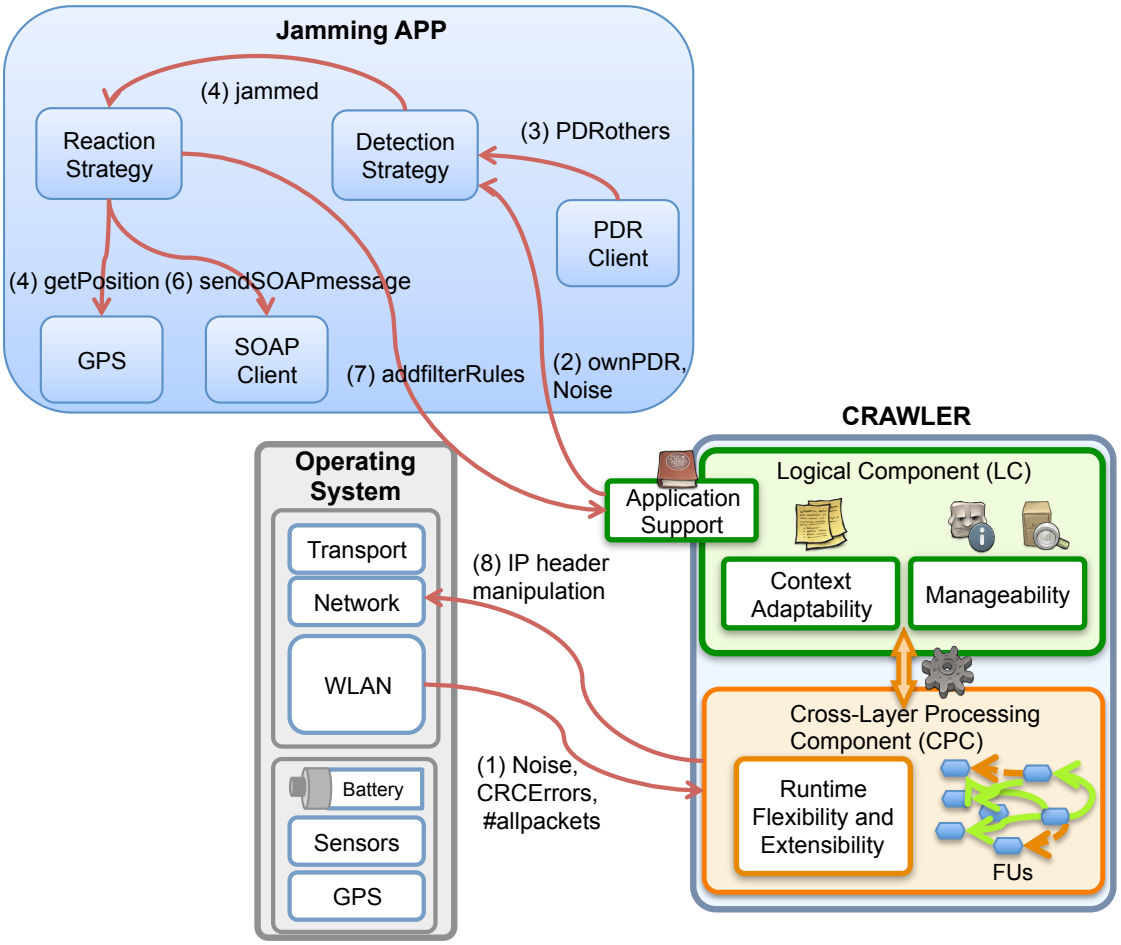


Figure 4.12 Abstract description of the notification message exchange process.

The thresholds available in the configuration were selected based on previous experimentations and observations. However, they are easily adjustable by changing the configuration. Moreover, due to CRAWLER's runtime flexibility and extensibility feature we enabled an interactive mode in the console to modify the thresholds at runtime.

The complete processing of our jamming detection and reaction scheme is shown in Figure 4.12. In a first step, the parameters noise, number of all received packets, and number of CRC errors is accessed by CRAWLER (cf. Step 1 in Figure 4.12). While noise is directly delivered to our jamming application, particularly to the detection strategy, the number of received packets and CRC errors are used to calculate the own PDR, i.e., used to exchange with neighboring nodes (cf. Step 2). In a next step the PDRs from all other nodes are collected by our PDR client (cf. Step 3). After detection of the jammer by our jamming detection strategy, the **reaction strategy** is notified (cf. Step 4). In a scope of a project, it was important to reliably inform the headquarter about the detection of a jammer and the current location in order to initiate further counter measurements. Therefore, it was also necessary to gather the current location from a GPS device (cf. Step 5). We included the location information into a specified XML exchange format using SOAP-messages (cf. Step 6). In addition, it was necessary to label the created SOAP-messages at the IP-layer since these specifically labeled packets are treated differently by the routing protocol. To achieve this, we injected a cross-layer coordination algorithm using CRAWLER's

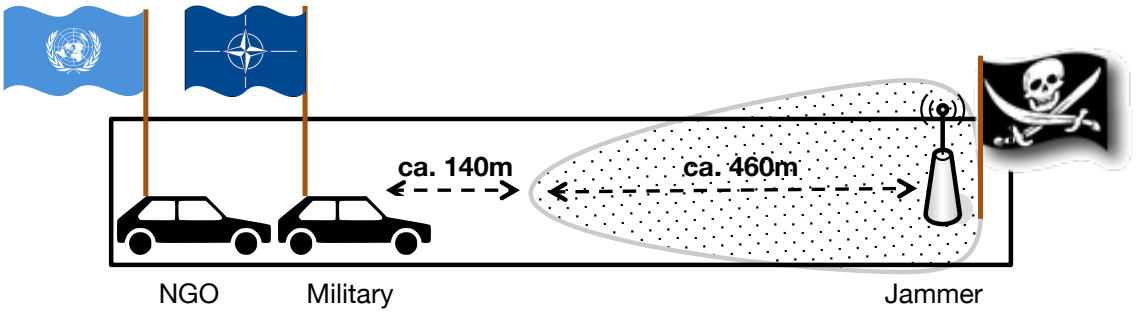


Figure 4.13 Mobile test scenario conducted in Greding, Bavaria, Germany. The military vehicle and the NGO were driving along a road, starting around 600m away from the constant jammer. After approximately 140m, the vehicles reached area affected by the jammer and further approached the jammer until they reached the it after another approximately 460m.

API (cf. Step 7) in order to filter packets belonging to our jamming application at the IP layer. In particular, we used the Linux kernel netfilter hooks to modify the TOS field of IPv6 (IPv4 is also supported) packets and fed them back to the network stack (cf. Step 8). All outgoing packets in the protocol stack are redirected to this hook and processed by our stub before transmission, but only those belonging to our application were modified. We used a satellite link to inform the headquarter. During our test, shown in the next section, we observed that these packets arrived successfully at the headquarter.

In the next section, we present results of our naïve jamming detection approach.

4.4.5 Validation

The evaluation of our detection strategy were conducted on a military testing ground in Greding, Bavaria, Germany. This testing environment was an open space rural area permitting wireless communication without major disturbances. The testing scenario was as follows. A military vehicle was instructed to escort a non governmental organization (NGO) vehicle in order to protect the NGO vehicle from enemies using jammers. Both vehicles were equipped with x86 computers running Vyatta-Linux, CRAWLER, and our application for jamming detection. The scenario is sketched in Figure 4.13.

At the beginning both vehicles were outside of the jamming affected area. The NGO and military vehicles were 20m away from each other and moved along a road at a constant speed of about 20km/h in direction to the constant jammer. The jammer used a directional antenna and an additional amplifier of 1W enabling the jammer to disrupt a wireless communication entirely, up to a distance of at about 460m. The jammer was hidden next to the road. Throughout the experiment, the vehicles exchanged ping messages and PDR exchange messages. The ping messages were secured by ssh connections and should mimic data traffic. Beside this, no further traffic was generated, neither by surrounding wireless nodes nor by other radio access technologies. After both vehicles reached the jamming-affected area, our detection strategy was able to detect the presence of the jammer as shown in Figure 4.14.

The curves of the PDR, Noise, and RSS graphs are very stable outside of the range of the jammer. Communication was not effected, i.e., all packets sent between

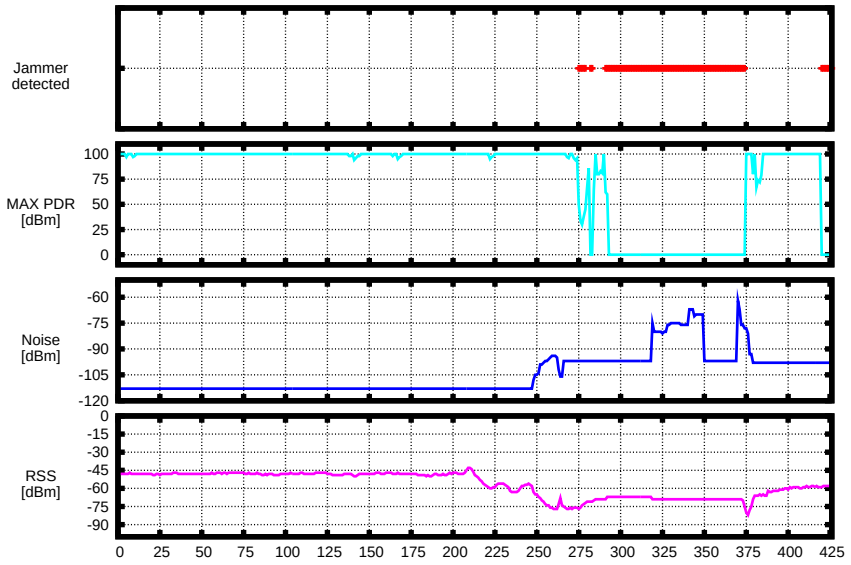


Figure 4.14 Results of the military vehicle in the mobile test scenario. Both vehicles were 20m apart and drove with nearly constant speed of 20kmh in direction to the constant jammer. The jammer was emitting a jamming signal of -13 dBm amplified by 1W.

the NGO and military vehicle could be received. With decreasing distance to the constant jammer, both vehicles entered the jamming-affected area. This resulted in an increase of the noise level (-62 dBm at maximum) and in a reduced RSS[7] (of -82 dBm at minimum). Between 325 and 375 seconds we observed a straight line for the RSS which is caused by the impact of the jammer which fully disturbed packet transmissions. In such a case, RSS measurements are not available[8] and the driver provides the latest available value. However, the collisions between packets, sent from the embedded computers inside the vehicle, and the jamming signal caused the maximum PDR to drop down to 0%. As a result, the jammer was detected and further countermeasures could be initiated.

In our next experiment, after reaching the jammer, the vehicles stayed in the range of the jammer for a longer period of time which we refer to as the pause scenario. Here we gained some new interesting insights as shown in Figure 4.15. After entering the jamming range, indicated by the red bar, the noise starts fluctuating although the vehicles did not move. This exhibits a periodic behavior which has also been observed in [PnAG12] using a constant jammer and an Atheros wireless network card. Puñal et al. assume that this is caused by ANI [BES05] which probes different noise immunity configurations (in discrete steps) due to false signal detections. Despite ANI, the embedded computers were not able to decode received packets correctly or did not detect packets indicated by the maximum PDR of 0% and the missing RSS reports. After a while, the vehicles start moving again, increasing the distance to the jammer. As a result, all monitored values start recovering.

[7] Note that the RSS also depends on the noise. Therefore, it is reduced in presence of the jammer.

[8] In case of corrupted packets, RSS measurements are discarded and after a certain amount of time the data structures for the associated node are set to zero what we will see in the next experimentation

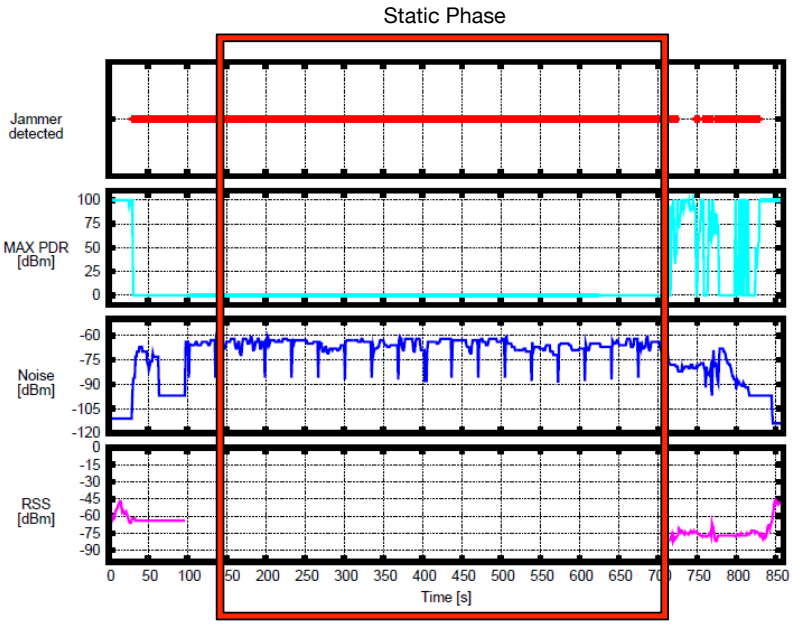


Figure 4.15 Results of the military vehicle in the pause scenario . After reaching the jammer, both vehicles stayed for a longer period of time close to the constant jammer. The jammer was emitting a jamming signal of -19 dBm amplified by 1W.

4.4.6 Summary and Discussion

To summarize, using CRAWLER we were able to successfully detect a jammer and react to it. CRAWLER enabled to monitor variables and their sensitivity to jamming. Based on our observations we correlated and manipulated protocol behavior at different layers. Moreover, we built a flexible and convenient framework for developers to add and remove detection and reaction strategies. The framework further allows to combine detection and reaction strategies in a way such that certain reaction strategies are triggered by predefined detection strategies upon detection of a jammer. Our evaluation demonstrated that we are able to successfully detect a jammer in a real-world mobile convoy scenario and initiate specific counter measures.

Although we have built a very flexible framework for jamming detection and reaction strategies, our strategy to detect a jammer is rather simple. In the following use case we suggest an advanced jamming detection scheme that utilizes further metrics and successfully detects different jammer types in challenging scenarios. Moreover, we discuss that manual setting of thresholds is a very complex problem and suggest a solution to handle that problem space.

4.5 Use Case: Machine Learning-based Jamming Detection

In this use case[9] we provide an advanced solution for jamming detection. The previous simple jamming detection strategy was implemented as a cross-layer coordination

[9]The content of this use case is partially based on the joint work with Oscar Puñal, Caj-Julian Schnelke, Gloria Abidin, Klaus Wehrle and James Gross published in "Machine Learning-based Jamming Detection for IEEE 802.11: Design and Experimental Evaluation", 15th International IEEE Symposium

algorithm which was injected into the kernel space of the OS using Crawler. In this use case Crawler only serves as a metric provider. In particular, six metrics are provided via Crawler to a machine learning application that decides about the presence of a jammer. Thus, we moved the jamming detection decision from the kernel space to the user-space while offering the same features of our jamming framework which also highlights Crawler's flexibility.

4.5.1 Motivation

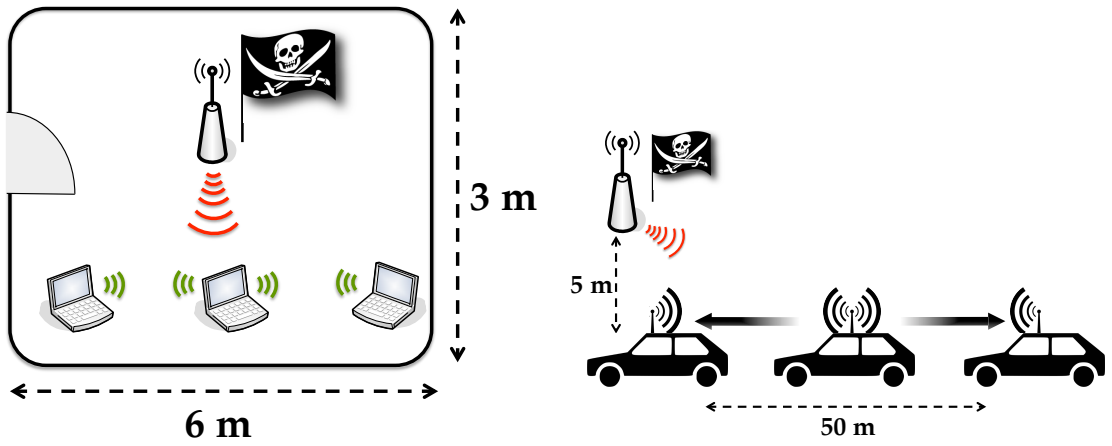
Reliable detection of a jammer is a crucial task since based on it critical countermeasures could be initiated. Even if the detection of a jammer is a simple binary output, i.e., being jammed or not, an accurate jamming detection approach needs to differentiate equally good between both being jammed (true positive) and not jammed (true negative) respectively. In other words, an efficient detection approach needs to separate jamming attacks from "normal" operation such as network congestion, bad channel conditions, and packet collisions to keep false positives and false negatives low.

However, implementing such a jamming detection approach typically requires many efforts from a developer. In a first step, a developer has to monitor potential variables in a system with and without the presence of a specific jammer. Based on her observations, variables that significantly react to jamming are identified as relevant metrics. In a next step, the metric combinations and their thresholds have to be determined. These two steps suffer under the following problems: First, the time demanding selection of parameters as metrics, the subsequent metric combinations and their thresholds are highly coupled with the developers capacity to understand interrelations. Second, "normal" operation conditions such as network congestion, bad channel conditions, and competing communication further increases complexity and accordingly interrelations. Differently put, while the observation of a single parameter is a relative easy task for a human, the holistic understanding for all constellations of parameters under varying conditions is a difficult task.

In this section we present a machine learning based jamming detection approach for 802.11 networks which automatically determines the impact of metrics from a given set of parameters and relieves the developer from the burden of finding the right set of thresholds independent from the complexity of a scenario. Our approach accesses parameters from the system by utilizing Crawler which alleviate the problem of complicated access. The accessed parameters are gathered from unmodified firmware of commodity hardware. Afterwards, parameters are provided to our machine learning approach using Random Forests [Bre01] to decide about the presence of a jammer. We show the high accuracy of our approach in several scenarios (static indoor and mobile outdoor), with different topologies (good and challenging bad link conditions) and two jammer types (constant and reactive).

In the following, we present our test setup and analyze the impact of different jammers on selected metrics and discuss their suitability for jamming detection.

on a World of Wireless, Mobile, and Multimedia Networks, 2014 (WoWMoM'14) [APS+14a] and in the master theses of Caj-Julian Schnelke [Sch13] and Gloria Abidin [Abi13].



(a) Indoor Scenario (b) Outdoor Scenario

Figure 4.16 Sketch of the indoor and outdoor scenarios considered throughout the evaluation.

4.5.2 Setup and Monitoring

Our reference scenario is conducted in a small office room located in the UMIC research centre at RWTH Aachen University. A sketch of this reference scenario is shown in Figure 4.16(a). We used three Linux PCs equipped with an 802.11g/n Atheros WLAN card running the ath9k driver [ath]. We let the three PCs build an ad-hoc network and continuously exchange messages on channel 11 within the 2.4 GHz band[10]. The distances among the nodes is provided by the sketch.

We wanted to imitate two diverse link conditions which we refer to as the **good link** and **bad link** case. In the good link case, which we consider as an ideal case, the PCs are placed close to each other and the transmission is parameterized such that, on average, a high packet delivery rate is achieved. In the bad link case, which can be considered as the challenging case, the communication can be characterized by a poor communication performance. This is mainly achieved by selecting a lower transmit power and/or by adding attenuation elements at the output of the radio front-end. For each of the cases, we collect data under **normal** and **jammed** conditions. In addition, to vary the impact of the jammer, we place it at different positions and vary its output power to impact the performance of the communicating nodes differently.

We implement the jammer on the Wireless Open-Access Research Platform (WARP) board [KCH+08], which provides an 802.11-like OFDM physical layer featuring a 10 MHz bandwidth and an output power of 18 dBm in the 2.4 GHz band. The jammer signal consists of a preamble and BPSK modulated random payload of variable length. We used the following two configurations:

Constant jammer: The amount of time that the boards can continuously transmit a single signal is upper-bounded for the WARP boards. The transmission consist of successive on and off phases. We measured the on-phase using a spectrum analyzer to be about 2.7 ms. Between two consecutive signals or on-phases there is a 10 μs gap or off-phase, which is required by the hardware in order to set up a new transmission.

[10] We opted for channel 11 since during our test it was not occupied by any other network. Nonetheless, the sporadic impact of neighbor 802.11 communications is not guaranteed.

Reactive jammer: After sensing energy on the channel above a predefined threshold, the reactive jammer starts a transmission. During our test, we set the threshold to -65 dBm to achieve a sufficiently high jammer sensitivity, while guaranteeing a low number of false detections, that is, avoid reacting to signals from neighbor 802.11 networks or other sources of electromagnetic activity. The jammer has a total reaction delay of 12 μs. This is fast enough to partially interfere the preamble of the 802.11 signal, which is known to increase the effectiveness of the attack [GWGS07].

4.5.2.1 Sensitivity of Metrics to Jamming

To differentiate jamming from normal operation, the impact of jamming while being enabled and disabled was analyzed under different conditions. To monitor the impact, we observed a set of metrics. Compared to the naïve jamming detection (see Section 4.4) we discovered further metrics. In particular, we investigate a set of *metrics* that significantly react to jamming attacks and *helper metrics* that do not show a reaction to jamming, but provide further context for an appropriate weighting of other metrics. Our set of metrics was selected based on two main criteria: (i) we only focus on metrics that are accessible via a common driver of commodity 802.11 network interface cards, (ii) the metrics should behave independent of the type of traffic exchanged between nodes. For example, we did not use the number of *frame retransmissions* although available on common drivers, since this metric requires the use of acknowledged frames when using uni-cast traffic that would not be available when sending broadcast messages. In the following we discuss and present the behavior of our six selected metrics in the presence and absence of jamming. We grouped our metrics due to their functional behavior into three categories: (i) channel, (ii) performance, and (iii) signal metrics.

Channel metrics: Channel metrics exclusively sample the state of the wireless channel and are, hence, measured independently from packet receptions. We identified noise and channel busy ratio (CBR) as relevant metrics. Noise is defined as the power measured on the channel (in dBm) during idle times of the transceiver [Noi]. Jamming signals that are transmitted while the legitimate nodes are idle (e.g., constant jammer) are likely to be included in the noise measurements of the cards as shown in Figure 4.17(a). We see that the constant jammer shows a different behavior compared to the no jammer and reactive jammer case. This behavior is similar with the CBR metric (see Figure 4.17(b)), which measures the time (normalized to the observation time) that the wireless channel was sensed as busy. The channel is considered being busy if the received power is measured to be above the clear channel assessment threshold. However, in the reactive jammer case only little impact is visible, since the attack is launched once the nodes have already gained medium access as shown in Figure 4.17(b).

Performance metrics: This type of metrics can only be obtained if a connection is established between two or more stations. We identify *inactive time (IT)* and *packet delivery ratio (PDR)* as suitable metrics. The IT corresponds to the time that elapses between two consecutive successful packet receptions, including probing, beacons, and payload frames. Specifically, we account for the maximum IT at a node measured over the links to its neighbors. To the best of our knowledge, we are the first to propose this metric for jamming detection. Figure 4.17(c) shows the very

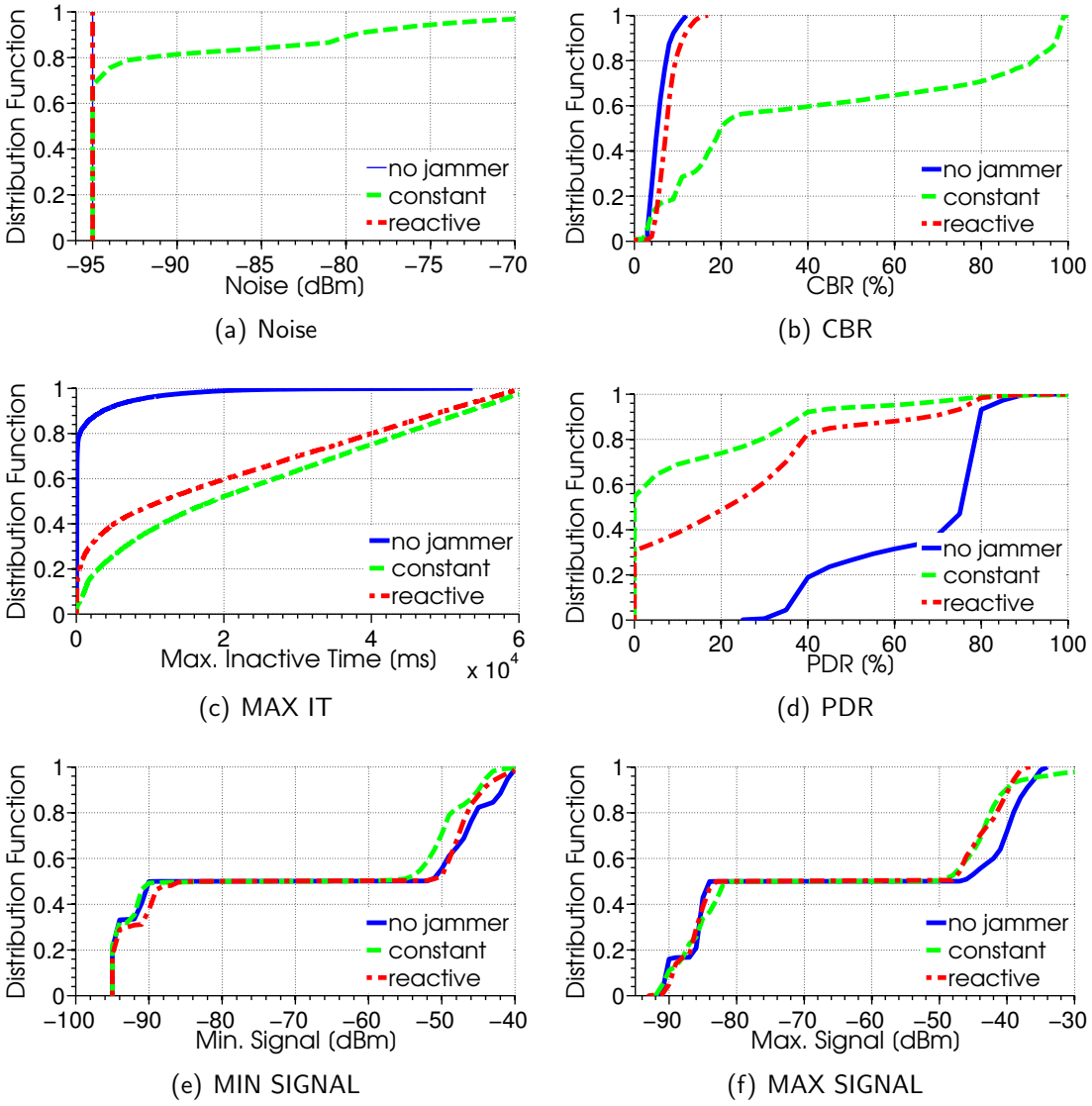
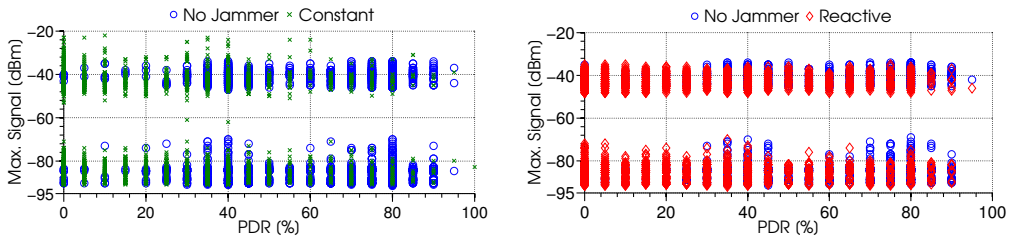


Figure 4.17 CDFs illustrating the impact of the **constant**, **reactive jammer** and the **non-jammed** case on selected metrics. Noise is unaffected by the reactive jammer, in contrast to the constant jammer. The CBR is strongly affected by the constant jammer, while the reactive jammer has only a marginal impact on it. PDR and max. IT are significantly affected by the presence of both jammers.



(a) Constant (b) Reactive

Figure 4.18 Application of the consistency check approach on the PDR vs. max. signal metrics as proposed by Xu et al. in [XTZW05] on the data gathered in the reference scenario. Samples for jammed and not jammed overlap, which makes a clear threshold identification and jamming detection impractical in 802.11 networks.

diverse behavior in the presence of both jammers compared to the no jammer case. As opposed to the other metrics, the PDR is not directly provided by the card. For its computation, each node is aware of the number of network members in its hearing range and of a predefined rate for generating probing packets. With that knowledge, and based on the number of correctly received probing packets, the PDR can be computed. Figures 4.17(d) shows that PDR also diverges for both jammer types in comparison to the no jammer case. The clear difference in behavior for the PDR and max IT from normal operation indicate that both metrics are good candidates for detecting the presence of jamming.

Signal metrics: Signal is the power measured upon arrival of a packet, but only passed to higher layers in case of successful reception. The received power (in dBm) is measured over the preamble of the packet only. We consider this metric as a *helper* metric, since it does not show an evident difference in behavior for any of the tested jammers, however provides a useful context (i.e., link quality) to the PDR and the max. IT metrics. For example, a low received signal power correlates with a low PDR even if the jammer is silent. This knowledge is important to appropriately weigh the significance of PDR and max. IT accordingly. In our experimentation, instead of collecting a single signal metric (e.g., the average power of the received packets), we have observed that the differentiation between *minimum* and *maximum* signal is most valuable.

4.5.2.2 Threshold Identification Problem

After finding metrics that react on jamming, the next step is to manually explore the right thresholds in order to decide the presence of jamming. For this reason, the behavior of metrics and their correlations have to be analyzed for a certain scenario. This is a difficult task, since correlations are not simply derivable which we illustrate with the following example. We applied Xu's approach [XTZW05] of a consistency check method where appropriate metrics are used in conjunction to exploit their correlation. In a first step, we jointly collected samples of PDR, signal strength and noise power values. We collected these samples in our reference scenario under normal operation, i.e., without jamming activity, following the description as proposed in [XTZW05]. These samples correspond to the blue circles as depicted in Figure 4.18. Subsequently, from all measurements the minimum and maximum

measured values are taken as thresholds. Specifically, the thresholds are set such that 99% of the unjammed samples are covered according to Xu et al. In a next step, we activated the jammer and evaluated how well the thresholds identified a jammer. Xu et. al. observed in their experiments that the measured values are clearly separable from normal operation, i.e., below and above thresholds. This worked fine for their testing scenario and metrics (signal and PDR) gained from 802.15.4 devices. However, for 802.11 in our test scenario we observed a different behavior. Figures 4.18(a) and 4.18(b) shows a clear overlap of jammed and unjammed samples in the presence of a constant and a reactive jammer respectively. These overlaps are an evident hint for inaccurate detection rates.

To conclude, from these figures we derive two major observations: (1) metrics for jamming detection proposed by related work in the context of general wireless networks do not necessarily work well in 802.11 and (2) finding appropriate thresholds even for only two metrics is already a difficult task. Moreover, we believe that the combination of multiple metrics will drastically increase the difficulty and make manual threshold setting impractical. This complexity will likely increase in scenarios where normal operation is harder to separate from jamming such as for concurrent traffic and hence demanding an mechanism that releases from threshold finding. Fortunately, machine learning is a well-suited approach for such multi-dimensional binary (i.e., being jammed or not) classification problems.

4.5.3 Cross-Layer Coordination Approach

We increased the amount of metrics in comparison to the previous jamming detection approach. Additionally, we used a reactive jammer which is known for being harder to detect than other jammer types [GLS+13] such as the pilot jammer which we used in the naïve jamming detection use case. However, by providing our selected set of metrics from diverse layers, i.e., from the NIC and application layer, to a machine learning application, we aim at (1) tackling the problem of manual threshold identification and (2) improving the jamming detection accuracy in general.

4.5.4 Realization with CRAWLER

Our jamming detection approach consists of two phases: (1) the collection of training data and (2) the application of machine learning on the collected data. The overview of this interplay is depicted in Figure 4.19.

In the **data collection phase** input for the machine learning algorithm have to provided in form of training data. Therefore, our selected metrics need to be accessed and forwarded to the machine learning component. Relevant information such as protocol and system information typically resides in the kernel-space of the operating system (OS). This OS restricted information nonetheless has to be provided to the user space in order to make it accessible. For convenient experimentation reasons, i.e., to flexibly include and evaluate the impact of diverse metrics with less effort, we used CRAWLER (cf. (1) in Fig. 4.19). With CRAWLER we collected the values of our metrics and created an output at the user space of the OS that is usable for the machine learning algorithm. Note that CRAWLER already supports the inclusion and

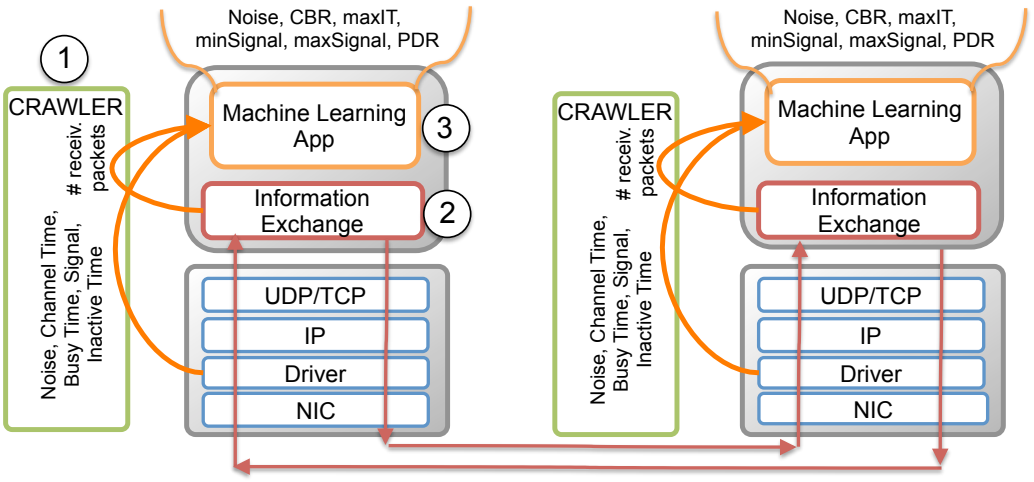


Figure 4.19 Design overview of our detection approach consisting of following three components: (1) Metrics are accessed via CRAWLER and are provided to the machine learning component. (2) Similarly, the probing client provides the PDR metric. (3) Based on the gathered data, the machine learning component decides on the presence of the jammer.

logging of many specific metrics as a wide range of accessors to read and write system information ranging from TCP-IP to our metrics gathered from the WLAN device have already be implemented. Remark, since the complete CRAWLER configuration requires too many rules for monitoring and computation of the diverse metrics, we skip it here but provide it for completeness reasons in Appendix C.

However, the jamming detection can be based on passively monitoring ongoing transmissions, but it can also be enabled to a cooperative mode for collecting feedback information from other nodes in the network. To achieve this, we implemented a information exchange component (see (2) in Fig. 4.19). When the information exchange component is configured to use the passive mode, nodes generate probing traffic to enable the computation of specific metrics (e.g., PDR), while in the cooperative mode the probing packets are further used to exchange information (i.e., the detection probabilities) for cooperative detection. We have implemented the packet exchange in a client-server manner running in the user space of the OS to measure the PDR as described in Section 4.5.2.1. Each node runs the server and the client. The client broadcasts UDP packets every 100 ms. These packets have a total size of 57 Byte (8 bit message type, 16 bit value for cooperative use, and 54 Byte for protocol headers and CRC checksums). Hence, the broadcast of probing packets introduces a per-station overhead of about 570 Byte/s. The overhead costs are not a must, but as we later show in our evaluation, the use of PDR as a metric and cooperation among neighboring nodes (by using the reserved 16 bit field for exchanging prediction probabilities) increases the jamming detection accuracy.

After all metrics are gathered, they are utilized in the **machine leaning phase** by a selected machine learning algorithm, namely Random Forests [Bre01], for learning and later predicting the existence of jamming activity (see (3) in Fig. 4.19). Random Forests is a learning based heuristic that exploits statistical dependencies in multi-dimensional decision problems and is known to be superior to most other (supervised) machine learning methods [Bre01]. Remarkable features are the fast convergence of the algorithm, the capability to face situations different than the

ones observed during training, and the ability to work with missing input variables. Since the performance of other machine learning approaches and the details about selected parameterization of random forest is out of the scope of this thesis, we refer to [Abi13] for more details.

On the training samples reserved for testing, we apply the machine learning algorithm and obtain a prediction probability of a jamming attack. Note that we have built our framework in way that we are also able to perform an online prediction by using available training data. In the following, we show the results that we have achieved with our machine learning-based jamming detection framework.

4.5.5 Validation

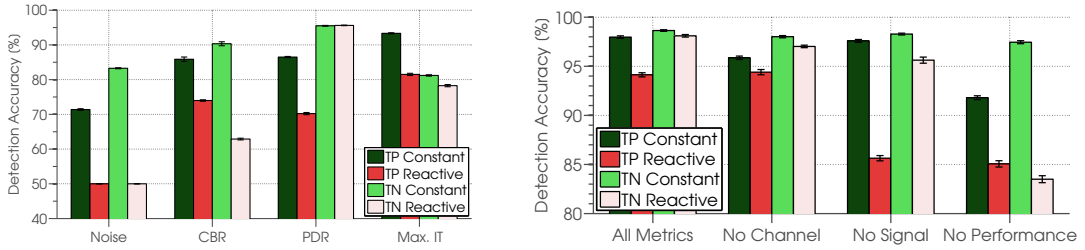
On starting our jamming framework, each node broadcasts beacon frames which are used to form an ad-hoc network. We used the test setup as described above (see setup for reference scenario in Section 4.5.2). Every second the nodes collect (and store for learning) the values of the metrics. For each link condition (i.e., good and bad link) we conducted multiple measurement runs with a duration of 60 s each. In order to avoid biased learning, we collected the same number of instances for both with and without jamming activity. We collected a total of 27000 samples, namely 9000 for each jammer and 9000 without jammer. Note that the number of instances is also evenly distributed among the different link qualities (i.e., good and bad link quality). From all gathered data, we randomly selected 60% for learning and 40% for testing. In particular, we conducted this selection 20 times to collect different subsets of training data in order to avoid biased learning. On the samples reserved for testing we attained the prediction accuracy. In this conjunction, we differentiate between the **true positive (TP)** rate, i.e., the correct detection of existing jammer activity, and the **true negative (TN)** rate, i.e., the correct identification that there is no jammer[11]. Please not that unless specified differently, our results show the average detection accuracy together with the 95% confidence intervals. For more details about Random Forest configuration and setup, we refer to [Sch13].

4.5.5.1 Indoor Detection Accuracy

From our observations four metrics emerged as suitable indicators for jamming activity. We first wanted to analyze if a subset of these metrics (or even a single metric) is able to yield a reasonable detection accuracy before using all of them together. In Figure 4.20(a) we show the detection accuracy achieved in the reference scenario when using only a single metric for learning. We can see that every metric has the potential of detecting jamming activity since all TP rates are larger than 50% with the single exception of the noise metric in case of the reactive jammer. This fact underlines our previous observations derived in Section 4.5.2. These results clearly show that relying on a single metric is not sufficient for reliable jamming detection.

In a next step we excluded a certain metric group (signal. channel or performance) from the whole set of metrics to further analyze the impact of certain metrics. Figure 4.20(b) shows the obtained detection accuracy with all metrics available in

[11] Alternatively, false positives or negatives can be achieved as 100% minus TN or TP, respectively.



(a) Prediction accuracy achieved with a single metric.

(b) Prediction accuracy obtained with all metrics in comparison to when a certain group of metrics is excluded from learning.

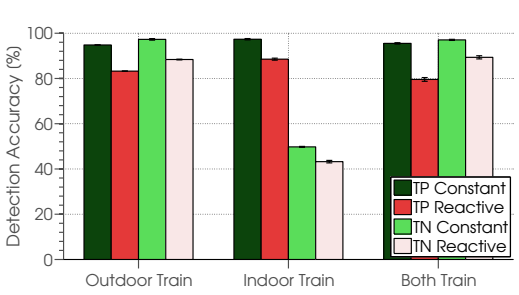
Figure 4.20 Reference scenario detection accuracies. Results emphasize using all metrics together provides best accuracy for jamming detection.

comparison to when certain metrics or a metric group is missing. As shown, using all metrics together results in higher detection rates. In particular, the *no channel* group (i.e., excluding noise and CBR) yields a high accuracy, although the detection of constant jammer decreases marginally. When excluding the *signal* metrics, the detection accuracy of a reactive jammer decreases significantly which highlights the relevance of these helper metrics. When we exclude the performance metrics (PDR and max. IT), this leads to significant accuracy drop for all detection rates highlighting the importance of this metric group. This is similar to our observations as presented in Section 4.5.2. To conclude, although a single metric can be utilized to some extent for jamming detection, a holistic consideration of all metrics provides the best performance.

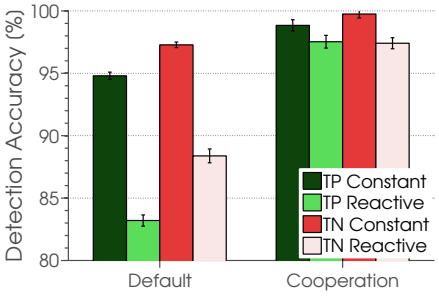
4.5.5.2 Impact of Outdoor Mobility

The aim of the following scenario is to evaluate the impact outdoor mobility. We conduct a test using three cars in a parking lot within RWTH Aachen University campus. We place two cars at the ends of a parking lot which are static during all test runs. The third car is mobile and moves back and forth between the static nodes reaching a maximum speed of 25 km/h. The jammer is placed close to one of the static nodes, as shown in Figure 4.16(b). With this scenario we mimic a convoy scenario that passes a jammer. The wireless link between the two static nodes is characterized (without jammer activity) by a low PDR of about 40% that may be even further reduced during the test, e.g., due to the wireless effects such as shadowing caused by the moving vehicle. Based on the position of the moving car, the quality of the links varies significantly and we attain PDR values that span the whole range (from 0 to 100%). We conduct multiple runs of 60 s each and collect a total of 4500 samples (1500 for each jammer and 1500 without jammer).

Figure 4.21(a) shows the detection accuracy of our machine learning-based approach in the outdoor scenario (cf. Outdoor Train case). For both jammer types we achieved high prediction accuracies, while the reactive jammer being the lowest (above 82%). The reason for having lower prediction accuracies in the outdoor mobility are that the nodes are differently affected by the jammer. We have observed that the resulting accuracy varies significantly from one node to the other. The high prediction



(a) Prediction accuracy outdoor compared to training indoor data but outdoor prediction, and mixing both training data for outdoor prediction.



(b) Comparison of prediction accuracy without cooperation (default) and with cooperation (coop) among neighboring nodes.

Figure 4.21 Outdoor scenario detection accuracies.

accuracy that we have achieved in the mobile scenario indicate that mobility is not a limiting factor for our approach.

Obtaining training data in outdoor environments requires much effort. Moreover, it is also very time-consuming (test scenario needs to be continuously monitored) and costly (hardware needs to be acquired dependent on the scenario complexity). Therefore, we investigate the reusability of indoor training for predicting the presence of a jammer in the outdoor scenario as also shown in Figure 4.21(a) as the "Indoor Train" case. As shown, performing learning without considering wireless outdoor conditions such as different wireless propagation effects and mobility leads to a significant drop in accuracy. The TN rates for both jammers suffer a dramatic loss (fall below 50%). From this observations we derive that it is important to collect training data samples under different conditions that are likely to emerge during the scenario being tested. However, in Figure 4.21(a) we show that by combining training data samples from both scenarios (i.e., from indoor and outdoor), high TN and TP rates are obtained, which are comparable with the accuracy achieved with specific in-situ learning. Thus, when collecting and combining training samples under different conditions high prediction accuracies are achieved with our machine learning-based jamming detection approach.

4.5.5.3 Cooperation Between Nodes

So far, we have only shown results for the passive mode of our design, i.e., a node only monitors and decides for its own about the presence of a jammer without exchanging information with other nodes. We also support an active mode, where we cooperatively exchange information with other nodes in the network. In this section we show our evaluation results for cooperative jamming detection, i.e., where neighboring nodes exchange information about their individually computed detection rates. This information can easily be delivered within the probing packets without adding complexity or overhead[12]. The utilization of the information from neighbors is very simple in our approach. Every node decides on the presence of a jammer

[12]Note that detection probabilities of the neighbors are available once they are computed. Hence, since we detect on a one second basis, there is a mandatory delay of one second that we consider in our evaluation.

based on the *average* detection probability across all nodes. Based on the scenario, the importance of this information might vary. For example, the accuracy of a node that is being affected by a jammer can decrease when the information of nodes that are not affected by the jammer (and, hence, forward a low probability for the presence of jamming) is used. An improvement could be the use of GPS information to appropriately weigh the probability of the neighbors. Note that such investigation is out of the scope of this thesis.

Figure 4.21(b) shows the prediction improvements that we achieved in the outdoor scenario when using the active mode of our framework, i,e., using a cooperative jamming detection. The prediction rates for both jammers increase significantly, especially for the reactive jammer where we achieved an improvement of around 14% and 9% for the TP and TN rates, respectively.

In the following, we discuss related work for our jamming detection approaches.

4.5.6 Related Work

Several jamming detection approaches have been proposed in the past few years for wireless networks [XTZW05, GLS+13, TMS11, TW08, HBOM09]. However, the majority of these works only evaluate the proposed approaches using simulations [TMS11, TW08] or theoretically [HBOM09]. In contrast, the approach proposed by Xu et al. [XTZW05] and Giustiniano et al. [GLS+13] have been implemented for 802.15.4 and 802.11 respectively. Both of these approaches are focused on a certain jamming detection strategy for a specific scenario. Non of them offer for developers a dynamically flexible framework which allows to add and remove jamming detection and reaction strategies.

Regarding the jamming detection strategy, Xu et al. use a consistency check method requiring energy measurements together with the packet delivery ratio for jamming detection. The authors showed that different jammer types (constant, reactive and deceptive) can be identified. However, their approach is not directly applicable to 802.11 systems since it suffers under following weaknesses when directly transferred. Energy measurements are not feasible due to the packet coupled nature of 802.11 systems. Furthermore, their PDR calculation scheme which will be influenced by the rate adaptation of 802.11 networks making both of the metrics less useful for 802.11 systems.

Giustiniano et al. [GLS+13] suggested an approach for detecting only a reactive jammer and only in direct sequence spread spectrum (DSSS) wireless systems (e.g., 802.11b/g). The authors characterize the relationship between the chip error rate measured over the preamble and the actual frame error rate under normal operation. In cases where the transmission events diverge from the previously characterized behavior, the author assume the presence of a jammer. They implemented and evaluated their approach on a USRP platform featuring only a DSSS physical layer. Hence, compared to our approach it does not support all current and future 802.11(a,n,ac) systems which uses a OFDM PHY. Furthermore, the proposed metric is not attainable by commodity 802.11 devices.

4.5.7 Summary and Discussion

We have presented a machine learning-based jamming detection approach for 802.11 systems which reliefs developers from the effort and complexity of finding the right thresholds for their metrics. In total we investigated the use of 6 metrics including two novel metrics which we gather from commodity off-the-shelf hardware. We conducted static indoor and mobile outdoor real-world experiments where we achieved notably high detection accuracy both for true positives and negatives under different propagation conditions, and for constant and reactive jammer types. Although our approach is mainly designed in a way that it does not rely on other applications or information from other nodes in the network, we have enabled a simple cooperative approach that can be enabled on demand to exchange information with neighboring nodes. Our experiments show that cooperation significantly improves the detection accuracy without rising higher transmission overhead. We have implemented our machine learning-based jamming detection to an extent where we are able to perform online detection by using available training data from previous test runs.

In contrast to previous use cases, CRAWLER only served as a metric provider. Thus, CRAWLER simplified the access to parameters, prepared them as useful metrics for jamming detection and forwarded them to another application for jamming detection.

4.6 Conclusion

In this chapter we have presented different use cases from different fields of networking. The use cases demonstrate the use of CRAWLER as a monitoring and experimentation tool. In this context, we particularly highlighted the convenience of CRAWLER to monitor the system, obtain specific cross-layer coordination ideas and to rapidly realize these ideas. In particular, the use cases highlight the following features of CRAWLER.

The first use case, mainly focused on how to use CRAWLER for simple access, i.e., to read and manipulate parameters residing at different layers and components in the system. Based on observed RSSI values at the MAC layer, we manipulated the behavior of the transport layer protocol TCP (i.e., send congestion window) and the application Skype.

The second use case, aimed to highlight CRAWLER's runtime flexibility and extensibility feature. We switched between two different TCP congestion control algorithms based on wireless link conditions. Moreover, we highlighted the application support and context adaptability feature, by feeding the cross-layer coordination algorithms on demand into the system.

With the third use case we focused on improving application behavior or rather user perceived quality. We proposed an automatic codec switching scheme that based on the observed network parameters selected the most adequate audio codec. CRAWLER was used to provide the necessary network parameters. In this conjunction, it was necessary to use two different applications and to establish a cooperation between them which we achieved by using CRAWLER's application support feature.

In the fourth use case, using CRAWLER's application support and runtime flexibility and extensibility features, we proposed a very flexible framework for jamming detection and reaction strategies. This framework allows to conveniently add and remove jamming detection strategies at runtime. Moreover, upon jamming detection predefined set of jamming detection strategies could be initiated. In the scope of a project, we have implemented a jamming detection and reaction strategy. Both the jamming detection and reaction strategies are realized as cross-layer coordination algorithms.

In the fifth use case, we proposed an advanced solution for jamming detection. In this use case we shift the jamming detection from the kernel space to the user space of the OS which highlights also CRAWLERs flexibility to realize and move the logic of cross-layer coordination algorithms to different places of the system. In this use case, CRAWLER mainly served as a metric provider for an application that uses machine learning to decide about the presence of jamming.

5

Coping with Multiple Cross-Layer Coordination Algorithms

In previous chapters, we introduced the cross-layer design paradigm, proposed the cross-layer architecture Crawler, and by using Crawler presented several cross-layer coordination algorithms and their benefits. Especially, we have emphasized how cumbersome the process of designing and realizing of even a single specific cross-layer coordination is. This is mainly because the protocol stack and drivers controlling system components are deeply integrated into the operating system which provides only a few limited interfaces. Moreover development support is missing for adding, modifying and removing a single specific cross-layer coordination at runtime when a certain context is available.

However, while the previous chapters primarily focused on designing single specific cross-layer coordination algorithms, this chapter focuses on the even more difficult step of designing and experimenting with multiple cross-layer coordination algorithms. In particular, we present a further major contribution of this thesis, namely by answering our third research question, i.e., how to handle problems caused by multiple cross-layer coordination algorithms?

The remainder of this chapter is structured as follows. Section 5.1 motivates the problems that a developer faces when running multiple cross-layer coordination algorithms. In Section 5.2 we elaborate these problems and present two distinct approaches to tackle them. First, in Section 5.3 we classify the problem of contradicting cross-layer coordination algorithms and dependent on the problem present different approaches to support the developer. Second, in Section 5.4 we present the extension of Crawler to automatically detect redundancies of multiple cross-layer coordination algorithms without developer interaction. Finally, Section 5.5 concludes this chapter.

5.1 Motivation

Designing a single specific cross-layer coordination is already tedious as emphasized in previous chapters. We demonstrated how Crawler helps to simplify this process. But the process of designing gets significantly worse for multiple cross-layer coordination algorithms. In particular, while a single specific cross-layer coordination is developed with the goal to improve the system performance, running several cross-layer coordination algorithms can lead to unintended problems. Let us consider two exemplary coordination algorithms. The first cross-layer coordination provides fine-grained localization information to an application by employing both Wi-Fi and GPS related information. In contrast, the second cross-layer coordination provides coarse-grained localization information by turning off GPS related hardware to save energy. While both coordination algorithms are designed to improve the system behavior when running exclusively, they may have redundant processing (e.g., parts of the localization might be similar) or contradicting effects (e.g., accurateness vs. energy) when running in parallel. Moreover, in this example, the latter case even negates some processes performed by the former. Accordingly, developers would highly benefit by having a framework that helps to detect, analyze and resolve such circumstances.

5.2 Problem Analysis

From the motivation presented in the previous section we derive two major problems that occur when running multiple cross-layer coordination algorithms in parallel.

Lack of developer support to cope with contradicting cross-layer coordination algorithms
The ability to add and modify cross-layer coordination algorithms without the knowledge about existing cross-layer coordination algorithms could also lead to **conflicts**, i.e., unintended interdependencies between cross-layer coordination algorithms leading to peculiar system behavior [KKTC05]. In some cases, even if the developer knows the set of running cross-layer coordination algorithms, the unintended interdependencies between multiple coordination algorithms could become very complex and the source of the conflict hard to detect. In some cases the only indicator is the performance drop or oscillation of a certain metric such as throughput. But this could also occur due to other reasons such as poor wireless conditions and thus increasing the difficulty to draw conclusions about the real root cause of the performance drop. Therefore, a tool to experiment with a set of cross-layer coordination algorithms and the ability to monitor states of cross-layer coordination algorithms and system variables to analyze to source of conflicts will be beneficial for developers.

Lack of automatic redundancy removal of multiple cross layer coordination algorithms
We believe that application developers know best about their applications and the need of cross-layer coordination algorithms to improve them. But giving application developers the freedom to add or modify running sets of

cross-layer coordination algorithms in the system without even knowing what is already running in the system might lead to **redundant** parts of cross-layer coordination algorithms. For instance, if several applications implement the same access to localization or protocol information. If each application provides its own algorithm, we run multiple instances of the same algorithm wasting CPU and memory over and over again. Accordingly, a mechanism that is able to automatically detect and resolve these redundancies is beneficial.

In this chapter we discuss both problems in detail and present solutions how to tackle the problems beginning with cross-layer conflict detection.

5.3 Cross-Layer Conflict Detection

The overall goal of a cross-layer coordination is to improve a performance metric such as energy, throughput, delay or user perceived quality of service. While we observed in previous sections that running a single specific coordination algorithm leads to respective performance improvements, multiple cross-layer coordination algorithms in parallel could lead to unintended contradicting effects resulting in severe performance degradation. In the cross-layer design domain this problem is referred to as *cross-layer conflicts*.

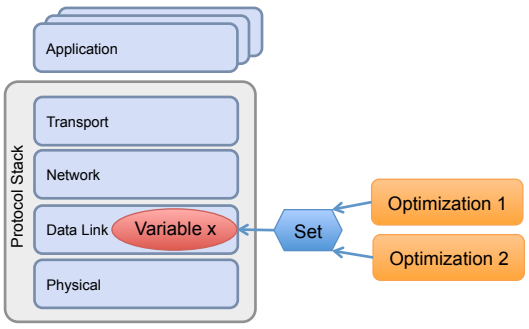
Although cross-layer conflicts are a well-known problem [KKTC05, SM05, Wil08], the existing cross-layer architectures fail to assist the developers in detecting such conflicts and in finding the right set of coordination algorithms. In this section, we classify conflicts and present an architectural extension to Crawler that supports developers in experimenting and detecting cross-layer conflicts. This architectural extension provides feedback to developers on conflict detection. Thus, Crawler helps developers by informing about a problem, but the resolution of conflicts nonetheless needs to be manually performed by the developer. Fortunately, by using Crawler this step is supported as developers can easily modify cross-layer coordination algorithms and experiment if changing sets of cross-layer coordination algorithms still lead to conflicts. In the following we first classify different types of cross-layer conflicts before presenting our approach to cope with them.

5.3.1 Classification of Cross-Layer Conflicts

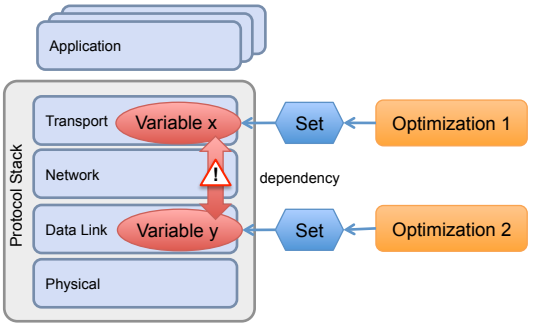
We classify[1] cross-layer conflicts into (1) **direct conflicts** and (2) **indirect conflicts**.

Direct conflicts occur when multiple cross-layer coordination algorithms try to manipulate the same variable in a certain protocol as shown in Figure 5.1(a). Here, multiple coordination algorithms try to manipulate a single variable at a certain layer via a set-FU. Hence, it is possible that two conflicting coordination algorithms have contradicting effects on the variable possibly leading to the oscillation of a variable, and accordingly, an overall performance degradation or missing the intended

[1] The content of this and subsequent sections are partially based on Nikolaus Koemm's Diploma Thesis [Koe11].



(a) Direct conflicts: Multiple coordination algorithms manipulating the same variable in a certain protocol.



(b) Indirect conflicts: Coordination algorithms manipulate different variables but nevertheless affect each other.

Figure 5.1 Conflict classification: direct and indirect conflicts.

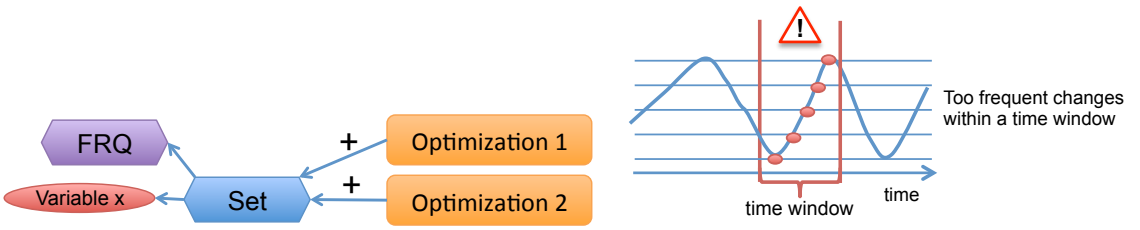
goals. For example, an energy related coordination is interested in saving energy by decreasing the transmission power. Conversely, a connectivity oriented coordination is interested in keeping long range connectivity by increasing the transmission power. Hence, when both of these coordination algorithms run in parallel, they could lead to the oscillation of transmission power and an overall uncoordinated behavior of the system.

Indirect conflicts are caused by multiple coordination algorithms that influence each other even though they do not manipulate the same variable. In this case, although several variables residing in separate protocols are manipulated, they still cause a functional dependency. This complicates the understanding of the dependencies and thus their detection. Therefore, indirect conflicts are significantly harder to detect than direct conflicts. Figure 5.1(b) shows variables of different protocols being manipulated by different cross-layer coordination algorithms. For example, two different coordination algorithms try to improve the ARQ error control at TCP and MAC layers, respectively. Two error control coordination algorithms enabled at the same time may lead to throughput degradations if not coordinated properly. This is because each of the ARQ error control schemes may cause additional overhead and waste capacity of the wireless conneciton which result in throughput degradation.

In the following sections we present extensions to CRAWLER to detect both conflict types, beginning with the simpler case of direct conflicts.

5.3.2 Detecting Direct Conflicts

Since in direct conflicts several coordination algorithms compete for the same variable, detecting such conflicts is rather simple. Thus, our first step is to determine the number of parallel coordination algorithms that are accessing the same variable. For this purpose, CRAWLER automatically counts the number of FUs using a *set-FU*. Remember that a *set-FU* is a special FU running in the CPC (in the kernel) of CRAWLER and allows to write into a specified system variable. As motivated above, the manipulation of a certain system variable by `set-FUs` acquired by different coordination algorithms is a strong indication for a potential conflict. However, since we assume that the manipulation of a single variable by several coordination algorithms



(a) Frequency FU FRQ checks if within a time window a variable is updated too many times.

(b) The FRQ FU detects that more than four modifications are conducted within a time window.

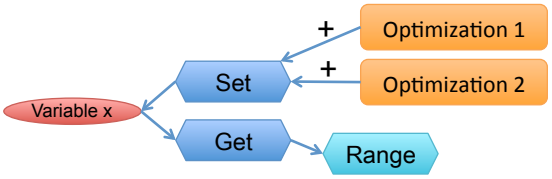
Figure 5.2 Multiple coordination algorithms try to change the variable frequently within a specified time interval. Our frequency checking FU FRQ is able to detect and report this to the conflict monitoring application.

is not a guarantee for conflicts (e.g., when two coordination algorithms with the goal of energy efficiency turn of the radio) and a subsequent performance decrease, we implemented the following three special FUs that helps us to understand and detect direct cross-layer conflicts.

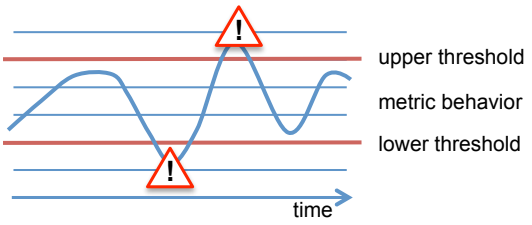
Frequency FU (FRQ): This FU counts the number of accesses to a variable within a time window (period). The higher the frequency of access to a variable is, the greater is the probability that many coordination algorithms tried to mutually adapt this parameter. This could result, for instance, that one coordination algorithm adapts a variable although it has already been modified by another one leading to a suboptimal state of that variable for the whole system. Figure 5.2(a) shows how two different coordination algorithms try to access and manipulate a variable. Each time a change occurs, the FRQ takes notice. The frequent changes may occur due to the reason that the coordination algorithms work contrarily. The FRQ can detect these frequent changes and can, for instance, provide necessary feedback to a conflict monitoring application or to other FUs. Figure 5.2(b) shows and example where the FRQ detects too frequent changes within a time window.

Range FU: This FU checks if the variable exceeds values from a certain predefined range. If, for example, two coordination algorithms increase or decrease a variable simultaneously, the value of the variable may result in an irrational or undesirable state and thus leading to misbehavior. Figure 5.3(a) shows two coordination algorithms that are trying to increase a variable which is also monitored and checked by the Range FU if the changes exceed the predefined thresholds. Figure 5.3(b) emphasizes when the Range FU considers misbehavior and notifies other FUs or a monitoring application.

Oscillation FU (OSC): This FU observes if the values of a certain variable are fluctuating considerably. It has two functions: First, the FU observes whether two subsequent assignments of a variable deviate beyond a certain predefined margin, reporting a possible misbehavior. Second, it also provides the ability to measure the scale of the deviation. For example, if the sampling frequency of the FU is set too high, then single peaks within a short timeframe are detected as misbehavior as illustrated in Figure 5.4. In contrast, if the sampling frequency is set to a low value, then short peaks are not considered but alterations over a longer period can be detected as misbehavior.

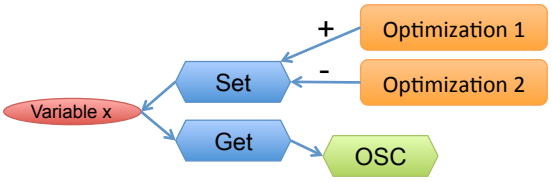


(a) Range FU checks if lower and upper thresholds are violated.

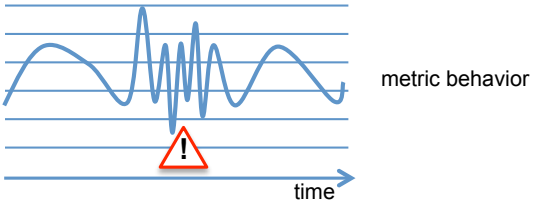


(b) Metric exceeds upper and lower thresholds which is detected by the `Range` FU.

Figure 5.3 Multiple coordination algorithms lead to an excessive increase or decrease of a certain metric. Our range FU checks if the value is unintentionally changed too much resulting in out of bound increase.



(a) Oscillation FU checks if a certain metric fluctuates excessively in a certain period of time.



(b) Metric fluctuates too much in a short period which is detected by the Oscillation FU.

Figure 5.4 Multiple coordination algorithms causes a certain metric to fluctuate in a short period of time. Our Oscillation FU checks (i) if the change exceeds certain thresholds too many times and (ii) checks in which time frame this occurs.

These three FUs assist a developer in detecting possible conflicts when running multiple coordination algorithms and extends Crawler's set of FUs. To use these special FUs, similar to the remaining FUs (as introduced in Section 3 and listed in Appendix B.2), requires to specify a Crawler configuration. Thus, besides using Crawler's configuration language for describing rules for cross-layer coordination algorithms, we also augment the configuration with rules for conflict detection. This process is similar to writing programming code (i.e, cross-layer coordination algorithms) and adding debug-information such as assertions (i.e., rules for conflict detection).

In the following we discuss Crawler's support to detect the more complex case of indirect conflicts.

5.3.3 Detecting Indirect Conflicts

Indirect conflicts can introduce complex interdependencies among variables of different protocol layers resulting in performance degradation in terms of metrics such as energy, throughput, jitter or delay. To detect indirect conflicts Crawler monitors metrics (specified by a developer) based on the recently observed network traffic conditions and application requirements.

Besides observing current network conditions, Crawler also needs first-hand knowledge about the application requirements to determine whether multiple concurrent coordination algorithms are in conflict with each other. For example, a throughput

decrease during a VoIP call initialization might be irrelevant for a VoIP application but delay is not. Therefore, the performance requirement of an application is an essential information to decide if the performance degradation is relevant for detecting a conflict. If performance deteriorates with respect to any of the collected metrics (i.e., network conditions and application requirements), CRAWLER notifies all the registered applications regarding the possible occurrence of a conflict among concurrent coordination algorithms.

Hence, to find out the relevant metrics, we classify applications. Our classification is based on the comprehensive QoS based classification in [AR99]. We used this classification scheme as it provides the details for threshold settings and comprehensibly groups applications. Alternatively it is also possible to use other application classification schemes.

However, after agreeing on a certain classification scheme, we assume that the class of the application that it belongs to is given by the application developer as they know best about requirements. Thus, in our current implementation, the QoS class of an application is statically determined when the application is registered with CRAWLER. We use a specific conflict monitoring application that keeps track of the registered applications and their classes. It also instructs Wbest [wbe] to probe and monitor for current network conditions such as bandwidth and delay. After registration of an application, the conflict monitoring application monitors the relevant conditions for the respective application classes. Even if many applications are loaded simultaneously, the corresponding rules for monitoring are established automatically.

5.3.4 Related Work

The problem of cross-layer conflicts have been first described in [KKTC05]. In particular, the authors describe two coordination algorithms where each lead to performance improvement when running exclusively. Interestingly, although each of the two coordination algorithms have been designed to improve the performance, running both in parallel decrease performance. Many papers [SM05, CMTG04, Wil08] reference this work by only describing the existence and complexity of the problem. But to the best of our knowledge, none of them classify the problem space to the degree that we do and non of them provide solutions to tackle the problem. Accordingly, we are the first who provide a tool to support developers to experiment with multiple cross-layer coordination algorithms and to address the problem of cross-layer conflicts.

5.3.5 Summary and Discussion

We first classified the cross-layer conflicts into two classes, that is, direct and indirect conflicts. Afterwards, we showed how we extended CRAWLER with debugging capabilities to support the detection of these two classes of conflicts. To address the problem of direct conflicts, CRAWLER automatically counts the number of coordination algorithms accessing a certain variable. This already provides hints about potential conflicts. For more advanced and manual debugging, we added further

FUs to provide developers sophisticated debugging support. Regarding indirect conflicts, we extended CRAWLER with Wbest to monitor network traffic conditions. Afterwards, based on the observed network conditions and application demands, CRAWLER informs all applications about a conflict.

However, for more sophisticated debugging, for instance, in case of indirect conflicts, again the whole set of FUs and stubs of CRAWLER can be used to build rules for conflict detection. Moreover, while experimenting CRAWLER allows to monitor several system variables (see Appendix B) which help developers to analyze the system behavior and to adjust the set of running cross-layer coordination algorithms to environmental conditions.

5.4 Cross-Layer Redundancy Removal

We believe that since application developers know best about what their applications need and how to make them fully adaptive, they should have the feasibility to provide cross-layer coordination algorithms. Moreover, as applications start and terminate unpredictably (i.e., based on user requirements), adding and removing cross-layer coordination algorithms at runtime is a further requirement. In such a case the major problem is, in addition to cross-layer conflicts, that multiple independently added cross-layer coordination algorithms[2] could have common processing states wasting unnecessary CPU time and memory. For instance, many applications use localization information which is processed and aggregated over and over again for each application. Having one instance of that algorithm which many applications can reuse is reasonable.

In the following we focus on the problem of redundant processing of multiple cross-layer coordination algorithms and provide first a general graph-based approach that automatically detects redundant parts of cross-layer coordination algorithms and removes them from the system. Afterwards, we present the specific implementation for CRAWLER which allows to add, remove and change cross-layer coordination algorithms at runtime, and enables third-party application developers to independently insert their own set of cross-layer coordination algorithms. As a result, CRAWLER is notably vulnerable to redundantly running cross-layer coordination algorithms, which can be detrimental for system performance. Since we believe that cross-layer architectures should support applications with bundled cross-layer coordination algorithms (as discussed in Section 3.4.2), this necessitate the need for mechanisms that can detect and resolve redundancies in multiple cross-layer coordination algorithms.

Since CRAWLER employs module-based[3] software development (i.e., modules are composed together) to realize cross-layer coordination algorithms, redundant parts of cross-layer coordination algorithms can be found by exploring equal module compositions[4]. To achieve this, our approach iteratively compares each pair of modules

[2]Note that we explicitly use the term cross-layer coordination instead of cross-layer optimization to avoid misunderstanding with the term optimization.

[3]We consider a module as function that contains all the source code and variables necessary to realize a certain self-contained functionality.

[4]The content of this and subsequent sections are partially based on the joint work with Martin Henze, Muhammed Hamad Alizai, Kevin Möllering, and Klaus Wehrle published in "Graph-based Redundancy

in a composition. To determine whether or not two modules are equal, it analyzes each module and its connections. Afterwards, it rewires the connections of equal modules and removes the redundant module. As this approach is based on a generic graph-based algorithm, it is not peculiar to a specific development platform and can be utilized across a wide range of modular software development systems or networking scenarios.

5.4.1 Generic Design

Solving the problem of redundancy in cross-layer coordination algorithms requires the analyzation of program semantic to discover redundant functionality. Without the availability of semantic knowledge that has been provided with huge effort by developers, this is an undecidable problem according to Rice's theorem [Hro03]. Fortunately, there exist techniques or relaxations to tackle such problems and to achieve practically useful solutions. By composing modules to realize cross-layer coordination algorithms, the cross-layer architecture CRAWLER provides a good basis to tackle a relaxed problem. Before describing a concrete solution, we generalize the problem of redundancy of cross-layer coordination algorithms as redundancy in module compositions which makes our solution also applicable to other fields.

For our relaxed solution we particularly opt for an automatic solution without requiring developer interaction and the need for delving into program semantics because (i) it would require complex formal description of each module and its connections, and (ii) it overburdens the developers with a significant effort required to support and create such descriptions. When we talk about developers, we distinguish between module developers who create the reusable modules once and module composition developers (also known as system or software integrators) who utilize these modules to implement a certain algorithm. With our approach the former need to put effort once, while the latter are unburdened.

5.4.1.1 Constraints

The basic idea of our approach is to merge redundant module compositions together. But before discussing the technical details of our solution, we first discuss our primary constraints and assumptions that form the basis of our approach. For example, our constraint with regard to merging of redundant modules is that the modified system has to offer the same behavior as the unmodified system. Therefore, the basic requirement for handling modules is the ability to determine if these modules are *equal*.

Consider that we have two *equal* modules within a system, that for each input generate the same output. Any coordination that merges these modules has to fulfill two major constraints:

Removal Approach for Multiple Cross-Layer Interactions", 6th International Conference on Communication Systems and Networks (COMSNETS '14), January 2014 [AHA+14]. Furthermore, the content is also partially based on Kevin Möllering's Bachelor Thesis [Möl11].

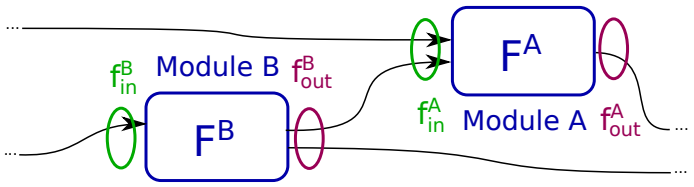


Figure 5.5 Example representation of two modules A and B with their functionals, input and output functions.

(1) *Output correctness* – If the merging process results in a modified composition of modules in a system, the output of the new system must be the same as the output of the unmodified system.

(2) *Modification transparency* – If a module composition developer modifies a composition of modules, the system must not require her to be aware of the merged compositions. In other words, a module composition developer should be able to parameterize modules and their compositions without having knowledge about the underlying optimized compositions.

5.4.1.2 Equality of Module Compositions

With respect to our constraints, to decide if a system (behavior) remains the same after merging modules, the modified and unmodified system must have the same behavior, i.e., produce the same output on a given input. In any system, the output of the system is a subset of the output of the contained modules. Thus, providing equality between the outputs of all modules in the original and the modified system is sufficient to prove the overall equality of the outputs of both systems. To achieve this, two properties must hold for every module: (i) the behavior of the module must remain the same and (ii) the inputs of the module have to be the same.

Before describing how to verify if both of these properties hold, we give a formal description of the problem which helps to describe the idea of the algorithm and simplifies the adaptation to other networking problems which we discuss later in Section 5.4.7. The description is based on the interpretation of a program module as a functional. As shown in Figure 5.5, a module M has an input function f_{in}^{M} and an output function f_{out}^{M}. The mapping of the input function to the output function is done by the *module functional* F^{M}. For example, let us consider a `Sum`-Module M with the following input function $f_{in}^{M} = (4, 1)$. Since the module calculates the the sum of its inputs, the output function is $f_{out}^{M} = (5)$.

Thus, the functional F^{M} maps an input function f_{in}^{M} to the corresponding output function f_{out}^{M}. If one can verify that two modules A and B provide the same functional, i.e., $F^{A} = F^{B}$, they both produce the same output, given the same input.

Definition *(Equality):*
A module A is equal to a module B (and thus mergeable) if and only if the following two prerequisites hold:
(1) *input equality* – the input functions of both modules are equal, i.e., $f_{in}^{A} = f_{in}^{B}$, and
(2) *behavior equality* – A and B have the same *module functional*, i.e., $F^{A} = F^{B}$.

Based on these two conditions we obtain output equality, i.e., $f_{out}^{A} = f_{out}^{B}$. In the following, we present an algorithm that checks particularly for these two conditions.

5.4.2 Graph-based Iterative Merge Algorithm

An intuitive algorithm for merging subsets of equal module compositions can be directly extracted from the requirements described in the previous section. Thus, two modules can be classified as mergeable if they satisfy the input and behavioral equality property. To achieve this, our algorithm consists of three main steps: (i) Check if two selected modules satisfy the equality property, i.e., can be merged, (ii) if yes, merge them together, and (iii) repeat the previous two steps until no merges are possible anymore. Listing 5.1 shows our algorithm.

We now discuss the major requirements for satisfying input equality (cf. mark `(1)` at Listing 5.1), behavior equality (cf. mark `(2)` at Listing 5.1) and the merging process (cf. mark `(3)` at Listing 5.1) in more detail. Afterwards, runtime and memory consumption estimates are provided for our algorithm.

```
1 input: m_graph
2 output: m_graph
3 operation iterative merge begin
4     changed <- true
5     while (changed) do
6         changed <- false
7         for all (A in m_graph[modules]) do
8             for all (B in m_graph[modules]) do
9                 if(A!=B AND
10 (1)              inputs_equal(A,B,m_graph) AND
11 (2)              behavior_equal(A,B)) then
12 (3)                  m_graph <- merge_modules(A, B, m_graph)
13                      changed <- true
```

Listing 5.1 Graph-based iterative merge algorithm

5.4.2.1 Input Equality

In a real system, the equality of inputs of two modules can be assured if all of the following three prerequisites hold: (i) The specific connection has to originate from the same source, i.e., from exactly the same node in the module composition graph. (ii) The position of a specific input connection within the input vector has to be the same, e.g., if we consider the position of variables a and b, then $isLess(a, b)$ is not the same as $isLess(b, a)$.

```
1 input: A, B, m_graph
2 output: equal
3 operation inputs_equal begin
4     size1 = size_of(inputs_module(m_graph[inputs],A))
5     size2 = size_of(inputs_module(m_graph[inputs],B))
6     equal <- size1 = size2
7     if(equal) then
8         for all (in1 in inputs_module(m_graph[inputs],A))
9             found <- false
10            for all (in2 in inputs_module(m_graph[inputs],B))
11                if(source_of(in1) = source_of(in2) AND
12                    position_of(in1) = position_of(in2) AND
13                    type_of(in1) = type_of(in2)) then
14                        found <- true
15            equal <- equal AND found
```

Listing 5.2 Checking if two modules A and B have the same input.

(iii) In a system with more than one connection type (e.g., information flow or detectable events), the type of a specific input connection has to be equal. Prerequisites (ii) and (iii) basically enforce that the type signatures of the modules are identical, while prerequisite (i) enforces that they are always called with the same input. Combining these three prerequisites leads us to the input equality computation algorithm, as shown in Listing 5.2.

5.4.2.2 Behavior Equality

Behavior equality in general addresses the question whether one program behaves in the same way as another program does. This non-trivial property is undecidable according to Rice's theorem [Hro03]. Since a module is just a representation of an arbitrary program, there is no assumption that holds in every module-based software system to solve this problem. As we want to provide an automatic solution without investing the manual effort of semantic descriptions, the only possibility to check for behavior equality is an exhaustive state search. This exploration for each possible state within the complete state space may lead to the state-explosion problem [GK02]. Thus, this only works for applications with a finite and specified runtime. Moreover, if the exhaustive state search has to be performed for any possible input, it becomes infeasible and useless in most cases. To overcome this issue, we identify a possibility to relax this problem. We assume that there are only deterministic modules in the system. Thus, for deterministic modules one specific instruction set which is running on one specific state produces exactly the same output on every run. This allows us to base the decision for behavior equality on the following two conditions.

Definition *(Behavior Equality):*
Two modules A and B have an equal behavior if the following two conditions hold:
(1) *module type equality* − the implementations for both modules are exactly the same, and
(2) *module state equality* − the current variable allocation and execution position, i.e., states of the modules, are identical.

Due to the determinism assumption, equal state and type implies the exact same behavior on any equal input. Note that the reverse direction for this implication does not hold. Since we cannot provide an equivalent property to behavior equality, there may be cases where $F^A = F^B$ holds, which cannot be found with our algorithm. However, it will not lead to false optimizations, but only miss merging possibilities related to modules offending our assumptions. Furthermore, we assume that it is very unlikely to have two different implementations of exactly the same functionality. By introducing this relaxation, we identify a sufficient condition for behavior equality of two modules A and B, which can be practically implemented.

Although implementations of both conditions are application-specific, module type equality can often easily be checked by introducing a numeric type identifier or comparing the memory. We suggest that the comparison of module states should reside inside the module since module developers know best about their modules and relevant states. Accordingly, module developers can build in a module state

comparison function to verify if the module's state is equal to the state of another module. This only has to be done once. Afterwards, module composition developers can utilize modules and combine them without considering possible optimizations as our approach will automatically optimize the entire module compositions in the whole system.

5.4.2.3 Merging Modules

Once the algorithm finds two modules that satisfy the *behavior equality* and the *input equality* prerequisites, they can safely be merged. Merging two modules A and B is straightforward (cf. Listing 5.3):

```
1 input: A, B, m_graph
2 output: m_graph
3 operation merge_modules begin
4     for all (inc in in_connections(m_graph[inputs],A))
5         delete_input(inc)
6     for all (outc in out_connections(m_graph[inputs],A))
7         replace_source_of_connection(outc,B)
8     remove_module(A)
```

Listing 5.3 Merging two modules A and B and removing A afterwards.

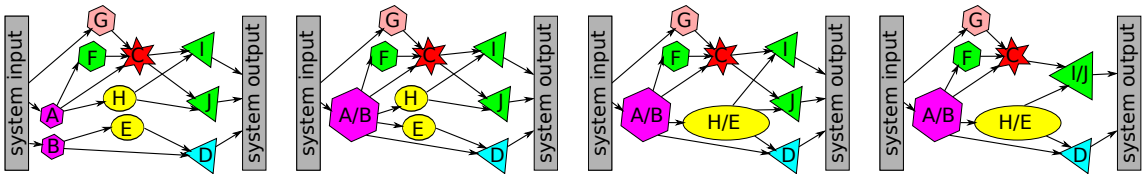
Module A is removed together with all its incoming connections (as these are already provided by module B). The outgoing connections of module A are then rewired to be outputs of module B.

5.4.2.4 Runtime and Memory Consumption

The runtime of our algorithm is determined by its three nested loops (cf. Listing 5.1) and the three marks. The two *for* loops iterate pairwise over all modules in the graph, i.e., $\mathcal{O}(|V|^2)$ with V being the set of modules. Within these two *for* loops each time the three marks are checked. At mark `(1)`, the `input_equal` function (cf. Listing 5.2) also iterates over two nested *for* loops. It first iterates over the connections in A and afterwards over all connections in B where it compares three parameters and accordingly has a runtime of $\mathcal{O}(|E|^2)$ with E being the set of connections. Mark `(2)` has a runtime of $\mathcal{O}(1)$ where the type and states are compared. The merging at mark `(3)` has a runtime of $\mathcal{O}(|E|)$ where connections are removed or rewired. If an equal module is found, all these operations are conducted again for the remaining modules. Therefore, the outer *while* loop requires a runtime of $\mathcal{O}(|V|)$, accordingly leading to a total runtime of $\mathcal{O}(|V|^3 \cdot |E|^2)$ for our algorithm.

With regard to the space complexity, our algorithm has no recursive calls that would increase the stack size of the program. Traversing lists requires the amount of space that is asymptotically equal to its length, i.e., $|V|$ for lists of modules and $|E|$ for lists of connections, i.e., in total $\mathcal{O}(|V| + |E|)$.

In a runtime-adaptable system, where modules and their compositions are unloaded, it is sometimes necessary to split modules again. This can happen, e.g., when the input or behavior of a merged module is changed at runtime. The module has then to be split (similar to the copy-on-write paradigm) again. How our approach handles this splitting is described in the next section.



(a) Not merged. (b) Merged A and B. (c) Merged H and E. (d) Merged I and J.

Figure 5.6 A run of our algorithm on an example module composition. The type of shapes defines the type of the modules, while the colors define its state.

5.4.3 Runtime Adaptation

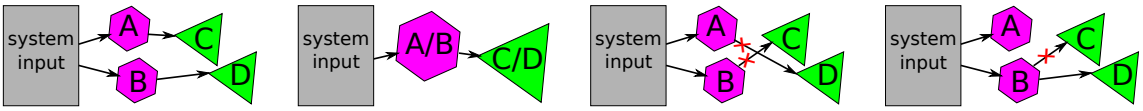
After discussing the challenges of the behavior and input equality constraints, and formulating an appropriate solution for it, we now turn our focus towards the *modification transparency* constraint: It requires that the optimizations induced by the merging process to remain transparent to the module composition developers who intend to manually modify module compositions. Although this desired transparency constraint is inherently achieved in static systems by providing *output correctness*, runtime-adaptable systems pose further challenges: E.g., to ensure *modification transparency*, the set of commands required to achieve a desired reconfiguration of the system has to be the same before and after the merging process.

However, in a runtime-adaptable system this is difficult to guarantee because of two types of reconfigurations. First, the reconfigurations that occur due to a change in the functionality of modules. These kind of reconfigurations could occur, for example, when removing modules, modifying the state of a module, or modifying the function of a module. Second, the reconfigurations that occur due to a change in module compositions. Such reconfigurations could occur, for example, when modifying incoming data connections of a module (e.g., adding new connections or modifying connection properties).

Such configurations are usually done in an adaptation engine encapsulating the construction logic which sends commands to the adaptable software realizing the actual implementation [ST09]. Commands are instructions on how to modify or rather configure the adaptation software such as creating an object and connecting it with other objects. Our redundancy removal system can be placed in between these two parts to track commands. After receiving a command C, our redundancy removal system can either modify the command to fit it to the optimized version of the system obtained after the merging process, or revert specific merging optimizations to allow C to be processed normally.

5.4.3.1 Challenges when Adding/Removing Modules and Connections

The challenging nature of the commands that induce modifications on modules that have been merged is visualized in an example depicted in Figure 5.6. The shape of the modules represents the type while the color represents its state. For the sake of simplicity, we assume all data connections to be equal with respect to their type, ordering and any other feature. We can see that the modules A and B can be merged



(a) Original state. (b) Merged state. (c) Possible splitting mistake. (d) Another splitting mistake.

Figure 5.7 Possible problems when not providing a reasonable history of the connections and splitting multiple modules.

after validating state, type, and input equality. Similarly, the next iterations of our algorithm will also merge modules H and E and afterwards I and J.

Modification transparency requires that any developer who changes the module compositions does not need to know about the underlying merging optimizations. Thus, the developer assumes working on the first composition of modules (cf. Figure 5.6(a)) even though the system has been optimized by the merging process resulting in modules compositions shown in Figure 5.6(d). In this example, the module G can be modified by a developer without violating *output correctness* or introducing any further ambiguities. However, any modifications either to the input of module A or its functionality, would change the output of A. Clearly, this modification should not have any effect on module B.

In order to avoid modifying node B, there is a need for the ability to split both modules again. However, the merging process has also merged modules H and E considering that A and B are mergeable (cf. Figure 5.6(b) and 5.6(c)). Since this prerequisite will no longer be valid after the modifications introduced by the developer to module A, we have to split H and E as well. As modules I and J have been merged independently of H and E, it is not required to split I and J.

5.4.3.2 Splitting Affected Modules

Splitting modules requires to maintain the knowledge about both the outgoing connections and the respective modules before the merging process. We propose a simple two step procedure which has to be performed before merging modules. In the first step, we identify all those modules that were merged with a certain module A due to their equality. In the second step, we split module A and all the merged modules that were identified in the previous step by restoring the original configuration of the system, i.e., reverting back to the original connections. For this purpose, we also need a mechanism to be able to access the previously defined connections. Otherwise, the splitting process could end up in misconfigurations.

To clarify this requirement, let us consider the example in Figure 5.7. If there are two or more levels of connected modules that can be merged as shown in the Figures 5.7(a) and 5.7(b), the original source of data connections can vanish. In Figures 5.7(c) and 5.7(d), two different compositions are shown that could result due to the lack of information about the connections within the original system. Both wrong compositions result from splitting the merged state in Figure 5.7(b). Maintaining information about all the original connections is dependent upon the actual implementation of the system, and hence, beyond the scope of discussion in this thesis. Nonetheless, for the sake of completeness and practical applicability, in

the following section we do provide a domain-specific solution to demonstrate a full fledged implementation of our mechanism for a cross-layer coordination architecture.

5.4.4 Specific Design for CRAWLER

After providing the theoretical basis for our approach, we now demonstrate its applicability in the domain of cross-layer design using CRAWLER. The mapping of the general approach to the needs of CRAWLER requires to map the three functions: (i) input equality, (ii) type equality and (iii) state equality.

But before discussing the three features remember that in CRAWLER we refer modules as functional units (FUs). FUs (see Section 3.4.3) are a specific realization of modules and offer properties such as a unified interface allowing flexible composition and being stateful, i.e., FUs can maintain a record of data and provide results based on the maintained record, for instance, a `History-FU`.

Coming back to the three features and beginning with **input equality**, CRAWLER provides two different types of input (or connections) between FUs: (a) a notify connection that is an event-based signaling and (b) a query connection that is a polling mechanisms. Redirecting connections are implemented as simple pointer redirections in C. While *notify connections* can be directly mapped onto outgoing data connections of the type `notify`, query connections have to be reversed. A query connection from one FU A to another FU B is realized by making A ask B for a piece of information. Thus, in reality information flows from B to A. Therefore, we interpret an outgoing query connection as an incoming data connection of type `query`. Furthermore, since the order of query connections matters, we have to take that ordering into account.

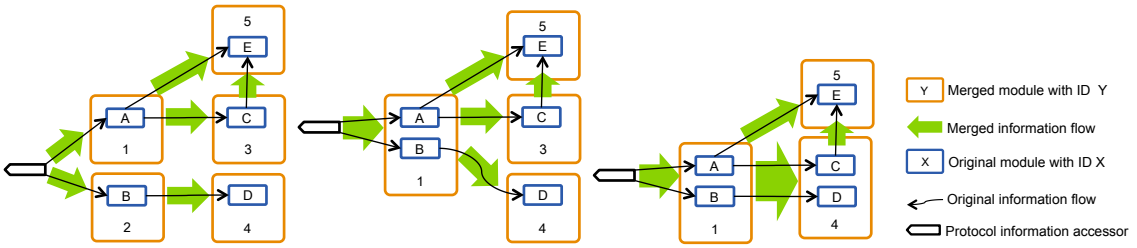
Implementing **type equality** checking in CRAWLER is straightforward, since each FU carries an identifier. For example, an object of the `AND` FU contains the numerical identifier `13` indicating the type. Accordingly only this numerical value has to be compared for type equality.

In contrast, **state equality** is a bit more complex. Since a comparison based on the internal state has to be provided by the FU, each functional unit developer has to decide how to implement the function that determines equality. CRAWLER's standard FU structure predefines such a state equality interface which is called by our merge algorithm. For example, the `History-FU` stores an amount of values. When the equality interface is called, the `History-FU` compares the values given in its signature with the stored values. Note that the FU developers know best about their FUs and accordingly need to implement once about what they consider as state equality. Afterwards, developers of cross-layer coordination algorithms can utilize them without needing to care about our merge algorithm.

So far, we know how to determine equality of FUs in CRAWLER, but we still need to deal with runtime adaptation. In the following we show how we extended CRAWLER to handle runtime adaptation of FU compositions.

5.4.4.1 Handling Runtime Adaptation

As discussed in Section 5.4.3, if module compositions change at runtime, we need to be able to keep track of the original module composition. We use CRAWLER's



(a) Each original module (orange) corresponds to exactly one merged module (blue).

(b) The modules A and B are realized by the module with merged ID 1.

(c) Additional iteration of our algorithm also merges original modules C and D to the merged ID 4.

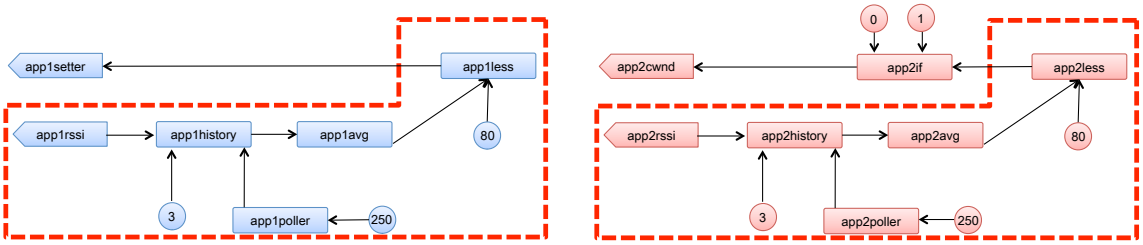
Figure 5.8 By reading off all original modules and connections the original composition can be extracted. By only considering the modules that reside in the CPC, one can get the *merged* composition which consists of only 3 modules. This example is restricted to query connections.

repository residing in the LC to keep track of such adaptations. Remember, if a developer changes a configuration, these changes are translated into commands that are delivered both to the repository and CPC. Here, the repository has two advantages: (i) It behaves similarly to a revision control system: Each time the configuration changes, the commands are automatically committed as a new revision. (ii) Unnecessary context switches between user and kernel space are avoided. As a result, the repository provides a good overview of running compositions and allows the developer to roll back to a previous set of cross-layer coordination algorithms if necessary.

We extend the representation of FUs and connections in the architecture to keep track of FUs both in the original and merged compositions. For this purpose, we store two different FU identifiers for two different views: (i) The *original view* consists of original FUs and their original connections which in fact represents the initial and unmodified composition graph. (ii) The *merged view* is used for handling the merged version of FU compositions, i.e., the optimized graph. For clarity reasons, the concept of the original and merged views is combined in one graph as shown in Figure 5.8. Additionally, for the sake of simplicity, we only consider one type of connections in this example representing the information flow in the graph.

Each node in Figure 5.8 represents one merged FU which encapsulates one or several original FUs. Similarly, a connection represents a merged connection encapsulating several original connections. The original FU and connections are necessary to recreate the original FU compositions. Figure 5.8(a) shows the FU compositions in the initial unmodified state. In Figure 5.8(b), the FUs with identifier 1 and 2 are merged and their original identifiers A and B are stored in the FU along with one of the merged FU identifiers, in this case identifier 1. Similarly, in Figure 5.8(c), FUs with identifiers 3 and 4 are merged along with their connections, in this case the merged FU with the identifier 4 is used for merging the original FUs C and D.

In the following we show a real-world example of how our algorithm merges and splits FU compositions automatically after a runtime adaptation occurs.



(a) FU composition for cross-layer coordination 1. (b) FU composition for cross-layer coordination 2.

Figure 5.9 FU composition of two installed cross-layer coordination algorithms. The red dashed-line indicates equal composition detected and merged by our algorithm.

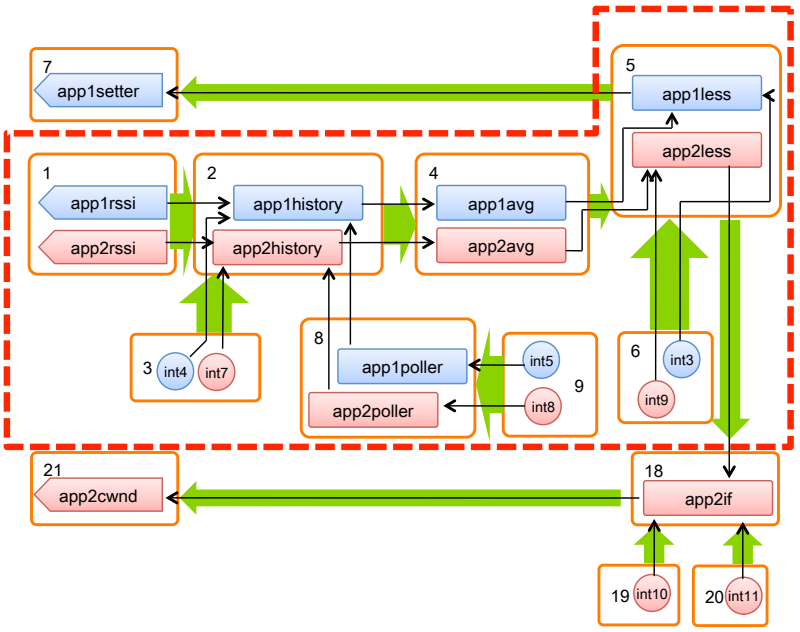
5.4.5 Evaluation and Validation

To verify the correctness of our proposed algorithm, we performed a complete system test. We loaded the CRAWLER architecture with an empty initial configuration, i.e., no cross-layer coordination algorithms were running in the beginning. Then we started two applications that use CRAWLER's API to feed a cross-layer coordination into the system. Although the naming of FUs and the overall compositions differ for both applications, some parts are equal and thus we expect that our algorithm will merge them. However, later on we will modify one application's FU composition at runtime and accordingly expect that our algorithm splits affected compositions and conducts the necessary modifications. In the following we show the cross-layer coordination for the two applications.

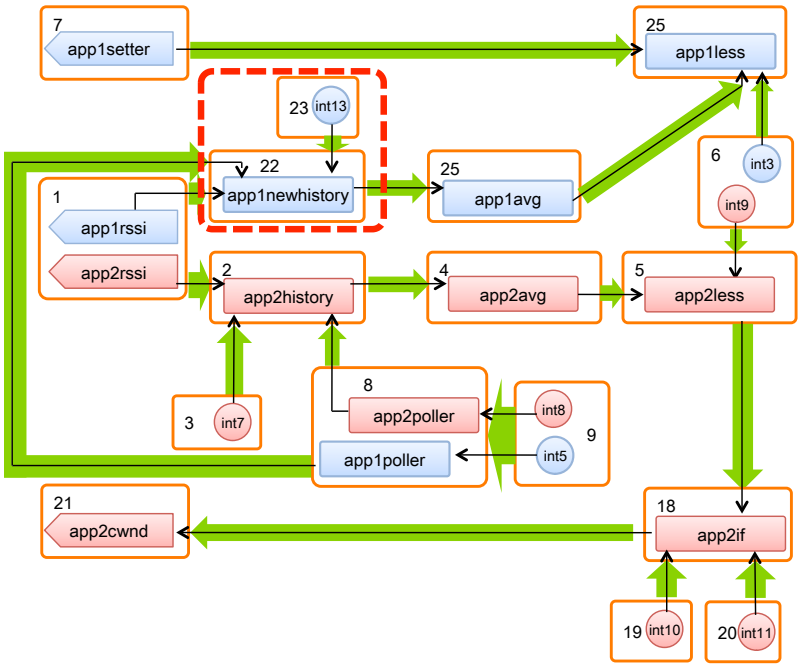
Cross-layer coordination of application 1: The first application is interested in knowing whether the current received signal strength indicator (RSSI) of the wireless connection is good or bad. The RSSI is measured every 250 msec. To reduce the amount of oscillation in this signal, the base for the decision is calculated by averaging over the last three values. If the result does not exceed a threshold of 80, the signal is defined as being bad. The resulting FU composition is shown in Figure 5.9(a).

Cross-layer coordination of application 2: The configuration used by the second application implements a simplified version of a cross-layer coordination for the TCP congestion control algorithm [ASA+12]: It monitors the RSSI of the wireless device and freezes the congestion window size (CWND) in TCP if the system operates under bad WiFi conditions. This prevents the congestion control algorithm to reduce its congestion window and thus entering the slow start phase. As shown in [GMPG00], the slow start of TCP is unsuited for short-term disturbances on the physical layer. To stop the CWND from trying to adapt to the conditions, a 0 is written to the stub accessing the corresponding TCP variable. The composition for this example is visualized in Figure 5.9(b).

Runtime modification: By using CRAWLER's API, application 1 feeds its cross-layer coordination algorithms into the system. Shortly after, Application 2 also feeds its cross-layer coordination algorithm into the system. Our algorithm automatically detects equal compositions. Due to the given input, state and type equality, the algorithm merges the chains starting at `app1less` and `app2less` respectively. Figure 5.10(a) depicts the result after the merging process.



(a) Resulting composition after the merging with our algorithm. The red dashed-line shows merged FU compositions.



(b) The `History-FU` is exchanged at runtime by another new `History-FU` (red dashed-line) . This causes our algorithm to split effected FUs.

Figure 5.10 Merging equal FU compositions and splitting them if necessary, i.e., if changes are conducted at runtime to FUs and to their compositions.

After Application 2 has inserted its cross-layer coordination algorithm, Application 1 adapts its coordination. One possible reason could be that the resulting information about the link still jitters too often. To smoothen the output, Application 1 replaces the `History-FU`, which stores three values, by another FU that stores ten values. In Crawler this runtime adaptation can easily be expressed with the following two lines:

```
1 app1newhistory:history(app1rssi,10)
2 replace(app1history,app1newhistory)
```

The first line creates a new `History-FU` with 10 elements and the second line instructs to exchange the old `History-FU` with the new one. When Application 1 tries to exchange `app1history` by `app1newhistory`, the connections (incoming notify connections and the outgoing query connections) of `app1avg` are modified. Thus, we expect that our algorithm will split `app1history` from the FU it has been merged with. Furthermore, we expect that the affected FUs `app1avg` and `app1less` are also splitted since they do no longer share input equality with `app2avg` and `app2less`. Figure 5.10(b) shows that the modification request conducted by Application 1 has been successfully and transparently realized by our algorithm. To conclude our evaluation, our algorithm is able to merge, i.e., optimize, and split FU compositions at runtime. The reason why we don't give detailed memory and CPU usage numbers for that particular example is twofold. First, the FU compositions are realized in the kernel part of Crawler as loadable kernel modules and retrieving such numbers for kernel modules are tedious. Second, providing numbers for that particular example is not representative and meaningful: It is easy to construct composition examples with more or less savings.

5.4.6 Related Work

Modular software development process (or modularization) [SPKW07, KMC+00, Weh01, HP91] is widely used in the scope of networking in order to deal with this rising complexity in designing, implementing and maintaining protocols for distributed systems. Modularization has also proved to be very practical in the cross-layer coordination domain [ASA+12]. But in spite of, that problems occurring with multiple cross-layer coordination algorithms is a very unexplored field. Only the problem of conflicts, i.e., possible performance degradations [KKTC05] caused by multiple, contradicting coordination algorithms, have been mentioned. To the best of our knowledge, the problem of redundancy in multiple cross-layer coordination algorithms have not been addressed so far.

By transforming the problem of redundancy in cross-layer coordination algorithms to redundancy in module compositions, we can **theoretically** consider it as using directed and labeled graphs. Here, a module is mapped to a node, while a data flow is mapped to a directed edge from the generator (source) to the receiver (sink). This allows us to research a suitable algorithm in the field of graph theory. While there are many interesting findings in the field of subgraph isomorphism, research on the *isomorphic subgraph problem* seems rather scarce. While the isomorphic subgraph problem cannot easily be reduced to other better-known graph isomorphism problems, in [BB02] it is proven to be $\mathcal{NP}$-hard by reduction to the 3-partition problem. If we consider circle free graphs, our algorithm is able to provide an optimal solution

in polynomial runtime. If circularities occur, we may miss optimization possibilities, however, we provide a suboptimal solution in polynomial time. All in all, the need for a polynomial time algorithm depends on the use case: (i) If modules and their compositions are regularly (un)loaded or changed (ii) how big the size of the modules and their compositions are, and (iii) how fast the system needs an optimization. Thus, it may be appropriate to use a variation of the isomorphic subgraph problem. However, since we provide a polynomial time algorithm, our solution will fit to all of these three uses.

Practically, in the field of runtime (self)adaptive software, there exist approaches which allow the exchange and modification of modules at runtime [ST09, PPS+09]. However, these approaches mostly use coarse granular modules such as in Eclipse, .net and the OSGi framework making it difficult to use our approach since the redundancies in such system will appear less often. Another solution that focuses more on ontology-based systems is [GWCA11]. This approach requires the developer to put effort to describe the semantic of their modules. In our approach the effort is much less since only a specific functionality (i.e., state equality) has to be compared. In particular, only the module developer has to implement once what she considers as state equality, but from then on due to the reusable nature of the modules, module composition developer (i.e., the software integrators) can freely compose without putting additional effort.

To summarize, we focus on an automated system that is able to resolve redundancy of multiple cross-layer coordination algorithms at runtime and does not burden the module composition developer about finding the overall optimal module composition.

5.4.7 Summary and Discussion

We proposed a graph-based redundancy removal algorithm to automatically detect and resolve redundancies in multiple cross-layer coordination algorithms. In particular, we transformed the problem of redundancies in cross-layer coordination algorithms to redundancies in module compositions. For this problem, we provided a general theoretical graph-based description, making it applicable for a wide range of modular systems or networking scenarios. Based on that, we suggested a general algorithm to automatically find redundant module compositions (i.e., parts of cross-layer coordination algorithms) and to merge them together. With regard to runtime-adaptable module compositions, we showed that more adaption then only a naïve removal of redundancies is necessary, since runtime adaptation can lead to invalid module compositions and accordingly suggest how to resolve this issue by bookkeeping. We validated the practical applicability of our approach by implementing it for our cross-layer architecture Crawler. Our evaluation demonstrated a real use case where we successfully resolved redundancies in cross-layer coordination algorithms at runtime and recreated the original state if necessary.

Although we have applied this generic approach for the cross-layer design domain, the problem can also be mapped to other scenarios such as to a graph of nodes in the network (wired and wireless networks) to detect redundant node compositions, services, and interactions. Detecting such redundancies can help in turning off services and even nodes, e.g., to save energy in battery driven devices and improve network life time. With our generic solution the adaptation to other fields should be simplified.

5.5 Conclusion

In this chapter we have introduced two different problems that appear when running multiple cross-layer coordination algorithms, namely conflicts and redundancy. For each problem, we have presented extension to Crawler to cope with the problems.

Regarding conflicts, we classified the problem into subproblems of direct and indirect conflicts and based on the subproblem suggested different approaches to improve Crawler monitoring capabilities to support developers to detect conflicts. In this context we have implemented and added further FUs to the set of reusable FUs which contribute to the versatility of Crawler in experimentation and monitoring of cross-layer coordination algorithms. However, on detection of a conflict applications including a special conflict monitoring application are informed.

Regarding the problem of redundancy in multiple cross-layer coordination algorithms, we proposed an algorithm to automatically detect redundant states of cross-layer coordination algorithms and resolve them. We first presented a general approach also applicable to other networking fields. Afterwards, we mapped the general solution to Crawler's needs and validated the approach. Compared to former approaches of conflict detection, the redundancy removal approach does not require developer feedback.

6

Evaluation Support for Cross-Layer Coordination

In the previous chapters, we presented Crawler, a versatile and runtime flexible cross-layer architecture to rapidly and conveniently design cross-layer coordination algorithms. We demonstrated its usability and flexibility on five use-cases from diverse fields of networking such as VoIP codec switching, TCP manipulations and jamming detection. Moreover, we also addressed and proposed solutions to problems that are involved when several cross-layer coordination algorithms are added into the system such as conflicting coordination goals and redundancy of cross-layer coordination algorithms. While these chapters primarily focused on the process of designing cross-layer coordination algorithms, this chapter focuses on the next step of evaluating the cross-layer coordination algorithms. In particular, we present a further major contribution of this thesis by answering our third research question, i.e., how to improve the evaluation of cross-layer coordination algorithms?

The remainder of this chapter is structured as follows. Section 6.1 analyzes the problems a developer faces when evaluating cross-layer coordination algorithms. In Section 6.2 we further elaborate these problems and give an overview about two approaches to tackle these problems. First, in Section 6.3 we present the extension of Crawler to support the experimenter during the evaluation process by enabling remote automation, configuration, and monitoring of cross-layer coordination algorithms. Second, in Section 6.4 we present Fantasy, a network emulation architecture that allows the fully automated setup and execution of experiments, enables convenient access to system information within the emulation and the collection of test results. Finally, Section 6.5 concludes this chapter.

6.1 Motivation

An essential step after implementing a cross-layer coordination algorithm is its evaluation. A very important concern in this regard is the monitoring of the system

behavior (relevant parameters) in order (i) to validate if the cross-layer coordination is working as intended and (ii) to measure performance improvements. This fact is comparable to debugging of software in general. In this context, a developer utilizes tools to build his code, but also has support in terms of debuggers to analyze the code. By enabling the ability to monitor states of variables and to control the program execution, debuggers support developers to understand their code and fix programming faults.

Such support is missing for the evaluation of cross-layer coordination algorithms. Although cross-layer architectures support the developer in implementing the intended cross-layer coordination, there is still a need for a feature to analyze the effects of a cross-layer coordination on the system. For instance, many factors such as wireless conditions or programming faults influence the behavior of the cross-layer coordination and accordingly the system. The ability to control or isolate these factors to fully understand and validate the behavior of a cross-layer coordination is missing. Moreover, with the absence of an ability to control these influencing factors in general and wireless effects in particular, many test runs have to be conducted till a certain credibility about the results is achieved. But in case that many testing devices are involved in the experimentation which also run diverse programs including cross-layer coordination algorithms, the execution of the experimentation might become very cumbersome, not even to mention the execution of several repetitions. In a nutshell, a tool to support the evaluation of cross-layer coordination algorithms would greatly benefit developers to validate and finalize their cross-layer coordination algorithms.

6.2 Problem Analysis

From the motivation presented in the previous section we derive two major problems that significantly increase the effort required to validate and finalize cross-layer coordination algorithms.

Lack of support for central and remote execution, configuration, and monitoring of cross-layer coordination scenarios

Testing cross-layer coordination algorithms can become very cumbersome when many nodes are involved. Each node might require to install diverse programs including the installation of the cross-layer coordination algorithms themselves. This step usually requires manual interaction. Although scripting is a common methodology, nonetheless the scripts have to be started and coordinated manually among all involved nodes. Moreover, after running the experimentation, developers are not able to remotely fine tune their cross-layer coordination algorithms. Any modification to the coordination algorithm would require the manual intervention of the developer on the respective device. Finally, at the end of the experiment test results have to be manually gathered from all devices. However, when there is a need to repeat the experimentations again, for instance, to gain more credibility about results, everything has to be conducted once again. Evidently, central and remote automation and configuration support among all involved nodes in the experimentation and the collection of test results can significantly support developers in the evaluation phase of their cross-layer coordination algorithms.

Lack of a method to isolate and control influencing factors
Typically a cross-layer coordination is designed for a certain use case such as protocol behavior improvement in a wireless environment, making its validation very difficult due to following reasons: (i) the volatile nature of the wireless medium, (ii) surrounding nodes accessing the medium, (iii) the cross-layer solution itself might have design flaws, (iv) programming errors, (v) no repeatability or reproducibility of results (vi) OS side effects and (vii) debugging limitations. With so many possible factors for misbehavior, understanding and validating the real benefits of the cross-layer solution is very difficult. We identify that isolation or control of these factors can help to understand the sources of problems while developing cross-layer coordination algorithms. In addition, the feasibility of automation and logging of the whole test scenario can further help to conveniently analyze and pinpoint the irritation factors.

This thesis present the two following approaches that tackle the above mentioned issues:

Remote Cross-Layer Evaluation
This approach primarily focuses on the lack of convenient evaluation support and is an enhancement of Crawler to improve the evaluation of cross-layer coordination algorithms in a real-world usage. In particular, we have extended Crawler's application support interface to remotely and centrally provide three complementary key features: (i) to automate and execute whole test setups with different settings remotely via a central device, (ii) to remotely add, remove, and modify cross-layer coordination algorithms, and (iii) the ability to monitor and log a specified set of variables and states of cross-layer coordination algorithms.

Network Emulation Tool – Fantasy
This approach complements the former by addressing the lack of a method to isolate and control influencing factors. We use network emulation as an approach to gain more control about the wireless environment which is running in a simulation environment. In particular, we couple Crawler with a network simulator resulting in a network emulation architecture that allows the fully automated setup and execution of an experiment in a controllable simulation environment to improve the monitoring and analysis of cross-layer coordination algorithms and their behavior. By using Crawler, the network emulation tool Fantasy enables convenient access to system information and collection of test results.

6.3 Remote Cross-Layer Evaluation

Testing of distributed systems is an exhausting process which is even more cumbersome when cross-layer coordination algorithms are involved. One major reason is the effort for **preparation** of the experiment as all involved nodes have to be booted and many specific programs including the cross-layer coordination algorithms have to be compiled and manually started. The running programs can be usually subdivided into two groups. First, the cross-layer coordination that is being evaluated such as an coordination to improve routing by using lower layer information as surveyed by

[QK04]. Second, helper software to support the evaluation of the former such as iperf or netem. Afterwards, the specific test nodes are ready for evaluation which typically consists of following additional effort.

First, **on starting** the software it usually requires different parameterization such as to adjust throughput, assign different IP addresses or to change thresholds for the link qualities considered in the cross-layer coordination. In this conjunction, an order of execution of the software might be relevant. For example, it might be reasonable to first start the routing protocol including the cross-layer coordination before starting the traffic generation. However, all of these aforementioned steps have to be performed manually on each node, requiring physical interaction by a developer. Ideally, remote automation of tests should be possible for a developer, i.e., for all nodes involved parameterization and timing of the software should be remotely and conveniently feasible.

Second, during **test execution** a developer might wish to change the parameterization of the cross-layer coordination such as changing the thresholds for the link quality used by the routing algorithm or switching TCP's congestion control algorithm [ASA+12]. Moreover, a developer might also be interested in controlling when to (de)activate a certain cross-layer coordination in order to analyze the performance of the distributed system with and without the running coordination on specific nodes. Both of these examples highlight the need for a remote (re-)configuration and control of cross-layer coordination algorithms at runtime which is not available to the best of our knowledge.

Third, the **analysis** of the effects while running and after finishing the test. The latter can be done by logging certain parameters which might require expert knowledge and much effort when accessing system parameters residing in the kernel space of the operating system. For remote analysis of nodes in the distributed system it also requires access to test nodes, which might be also very cumbersome. However, the analysis while running the experimentation is rather complex as it requires the ability to specify and access the parameter of interest and its level of logging granularity at runtime. For example, monitoring the received signal strength indicator (RSSI) of WiFi per second or per minute. Ideally, a developer should have the freedom to specify the set of specific parameters for logging, the granularity of reports and the node that should be included in the analysis.

We present an extension[1] to Crawler that tackles all these issues presented above by providing the following three key features:

- **Remote test automation** allows a developer to remotely describe whole experimentation setups, that is, start and termination of applications with their respective parameterization and cross-layer coordination algorithms, on a specified set of nodes without human interaction.
- **Remote configuration** allows to distribute cross-layer coordination algorithms to remote nodes, to control them (i.e., to add, remove, and modify

[1] The content of this and subsequent sections are partially based on the joint work with Oscar Punal, Florian Schmidt, Tobias Drüner and Klaus Wehrle published in "A Framework for Remote Automation, Configuration, and Monitoring of Real-World Experiments", 9th ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization (WINTECH'14), September 2014 [APS+14b]. Furthermore, the content is also partially based on Tobias Drüner's Master Thesis [Drü13].

coordination algorithms), and enables to conveniently access protocol and system information at runtime.

- **Remote monitoring** allows to conveniently specify and log a set of predefined parameters and states of cross-layer coordination algorithms from any node involved in the test and, if desired, to store them centrally. Moreover, live monitoring of parameters and the ability to adjust the level of report granularity are enabled at runtime.

The remainder of this Section is organized as follows: Section 6.3.1 presents an overview about the components in our architecture and their interplay. In Section 6.3.2, we describe our architecture in more detail and explain how we achieve the remote evaluation of cross-layer coordination algorithms. Implementation details are provided in Section 6.3.3. We evaluate our architecture in Section 6.3.4 and discuss related work in Section 6.3.5. Finally, we describe future improvements in Section 6.3.6, before concluding the remote evaluation enhancements of Crawler in Section 6.3.7.

6.3.1 Design Overview

The main goal of our extension to Crawler is to simplify the testing process of cross-layer coordination algorithms in distributed wireless networks. Through our extensions to the interfaces of Crawler, developers shall be able to conveniently perform real-world experiments including cross-layer coordination algorithms and monitoring their behavior without physically interact with nodes that are part of the experimentation. This section provides an overview of the design of our extensions to Crawler and the used components.

In order to enable developers to centrally and remotely automate, control and monitor their experiments, we opted for a client-sever architecture as depicted in Figure 6.1. The client is the central node that controls several servers running on different devices that are being part of the experiment.

The initial step to conduct a real-world experiment of a distributed system scenario including cross-layer coordination algorithms is done by writing a configuration file as indicated by ① in Figure 6.1. The configuration includes the required parameters for the entire experimentation. In particular, it includes information about the devices that are being part of the experimentation, the programs that should run on the devices including cross-layer coordination algorithms, the schedule of the programs and information about parameters that should be logged. Subsequently, the client parses the configuration and extracts the necessary tasks. Each task consists of its execution time, target device and instruction (e.g., to run a specific program).

At the scheduled execution time, the client sends the specific instruction to the listed devices via the network, see ②. Simply put, the client gives instructions about what should be done, when and on which device. The server is the counter part that receives these instructions and realizes them. Only the servers need to run Crawler.

To provide cross-layer coordination algorithms to a system, Crawler so far offered only a system-wide interface. By enhancing this interface, we enabled Crawler to

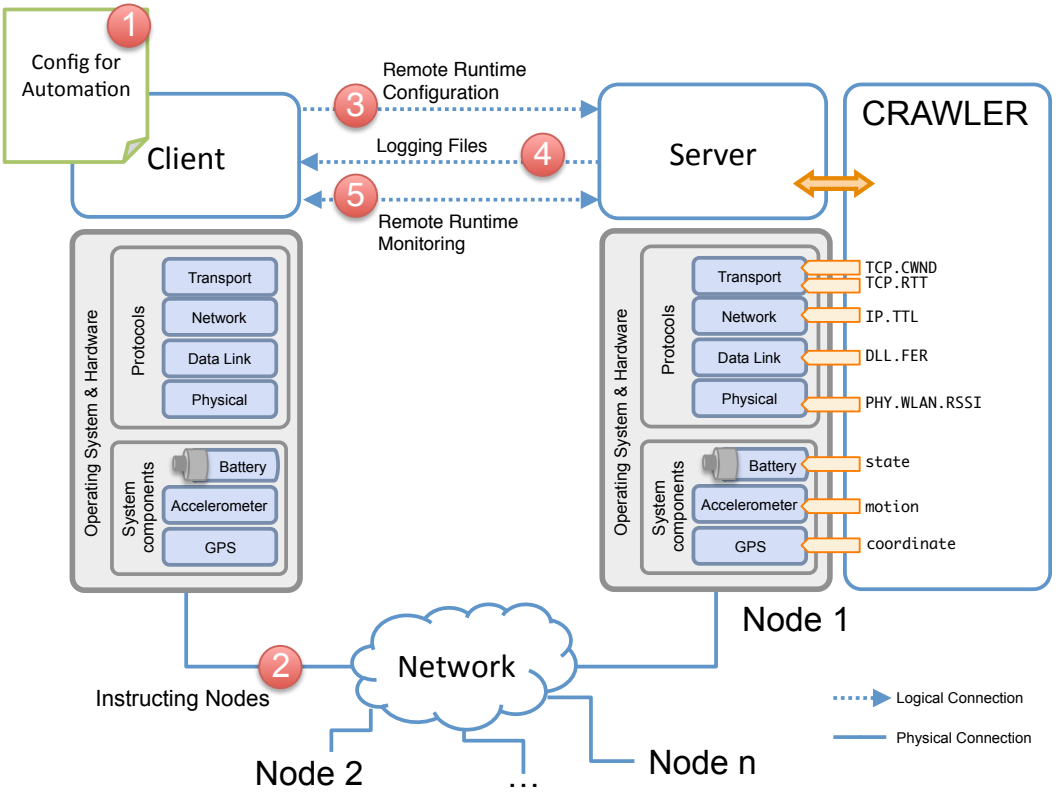


Figure 6.1 Conceptual view for remote automation, configuration and control.

distribute cross-layer coordination algorithms defined by our central client to other nodes in a network. Feeding cross-layer coordination algorithms into the system is not only easily possible at the preparation phase but also while running the experimentation which opens new possibilities to control the servers remotely while running, indicated by ③ in Figure 6.1.

The configuration file also contains information about which system parameters to log on the servers and their type of sorting. At the end of the experiment, the data is automatically aggregated at the client for easy evaluation, see ④.

Another advantage of CRAWLER is its live monitoring feature which conveniently allows to monitor a set of variables in running applications, protocols and system components at runtime. We extended this feature to allow remote access, see ⑤. This enables live monitoring of variables during the experiment, without having to wait for the aggregation of log files at the conclusion of the experiment.

To further support the experimentation, we created a graphical front-end, which (when enabled in the configuration) allows us to interactively control our three key features during experimentation.

6.3.2 Architectural Details

This section provides more details about how we realized the three key features of remote automation, configuration and monitoring and describe how we incorporated them into our interactive front-end.

6.3.2.1 Remote Automation

When many nodes are involved in testing, a prerequisite is the preparation of the involved nodes with the appropriate software and configuration. For instance, some of the nodes might have different IP addresses, traffic patterns, applications and cross-layer coordination algorithms. After running the tests, log files have to be analyzed on each node to understand the interplay of the programs. Moreover, due to the volatile nature of the wireless medium, it is necessary to repeat the test several times till solid and credible results are collected. Ideally, the adjustment, the repetition of the whole test and the collection of log files should be very convenient.

We propose a configuration which allows to centrally and conveniently execute whole test setups with different settings. The configuration includes the necessary instructions to automate the execution of desired settings including applications, cross-layer coordination algorithms and further helper programs. Note that convenient repeatability of the test is given by re-executing the configuration which can also be adjusted on each run. An example configuration is shown in Listing 6.1 which we later also use in the evaluation. The ready-to-use configuration is subdivided into sections by using the keywords [general], [crosslayerremote], [schedule] and [log].

```
[general]
interactive_mode:True

[crosslayerremote]
crosslayer_server_count: 3
crawler_default_path: /home/crawler/
crawler_default_port: 12345
[crosslayerremoteserver1]
alias: node09
ip_address: 137.226.59.183
[crosslayerremoteserver2]
alias: node05
ip_address: 137.226.59.21
[crosslayerremoteserver3]
alias: node04
ip_address: 137.226.59.50

[schedule]
logclear: [([1,2,3], 1, "rm logfile.txt")]
daemonclear: [([1,2,3], 1, "rm daemon.txt")]
crawler: [([1,2,3], 3, "load_crawler cfg80211Layer ethernetLayer")]
iwconfig_node09: [('node09', 3, "sudo iwconfig wlan1 txpower 8;")]
iwconfig_node05: [('node05', 3, "sudo iwconfig wlan0 txpower 8;")]
iwconfig_node04: [('node04', 3, "sudo iwconfig wlan1 txpower 4;")]

startjammingdet9:[('node09', 4, "crawlerapp mainJammingDetection
    +s: complexDetection -i wlan1 -c 172.16.0.6 -s 172.16.0.255
    -t 100 -o 1 -m 5")]
startjammingdet5:[('node05', 4, "crawlerapp mainJammingDetection
    +s: complexDetection          -c 172.16.0.5 -s 172.16.0.255
    -t 100 -o 1 -m 5")]
startjammingdet4:[('node04', 4, "crawlerapp mainJammingDetection
    +s: complexDetection -i wlan1 -c 172.16.0.4 -s 172.16.0.255
    -t 100 -o 1 -m 5")]
rssi:[(1,5,"monitorVariableApp wlan0.qual.rssi.max > rssi.txt")]
stopjammingdet:[([1,2,3], 124,"killall mainJammingDetection")]
```

```
37 crawler_finish:[([1,2,3],130,"load_crawler cfg80211Layer
38     ethernetLayer")]
39
40 [log]
41 log_crosslayer_remote_server: True
42 log_daemon: True
43 jammingdetLog: [([1,2,3], "logfile.txt")]
44 rssilog:[(1,rssi.txt)]
```

Listing 6.1 Setup of the configuration for our jamming detection scenario.

As the name hints at, the section `[general]` contains general information which do not directly belong to a specific test but are rather a tool configuration. For instance, the boolean variable `interactive_mode` indicates the (de)activation of the interactive mode as shown in line 2. More details about the interactive mode will be explained later in Section 6.3.2.4. Another example (which is not shown) is the variable `time_limit` which can be adjusted to define the amount of time that the specified servers are allowed to run after starting all their running instructions such as applications or cross-layer coordination algorithms. When that time expires, all processes are killed on the specified servers. However, the information in the general section are optional and can be omitted in contrast to the following sections which are mandatory.

The `[crosslayerremote]` section includes information about the involved remote servers in the test. For example, in line five `crosslayer_server_count` defines the number of remote servers used in the test. In line six the optional parameter `crawler_default_path` allows to set a path to a directory which should be used by the servers to start Crawler. Line seven gives the port number which is used by Crawler running on the servers for incoming requests for a remote connection. In the subsequent lines the details about the three servers are configured. To differentiate between several servers, we again used the bracket notation followed by the number of the specific server. For instance, in line eight we configure server one which we assign an alias, here `node09`, followed by the IP-address assigned in line ten. The number of the server in the bracket notation, here 1, is important for later specification such as scheduling tasks at specific servers or for collecting log files. The IP-address is mandatory for remote control of the servers. The alias is another option for more human readable naming of servers in the configuration. The two remaining servers are similarly configured (cf. line 11–16 in Listing 6.1).

The main part of the configuration is the `[schedule]` section which determines when to execute which instruction on which server. To exactly reflect this, we have used a three tuple notation `(<server>, <time>, <command>)`; the first index indicates the the server, the second index the relative time in seconds after the test has started, and the third index indicates the instruction that should be executed. Typically, the command is executed in a shell on the specified server except commands that consist of keywords such as `load_crawler` and `crawlerapp` which trigger special functionalities. Note that the experimentation starting time between all servers is synchronized[2], i.e., when starting the test on a specific server its time is set once to a common time zero. To give an example about our tuple notation, let us consider line 22 in Listing 6.1, where we use in the first index the alias of the node (here

[2]Note that this is not a continuous synchronization, only an initial signal is sent which might lead to time drift among the servers. Nonetheless this allows a causal order of the commands.

node09) to specify the server; the second index indicates the time of execution of a specific command (here 3 seconds right after starting the experimentation); and the third index indicates the command by itself that should be executed (here the manipulation of transmission power txpower which is modified using the wireless tools iwconfig). The subsequent lines (23–24) do similar modifications to the other remaining servers. To avoid having to execute the same instructions for different servers over and over again, we provide a bracket notation (inspired by Python arrays) to address all servers at once such as shown in line 21. In particular, the three servers are instructed to load Crawler (using the special keyword load_crawler) with some of its kernel modules after three seconds. In lines 19–20 we also used this notation to first clean all log files before starting to collect data with Crawler. However, the main instruction in this configuration is the start of a jamming detection application on each server. For instance, in line 26 on server one (i.e., with alias node09) at time four the Crawler application for jamming is started with further parameters such a jamming detection strategy (complexDetection) which is essentially a cross-layer coordination. Similarly, the remaining two nodes are configured but with different parameterization. It is also possible to specifically instruct the monitoring of variables for logging and later evaluation as indicated in line 35 where we monitor at node one the maximum radio signal strength indicator measured for all connected stations. However, in lines 36–37 the instructions to stop the whole test are given.

To deliver parameters that should be logged on the servers and delivered to a central place, i.e., to the client, the [Log] section is used. For example, line 43 indicates that all log files from all three servers should be collected. The specific logging of the signal strength is delivered to the client with the instruction as indicated in line 44. Results for this particular configuration will be presented later in our evaluation.

So far, we have only seen a single experiment without changing parameters. In the following we discuss how to support several test runs consisting of small variations.

Main and Custom Configuration

The configuration presented so far only considered one single test run. However, we can support the user in conducting test runs with different parameter changes. Thus, to relieve the user of our architecture from writing an additional configuration file when only minor changes are required, we distinguish again between main and custom configuration files. The main configuration file describes the major test setup while the custom configuration only includes the differences. The custom configuration describes the relative changes and overwrites the respective values of the main configuration. For example, if we want to have an earlier scheduling time (two instead of three) for loading Crawler as shown line 21, we only have to add a single modified line in the custom configuration in the respective section ([schedule]) such as `crawler: [([1,2,3], 2, "load_crawler cfg80211Layer ethernetLayer")]`. Our parser finds the respective variable and overwrites the main configuration.

6.3.2.2 Remote Configuration

CRAWLER already allows to add, remove and modify cross-layer coordination algorithms at runtime by using the keywords `load(rule)`, `unload(rule)` or `replace(oldrule, newrule)` in its configuration language. For more details about the runtime reconfigurability, we refer the reader to Section 3.4.4. However, applications that want to use these functionalities have to register beforehand at CRAWLER (running as a daemon) using a shared library. From then on, the shared library facilitates the system-wide signaling of states, i.e., the exchange of information between applications, protocols and system components. This only requires an application to include the library's header file `crawler.h` (which provides also the callback functions to read or write to the application variables) and link against the library. The interaction between CRAWLER and applications is performed by the shared library itself. So far, this functionality was only possible system-wide and remote access from another system was not feasible. Therefore, we have enhanced the shared library to allow a remote use. To understand how an application can (remotely) use the shared library, we give an example implementation in Listing 6.2.

```
1 #include "crawler.h"
2 ...
3 int main() {
4   char *APPNAME = "MyApplication";
5   int libId   =createLibrary("UnixLibrary",   "UnixProtocolC"); //1.
6   int netLibId=createLibrary("NetworkLibrary","UnixProtocolC"); //2.
7   int sslLibId=createLibrary("OpenSSLLibrary","UnixProtocolC"); //3.
8
9   // callbacks for variable updates
10  callback_ops co = {&setValue,&getValue,&error};
11
12  //network library specific settings
13  network_library_extraData netExtra;
14  netExtra.serv_addr = "137.226.12.27"; //server running crawler
15  netExtra.port      = 12345;
16
17  // open ssl library specific settings
18  openssl_library_extraData sslExtra;
19  sslExtra.serv_addr = "137.226.12.28"; //server running crawler
20  sslExtra.port      = 65432;
21  sslExtra.cert_file = "cert/client.pem"; //client certificate
22  sslExtra.ca_file   = "cert/rootcert.pem";
23  sslExtra.ca_dir    = NULL;
24
25  initLibrary(libId,co,APPNAME,strlen(APPNAME),NULL);        //1.
26  initLibrary(netLibId,co,APPNAME,strlen(APPNAME),&netExtra);//2.
27  initLibrary(sslLibId,co,APPNAME,strlen(APPNAME),&sslExtra);//3.
28
29  addChains(libId,config1,strlen(config1));    // 1.local
30  addChains(netLibId,config2,strlen(config2)); // 2.network
31  addChains(sslLibId,config3,strlen(config3)); // 3.openssl
32 }
```

Listing 6.2 To allow the access to system parameters and feed cross-layer coordination algorithms into the system, an application can utilize a UnixLibrary for local use (as indicated by 1), a NetworkLibrary for remote use (indicated by 2) and an OpenSSLLibrary for secure remote use (indicated by 3).

First, the header file for the shared library is included in line 1. In the main method of the application, first the specific library is created. For show-case reasons, in the listing we create three different libraries. In line 4 we declare a shared library for system-wide access, in line 5 a library for remote access and finally a library for secure remote access that uses OpenSSL for secure communication. For details about the secure communication we refer the reader to [Drü13].

Afterwards, in line 10 we declare two callback functions, where one allows the shared library to read a variable from the application and the other one to allow to write into the application variable. The callback functions have to be specifically implemented but we omitted it here for space reasons. An example implementation of callback functions is given in Listing 3.4.

Subsequently, we have additional settings for the specific libraries in lines 12-15 (for the network library) and 17-23 (for the openssl library). For example, the application running on the client needs the connection endpoint (IP address and port) if a remote access to the server is desired which is indicated in lines 14 and 15. Afterwards the specific libraries are registered using the callback functions. For instance, line 25 registers the system-wide (local) library, line 26 the library for network-wide access and line 27 the secure network-wide access.

Using the shared library, an application is able to access the system, even remotely when using the network or the OpenSSL library. So far, the cross-layer coordination algorithms that realize the access in the system are missing. To feed the cross-layer coordination algorithms into the system, the shared library offers the `addChains` method. For instance, for locally adding cross-layer coordination algorithms into the system, in line 25 we used the `addChains` method with the corresponding library (`libId`) and the configuration (contains the cross-layer coordination) being the attributes. Similarly, line 26 and 27 show how to remotely and securely add different cross-layer coordination algorithms into devices. From now on the cross-layer coordination is injected into the system and Crawler is able to (remotely) access the variables shared by that particular (remotely running) application in specified intervals or triggered by the cross-layer coordination.

In the following we present an application that demonstrates the simplicity and power of the shared library.

Helper Application for Remote Configuration

In order to further simplify the process of adding cross-layer coordination algorithms into remote devices, we implemented a helper application which we refer to as `addChainsApp`. This helper application allows to easily add cross-layer coordination algorithms into a system. The cross-layer coordination has to be provided as a configuration file using Crawler's declarative language which allows to describe the desired coordination at a high level of abstraction. For example, Listing 6.3 presents a cross-layer coordination to switch TCP's congestion control algorithm at runtime while keeping the internal values such as the congestion window without resetting them. To achieve a focus on the features of the helper application, we simplified this example which is an excerpt of the full configuration as presented in Section 4.2. In the first line of the configuration the average over 10 values of the radio signal

strength indicator (RSSI) is calculated. In line three this average value is compared to a specific threshold. If the threshold is exceeded as expressed in line three, the congestion control algorithm is switched in line four.

```
1 rssiAvg:avg(history(currentRssi:get("wlan0.qual.rssi"),10))
2 rssiIsLow:less(rssiAvg,60)
3 cwndAlg:if(rssiIsLow,"westwood","vegas")
4 setCwndAlg:set("tcp.cwnd", cwndAlg)
5 load(setCwndAlg)
```

Listing 6.3 A simplified version of a cross-layer coordination to switch TCP's congestion control algorithm.

The keyword `load` in line five instructs CRAWLER to automatically load the specific rule into the system. In cases of nested rules, the dependencies are automatically resolved and all necessary rules are all loaded into the system. A removal of rules or their replacement work similarly. In particular, by using the keywords `unload` or `replace` a rule is removed or exchanged with another, respectively. This only requires that these rules which are identified by their names, for instance `setCwndAlg`, are available in the system. After finalizing the configuration, we can now utilize the `addChainsApp` application to remotely (re)configure cross-layer coordination algorithms in a remote system. For example, the following command in the console allows to provide the CRAWLER rules described in the file `myconfig.cfg` (e.g., our TCP congestion control coordination) to the server reachable via the IP-address 192.168.0.5 and port 12345.

```
1 $>./addChainsApp --host 192.168.0.5 --port 12345
2     --chain myconfig.cfg
```

On calling this command in the console, the `addChainsApp` application first remotely registers at the server running on the specified remote node which is indicated by ① in Figure 6.2. Subsequently, the `addChainsApp` application sends the cross-layer coordination given in the configuration to the remote system, cf. ② in Figure 6.2. Afterwards, CRAWLER takes care to feed and realize the cross-layer coordination in the system. In particular, based on the configuration, functional units (FUs) are composed to realize the desired cross-layer coordination, cf. ③ in Figure 6.2. For details about how the FUs are composed based on a given configuration, we refer the reader to Section 3.4.1. Instead of only allowing a file that includes CRAWLER rules, we also support to directly inject rules in their string representation as an argument on calling the application. But for clarity and convenience reasons we suggest the usage of pre-edited configurations to avoid programming flaws.

To summarize, by using the shared library it is easily possible to feed cross-layer coordination algorithms into a remote system. Our helper application `addChains` further simplifies this process. In the following we present the necessary details about how to remotely monitor system parameters in general as well as the added cross-layer coordination algorithms.

6.3.2.3 Remote Monitoring

Access to system information, i.e., to protocols and system components such as sensors, is relatively difficult due to OS limitations. However, the access to system

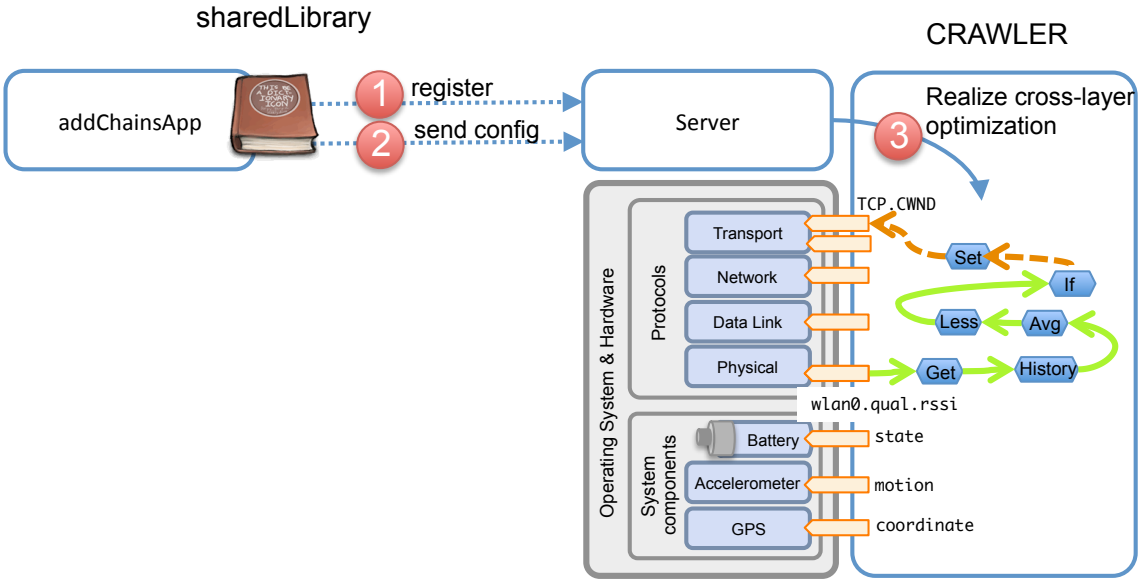


Figure 6.2 The `addChainsApp` application helps a developer to remotely configure running cross-layer coordination algorithms. After registration (indicated by ①), a configuration that contains the instruction to add, remove and replace a set of specified cross-layer coordination algorithms is sent to the remote system (indicated by ②). On reception, CRAWLER automatically realizes the given instructions (as indicted by ③).

information significantly helps to understand and debug the experiment and the interplay between algorithms and other effects such as the unpredictable wireless channel. Moreover, it will also help to measure and analyze the benefits of the envisioned algorithms. While CRAWLER provides the necessary interfaces to locally access these information, a remote access was initially not enabled. With the improvements to the shared library as presented above, we also lay out the foundation for remote monitoring. In the following we present the details about how to monitor system information and cross-layer coordination algorithms.

Monitoring System Variables

The extensions to the shared library allow us to remotely feed cross-layer coordination algorithms into the system. The same feature also allows us to access the desired system information. In particular, we generate a configuration that specifies a cross-layer coordination to read a certain variable in the system. Remember that variables are accessed by so-called stubs which provide read and write access to protocol and system information. They act as a glue element between the cross-layer coordination algorithms and the OS. Stubs offer a common interface and a very find-grained access to system information. Thus, to access the desired protocol or system variable, stubs need fully qualified, i.e., unique and hierarchical, names. We have implemented the helper application `monitorVariableApp` that allows to conveniently monitor stubs by specifying its fully qualified name and node of interest. For example, by calling the following command in the console it is possible to monitor the RSSI (specified by using its fully qualified name `wlan0.qual.rssi.avg`) every 100 milliseconds (update intervals) on the remote server with the IP address 192.168.0.5.

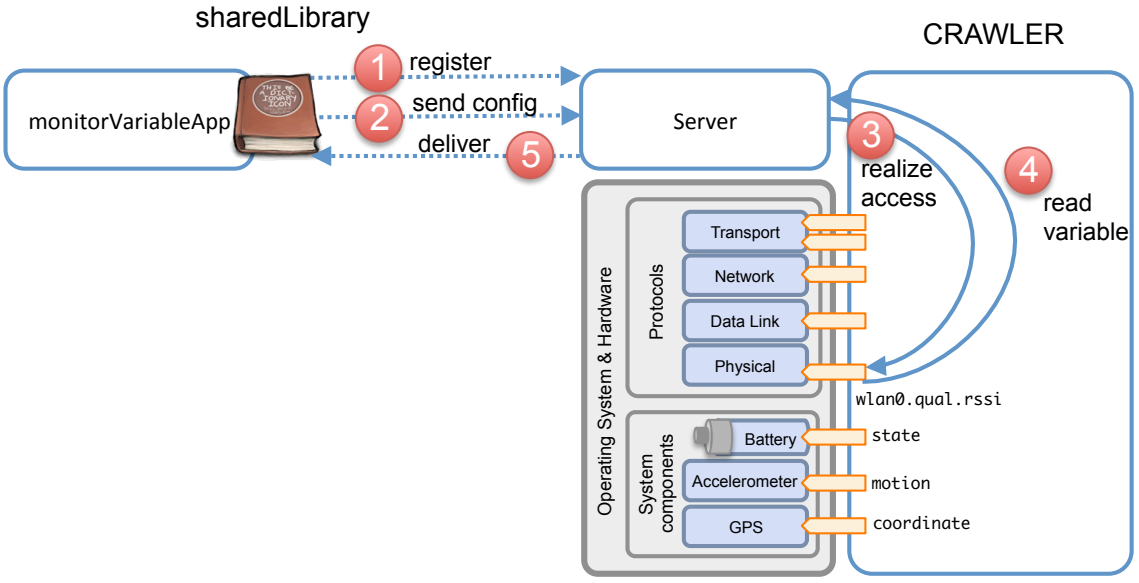


Figure 6.3 The `monitorVariableApp` application helps a developer to remotely monitor system variables. After registration (cf. step ①), a configuration that contains the instructions to monitor a set of variables is send to the server (cf. step ②). On reception, CRAWLER automatically realizes the respective accessors to the variables (cf. step ③). Subsequently, the specified variables are continuously monitored and delivered to the server (cf. step ④) which then delivers them to the client (cf. step ⑤).

```
1 $>./monitorVariableApp --host 192.168.0.5 --port 12345
2     --variablename "wlan0.qual.rssi.avg" --interval 100
```

On calling this command in the console, the `monitorVariableApp` application first remotely registers at the server running on the specified remote node as indicated by step ① in Figure 6.3. In a next step, based on the given name of the variable, a CRAWLER configuration is created that specifies the access to the desired stub. This configuration is then sent to the remote system (cf. step ②). Based on the given configuration, CRAWLER realizes the cross-layer coordination to read the specified variable (cf. step ③). Finally, the variable is continuously monitored and, based on the specified update intervals, provided to the server (cf. step ④) which then further delivers the monitored values to the `monitorVariableApp` application (cf. step ⑤).

As stubs are identified by using their fully qualified names, the `monitorVariableApp` only allows to monitor variables or rather stubs that are supported by CRAWLER. A list of stubs is given in Appendix B. However, adding further variables to monitor is supported by CRAWLER as presented in Section 3.4.3.2.

In the following, we present how we used the shared library to also monitor cross-layer coordination algorithms.

Monitoring Cross-Layer Coordination Algorithms

To realize a cross-layer coordination, information such as from protocols and system components might need to be aggregated, processed and exchanged. Understanding all the details of an interaction necessitates the ability to debug or rather monitor

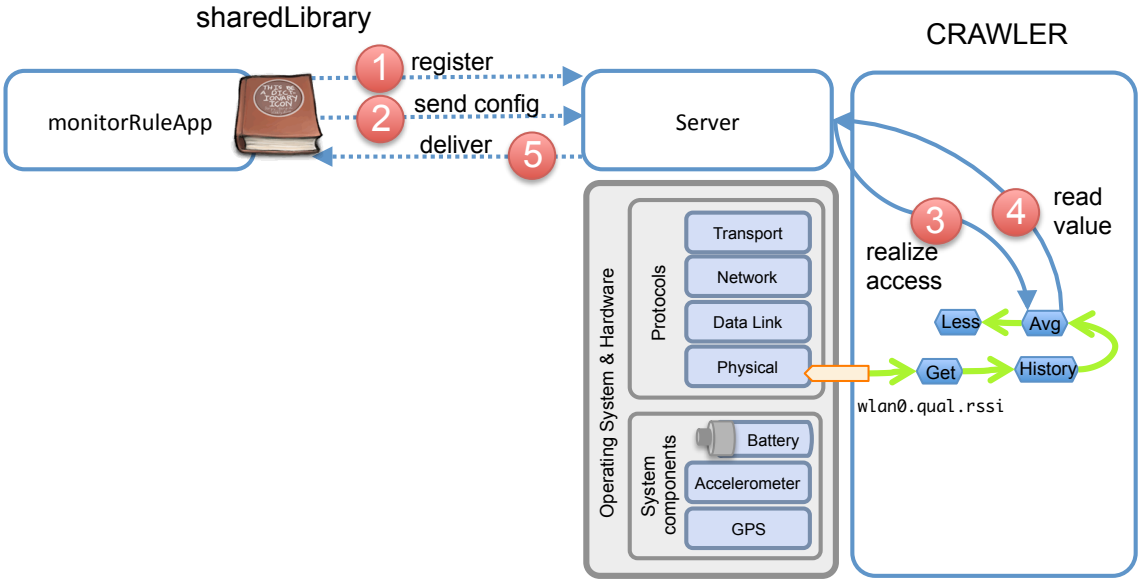


Figure 6.4 The `monitorRuleApp` application helps a developer to remotely monitor cross-layer coordination algorithms. After registration (cf. step ①), a configuration that contains the instructions to monitor a set of functional units (composed to realize a cross-layer coordination) is send (cf. step ②). On reception, CRAWLER automatically realizes the respective accessors to monitor the specified functional units (cf. step ③). Subsequently, the functional unit states are continuously monitored and delivered to the server (cf. step ④) which then delivers them to the client (cf. step ⑤).

information while being processed, aggregated and exchanged. With the presented `monitorVariableApp` application, we are only able to access parameters of the system such as the `wlan0.qual.rssi`. In addition to monitoring system variables, it would be also interesting to monitor intermediate states of the cross-layer coordination, for example, to monitor the average values of the RSSI in our coordination to modify TCP's congestion control algorithm (which we added in the previous section with our `addChainsApp` application). As CRAWLER realizes such cross-layer coordination algorithms by composing functional units (FUs), monitoring FUs already provides the essential monitoring ability to understand the behavior of the coordination algorithms. For this purpose, we have implemented a dedicated application which we refer to as `monitorRuleApp` which works similar to the `monitorVariableApp` application but allows to monitor a specified FU.

For example, by calling the following command in the console it is possible to monitor the average RSSI value every 500 milliseconds (update interval) on the remote server with the IP address 192.168.0.5.

```
1 $>./monitorRuleApp --host 192.168.0.5 --port 12345
2     --rulename "rssiAvg" --interval 500
```

Similar to the `monitorVariableApp`, on calling this command the `monitorRuleApp` first remotely registers at the server running on the specified remote node, cf. ① in Figure 6.4. But in contrast to `monitorVariableApp`, the `monitorRuleApp` accepts an identifier of a rule in CRAWLER instead of a fully qualified name of a stub. Based on the given identifier, a CRAWLER configuration is created that specifies the access to the desired FU. This configuration is then sent to the remote system (cf. step ②).

Upon receiving the configuration, CRAWLER realizes the access to the specified FU, for instance, the access to the FU that provides the average RSSI values (cf. step ③). From then on, the specific FU is continuously monitored and based on the specified update intervals provided to the server (cf. step ④). Finally, these values are further delivered to the `monitorRuleApp` (cf. step ⑤).

To summarize, by using the `addChainsApp`, we enabled to conveniently add cross-layer coordination algorithms into remote systems. While the application `monitorVariableApp` allows to remotely monitor a set of variables in a specified system, the application `monitorRuleApp` allows to remotely monitor internal states of cross-layer coordination algorithms. These helper applications can be conveniently used within any other software, while the applications `monitorVariableApp` and `monitorRuleApp` only requires that the software using them is able to process and pipe the standard input and output. In the following, we demonstrate the benefits of our three helper applications (`addChainsApp`, `monitorVariableApp` and `monitorRuleApp`) in a graphical and interactive front-end that further supports the convenience during experimentation.

6.3.2.4 Graphical and Interactive Front-End

So far, we have presented applications that allowed us (i) to control cross-layer coordination algorithms (`by using addChainsApp`), (ii) to monitor variables in the system (by using `monitorVariableApp`), and (iii) to monitor intermediate variables within the cross-layer coordination algorithms (by using `monitorRuleApp`). We refer to these applications as helper applications. Our three helper applications simplify the process of experimentation with cross-layer coordination algorithms.

To further support the developer while experimenting, we integrated these helper applications into a graphical front-end. Figure 6.5 shows the sequence diagram to provide a better understanding about the overall interplay within our architecture. In an initial step, our toolchain is started by using the following command in the console.

```
1 $>python ia_remote_eval_client.py --configs auto_config.cfg
```

Subsequently, the configfile `auto_config.cfg` is parsed, which contains all necessary automation information for central execution of the whole experiment. Based on the specifications within the configuration (e.g., as given in Listing 6.1), the connections to servers that are being part of the experiment are established. This only requires that on booting, the server is started on the remote devices which can, for instance, be configured in the autostart of the device. The configuration also includes information when to execute which command on the servers. These tasks are extracted and a schedule is determined.

The configuration also contains the parameter to enable the interactive mode. In case of a disabled interactive mode, only the experimentation is started and the tasks as determined in the schedule are executed till the end time of the experimentation which is also given in the configuration. In case of an enabled interactive mode, in addition to the experimentation also the interactive front-end is started.

By utilizing the helper applications, the features of the interactive mode are threefold. First, the `addchains` command makes use of the helper application `addChainsApp` and allows to add, remove and modify cross-layer coordination algorithms at

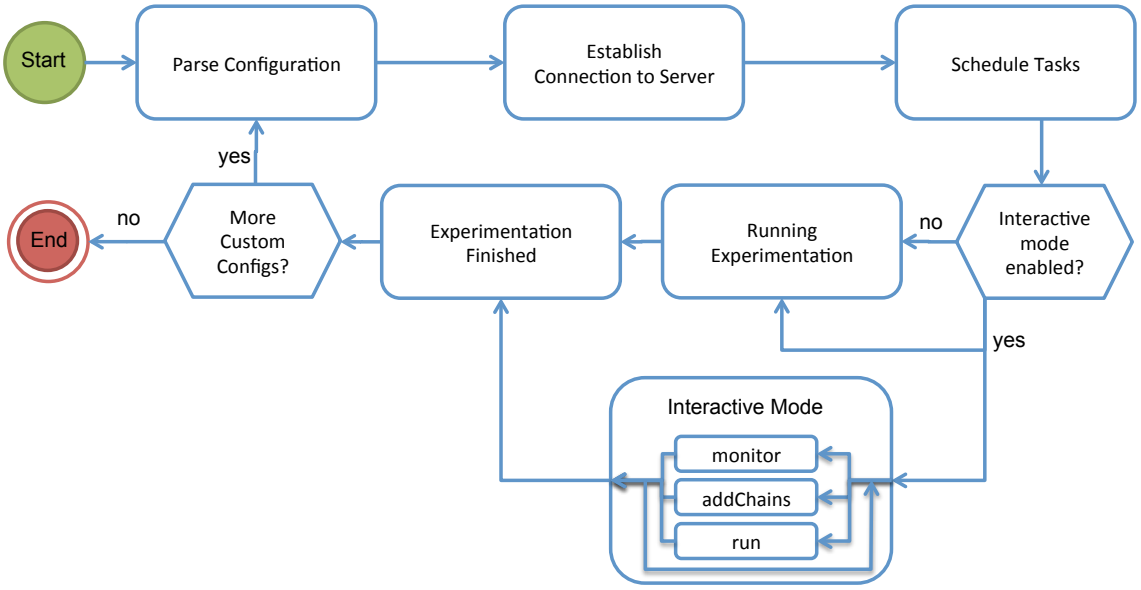


Figure 6.5 Sequence diagram of the interactive front-end. After starting our tool, from a configuration the necessary automation information is extracted and executed. If the interactive mode is enabled (in configuration), the front-end appears and allows the use of three commands during the experiment: `monitor` to monitor system variables and cross-layer coordination algorithms, `addChains` to control the set of running cross-layer coordination algorithms, and `run` which allows to control third party software. After finishing the experiment, if more custom configurations are available the procedure is started again, otherwise results are delivered to the client.

runtime. Second, by using the command `monitor` the helper applications `monitorVariableApp` and `monitorRuleApp` are used in the back-end. Remember that while the former allows to specify a set of variables for monitoring from applications, protocol and system components, the latter allows to monitor the cross-layer coordination algorithms and intermediate values. Third, the `run` command allows to control any program remotely or rather allows to intervene into a test such as starting or stopping programs, for instance, when misbehavior is observed. Another example could be to start further helper programs such as tcpdump or netem for better analyzes. This feature allows the same power as opening a shell and executing remote programs.

After the experiment is finished, our tool chain checks for further custom configurations. If more configurations are available, the whole procedure is started again, otherwise all the results from the servers are zipped and delivered to the client. The central logging at a central place improves the convenience of experimentation. In a nutshell, our three key features together combined in a single graphical environment enable an easy-to-use and a holistic solution for the experimenter.

Figure 6.6 shows a generic layout of the interactive front-end. The layout consist mainly of three areas: (i) an area that allows to provides the commands, (ii) an area for given additional information regarding the commands or providing feedback information from our toolchain, and (iii) the data that is being monitored. In the latter case, the area is subdivided into fields reflecting each parameter type such as radio signal strength. The field is labeled at the top with the node name and the name of the monitored variable, for instance, `node09 wlan1.cfg80211.signal.avg`.

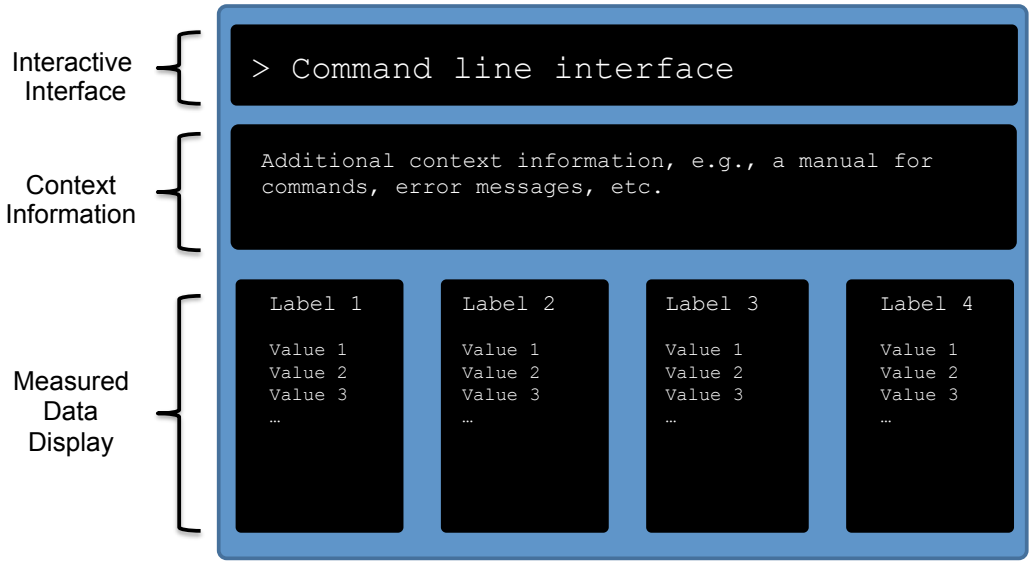


Figure 6.6 A sketch of the interface layout

Based on the screen size a certain amount of history values gathered from the specific node are listed below the label. At runtime it is possible with the interactive front-end to monitor further parameters. In this case, fields for the specified variable are added or removed at runtime.

6.3.3 Implementation

The implementation mainly consists of two separate parts:

The first part is the extensions of Crawler where we enhanced the application support interface in order to make it accessible over the network, i.e., without direct human interaction. For this it was necessary to add further functionality (remote configuration and monitoring) into the shared library within the application support subcomponent of Crawler. These enhancements are implemented in C++. We have also enabled a secure remote configuration using OpenSSL.

Second, the remote automation and the interactive remote front-end are implemented in Python. We opted for Python as it allows scripting in a high level, which makes it very suitable as a test automation language. The graphical front-end for the helper applications uses the library curses [ncu13]. However, the communication between the remote client and the remote server are realized with PYthon Remote Objects (PYRO). Connections are established to a remote host by using the IP address for each remote server. A client connects to remote servers using the PYRO interface and can directly use objects of the corresponding remote server implementation.

For more details about the implementation, we refer the reader to [Drü13].

6.3.4 Evaluation

With our evaluation we target at demonstrating our three key features (i) remote automation, (ii) remote configuration, and (iii) remote monitoring. To demonstrate

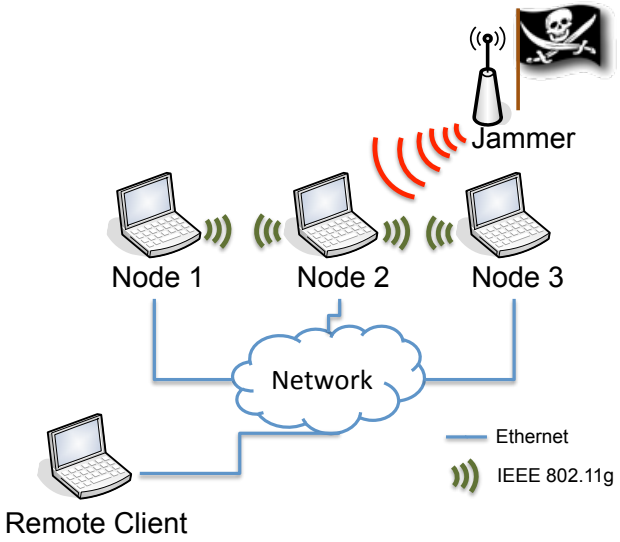


Figure 6.7 Reference scenario used to evaluate our three key features consists of a three nodes and a remote client. The remote client controls the nodes involved in the experiment and monitors the impact of several metrics in the presence of jamming.

these features we performed an experimentation in the field of jamming detection. By using CRAWLER we built a flexible jamming detection and reaction framework to dynamically load and unload jamming detection and reaction strategies. By using command line instructions we are able to select the strategies which are then inserted as cross-layer coordination algorithms into the system via CRAWLER. This required configuration and manual effort for each device before starting the experimentation. While running the experimentation we were able to monitor the reaction of metrics and the decision of our jamming detection strategy in the presence of jamming. This again was only possible locally on a specific device requiring physical interaction and effort. The evaluation results for this scenario without our remote evaluation feature are presented in Section 4.5. We repeated this experimentation with our extension to CRAWLER to centrally and remotely automate, configure and monitor the experimentation.

We conducted the experimentation in a small office room located in the ComSys institute at the RWTH Aachen University. We used three Linux PCs equipped with an 802.11g/n Atheros WLAN card running the ath9k driver [ath]. We let the three PCs build an ad-hoc network and continuously exchange messages on channel 11 within the 2.4 GHz band. Details about the original test setup are given in Section 4.5.2. As our scenario is a wireless experimentation and any wireless communication with the testing nodes may influence the experimentation results, we used Ethernet for our remote features to interact with the nodes. The only device that was not remotely controllable during the experimentation was the jammer, as it is implemented on the Wireless Open-Access Research Platform (WARP) [KCH+08], while CRAWLER is specifically designed for x86-based systems. A sketch of our reference scenario is shown in Figure 6.7.

In the following we show how we benefit from using the three key features during our experimentation beginning with the automation key feature.

6.3.4.1 Evaluating Remote Automation

The preparation of the experiment first requires to open various terminals on each communicating node. In one terminal the kernel modules of CRAWLER are loaded and it later displays kernel logs. A second terminal is used for adjusting the transmission power of the nodes (via iwconfig), starting CRAWLER, and printing log information of the CRAWLER daemon. In a third terminal the jamming detection framework is launched. Once the experiment is finished, the daemon running in second terminal and application running in the third terminal are closed and the respective log files (CRAWLER and kernel) are manually collected from each node. The tasks associated with the preparation and collection of results require several minutes of manual work. Clearly, such an approach does not scale well due to two main reasons. First, the required time increases linearly with the number of devices. Second, multiple experiment runs are generally required for a complete parameter study and to obtain statistical confidence in the results. However, once the short and intuitive configuration is specified as illustrated in Listing 6.1 (which only requires 44 lines of code for the whole experimentation), the above-mentioned steps with our framework spanned only few seconds. When running this configuration, we observed that the experimentation is conducted as specified and the log files are centrally stored at the client.

In the following, we present our two other key features and highlight their benefits during experimentation.

6.3.4.2 Evaluating Remote Configuration and Monitoring

A combined validation of remote configuration and monitoring is very suited as one functionality influences the system during experimentation and the other functionality helps to trace the influencing factors. Hence, in order to validate remote configuration, we show the proper working of the commands `addChains` and `run`. For validating the remote monitoring feature we need to show that we are able to monitor both (i) system variables and (ii) states of cross-layer coordination algorithms. To validate these features, we selected a set of instructions that were executed while simultaneously running a jamming detection experiment which are shown (in its order of execution) in Listing 6.4.

```
1 monitor node05 wlan1.cfg80211.signal.avg //monitoring a variable
2 addChains node05 rssi.cfg   //adding a crawler configuration
3 monitor node05 maxrssi  //monitoring a crawler rule
4 monitor node05 minrssi  //monitoring another crawler rule
5 run node04 iwconfig wlan1 txpower 20 //executing 3rd party programs
6 monitor stop node05 wlan1.cfg80211.signal.avg
7 monitor stop node05 minrssi
```

Listing 6.4 List of instructions that are conducted successively (i.e., each with having a gap of few seconds in-between) in the interactive mode.

With the first line we monitor on `node02` the average signal strength of messages originated at all neighboring nodes. The respective output of our front-end is depicted in Figure 6.8 which shows reported values (stable between −77 dBm and −78 dBm) for the specified variable. Note that the newly monitored values are at the top.

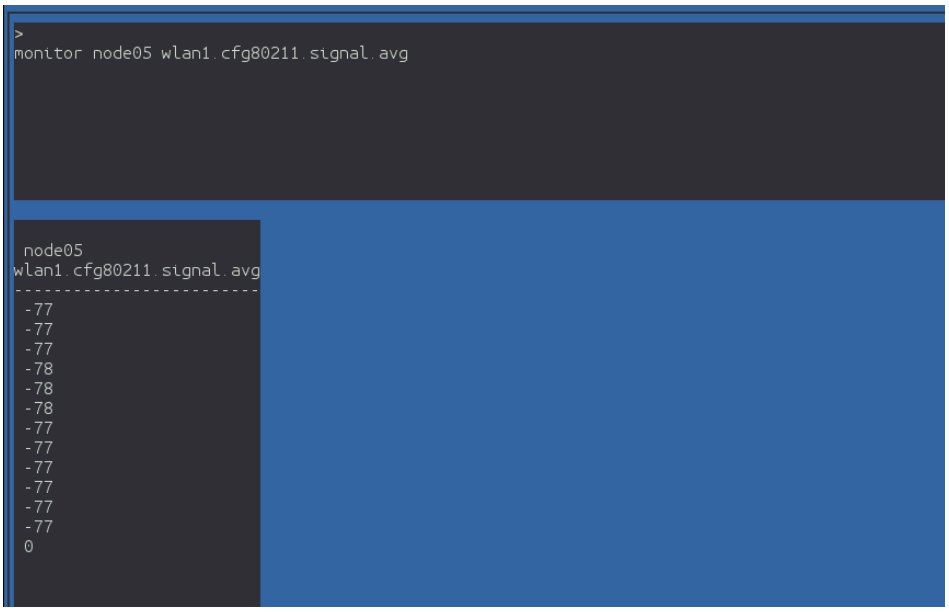


Figure 6.8 Monitoring a variable in the system is easily possible in our interactive mode. In this example, by using the keyword `monitor`, we monitor on `node05` the average signal strength of all connected nodes by using the stub `wlan1.cfg80211.signal.avg`. Afterwards, based on the screen size, the interactive mode displays for the specified variable and node, an amount of history values.

In Line two we add the CRAWLER configuration as shown in Listing 6.5, which includes two rules or cross-layer coordination algorithms. In particular, the instructions in Line 1 and 2 of this configuration compute, over a series of 20 collected values, the maximum and minimum of our monitored variable `wlan1.cfg80211.signal.avg`, respectively. In Line 3 and 4 we deliver the corresponding values, that is `maxrssi` and `minrssi`, to the application layer. The subsequent lines are notifications that determine the update intervals.

```
1 maxrssi:max(maxhistory:history(get("wlan1.cfg80211.signal.avg"),20))
2 minrssi:min(minhistory:history(get("wlan1.cfg80211.signal.avg"),20))
3 app_var1:set("application.mainJammingDetection.var1", maxrssi)
4 app_var2:set("application.mainJammingDetection.var2", minrssi)
5 timer:pollingtimer(250)
6 timer->maxhistory;
7 timer->minhistory;
```

Listing 6.5 CRAWLER config that we inserted into the system during experimentation.

Using the `addChains` command, we inserted these rules into the system. The validation of this step is easily possible with the instructions shown in line four and five of Listing 6.4 where we used the `monitor` command to monitor both rules. The output of the graphical front-end is depicted in Figure 6.9 where the middle column shows the `maxrssi` values (area is marked as ③ corresponding to the line in Listing 6.4) and the right column the `minrssi` values (area is marked as ④).

So far, we showed that using the `monitor` command we are able to monitor both variables in the system and cross-layer coordination algorithms. Next, we show that we are also able to run third party programs during experimentation. To validate that, we significantly increased the transmission power of the neighboring node `node03` by using the keyword `run` and the program `iwconfig` as shown in line

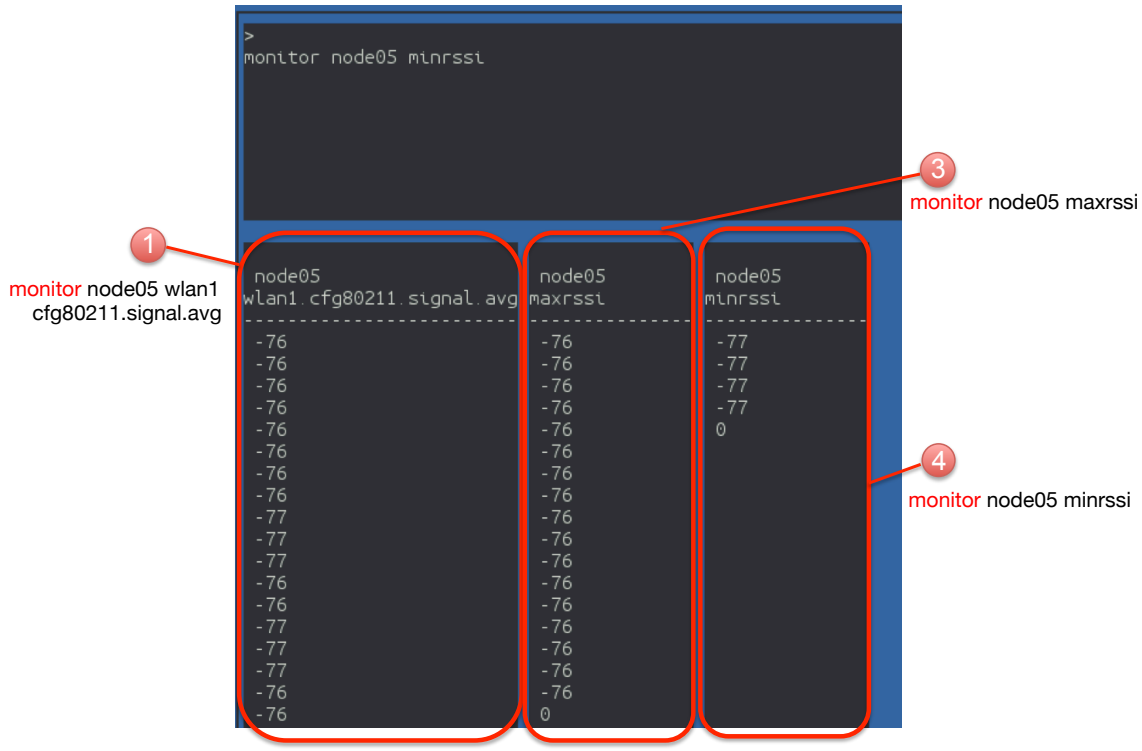


Figure 6.9 The interactive mode also allows to monitor cross-layer coordination algorithms in the system. After inserting coordination algorithms (in a previous step), monitoring them is possible by using again the keyword `monitor`. Particularly, we monitored on `node05` the `maxrssi` and `minrssi` rules as shown in the middle and right column respectively.

5 of Listing 6.4. This change improved the received signal strength at `node05` as shown in Figure 6.10. In particular, the `wlan1.cfg80211.signal.avg` significantly increased from −76 dBm to −68 dBm and similarly the `maxrssi` as it calculates the maximum of the last 20 entires of the same variable. In contrast, the `minrssi` rule shows −77 dBm as its history still contains the values before the adjustment of the transmission power.

If there is no need anymore for monitoring a specific variable or cross-layer coordination, the graphical front-end allows its removal from the display by using the keyword `monitor stop` as shown in line six and seven of Listing 6.4.

To conclude, in our evaluation we have demonstrated that we are able to automate an experiment with several nodes and programs including cross-layer coordination algorithms. Moreover, we showed how to remotely add cross-layer coordination algorithms into the system and run third party programs during the experimentation. By using our remote monitoring feature we monitored the effects on the system. Particularly, we remotely monitored system variables and cross-layer coordination algorithms.

6.3.5 Related Work

The key features of our CRAWLER enhancements are remote automation, configuration and monitoring. Regarding these features there are different projects that offer only one or two of these functionalities.

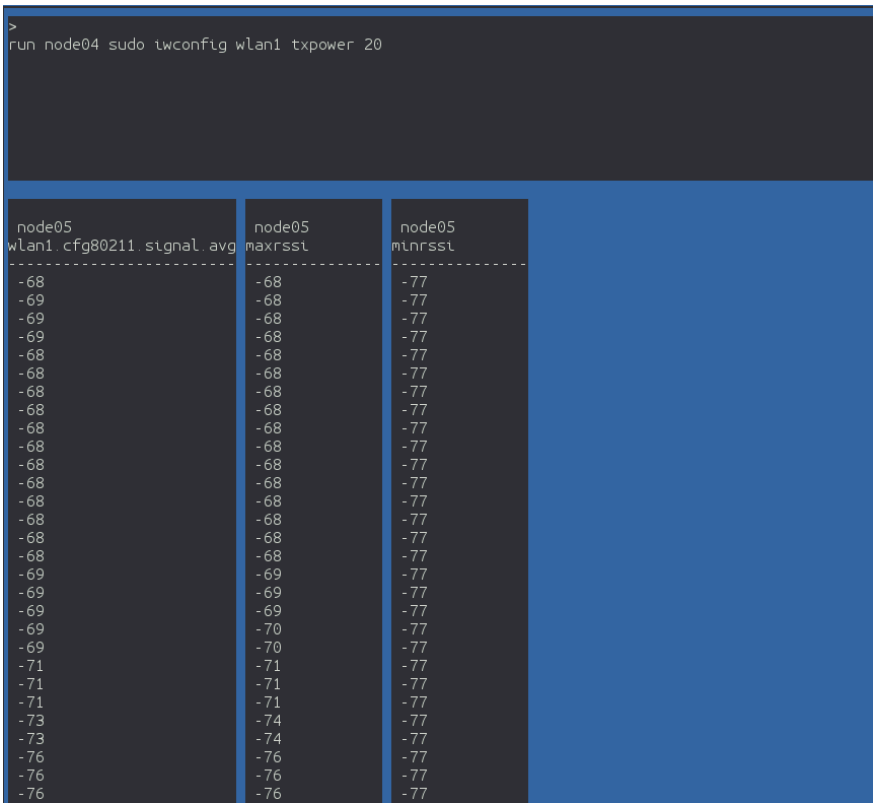


Figure 6.10 The interactive mode allows to run third party programs by using the keyword `run`. For instance, we increased the transmission power on `node04` (the command is shown on the top) which improved the received signal strength on `node05`. Particularly, we monitored on `node05` that the variable `wlan1.cfg80211.signal.avg` and cross-layer rule `maxrssi` increased significantly. On the contrary, as the history of the `minrssi` rule still contains the values before the transmission power adjustment, the low value of - 77 dBm is monitored.

An example with respect to **remote configuration** is OpenFLow [MAB+08] which is a communication protocol that allows remote access to the flow tables of switches and routers. For this, OpenFlow suggests a uniform interface for vendors to allow the configuration of flow tables without revealing internal details of the implementation. The remote devices (routers and switches) are controlled via a secure (SSL) channels using a dedicated controller running on the remote device which denies access for not legitimate peers. However, OpenFlow is primarily focused on high performance flow control and as a result is limited to only configuring flow tables on the remote devices. Accordingly, it neither allows the configuration of other functionalities in the operating system such as sensors nor allows to remotely control applications and all remaining protocol layers besides the network layer.

A candidate in the field of **remote monitoring** is Ganglia [MCC04] which is a system monitoring tool particularly built for high-performance systems such as clusters and grids. The information distribution of the monitored values follows a hierarchical pattern. In particular, servers access the information (called gmond), which is then incorporated into XML files. These files from the server can then be accessed via a polling scheme. Intermediate nodes (called gmetad) poll servers or other intermediate nodes and cache these data. Finally, the intermediate nodes are accessed by a client (referred to as ganglia-web) for visualization of data. However, the major focus of Ganglia is its focus on performance on the clusters. Accordingly, the

supported set of metrics only focus on metrics that give indications about the performance such as CPU and memory utilization. Therefore, monitoring of protocols is not supported, not to mention cross-layer coordination algorithms.

A similar approach is the IETF's standard remote monitoring (RMON) [Wal06] which targets at remote "flow" monitoring in LANs. In its initial version it focuses only on monitoring of OSI layers 1 and 2 in Ethernet and Token Ring. Later versions also include the network and application layer. Although RMON has a more general scope in terms of layer support than Ganglia, nonetheless its concept does not consider other sources of information in the system such as sensors, further wireless access technologies, novel protocols and cross-layer information. In contrast, CRAWLER's design provides the necessary extensibility for monitoring any information in the system.

Preeminent work regarding **remote automation** is Emulab [Emu] which targets at combining network simulation, network emulation and real networks to create a complex testbed. Initially, the testbed has ben created at the University of Utah with several hundred PCs and additional hardware connecting them. As a result of its success, several additional Emulab testbeds at different locations have been created. Emulab provides an infrastructure for testbeds including both (real) physical and simulated nodes. It virtualizes nodes and the links between them. Nonetheless, it allows to configure and automate test runs but this at the expense of high cost, restrictions and complexity to setup tests due to additional infrastructure requirements. This approach targets rather the general case of wireless communication testing. Using Emulab for testing of cross-layer coordination algorithms is not possible. Moreover, the ability of (live) monitoring and logging of variables from system components and protocols to the degree and convenience that we offer is not supported.

A more lightweight approach for remote monitoring is our FANTASY framework [AvLH+12] which uses network emulation as a methodology for experimentation. While the configuration in FANTASY is easy to use and does not require additional infrastructure such as Emulab, there is no support for live monitoring of the experimentation. Moreover, test results are only fetched and collected centrally at the end of each test and cannot be controlled at runtime. In Section 6.4 we present FANTASY in more detail.

Apporach	Remote Configuration	Remote Monitoring	Test Automation	Cross-layer support
OpenFlow [MAB+08]	+ +	−	−	− −
Ganglia [MCC04]	− −	+ +	−	−
RMON [Wal06]	− −	+	−	−
Emulab [Emu]	+ +	−	+ +	− −
FANTASY [AvLH+12]	+	+	+ +	+ +
Presented Concept	**+ +**	**+ +**	**+ +**	**+ +**

Table 6.1 Comparison of the testbeds: ++ very good, + good, − bad, − − very bad.

To summarize, regarding our three key features, related work only focuses on subsets of the functionality provided by our CRAWLER enhancements. For each key feature they target, they have a different focus which separates their work from ours. A major distinction from related work is also the flexibility and convenience provided

by CRAWLER's ability to access information within the system from diverse subcomponents and protocols. Moreover, to the best of our knowledge, none of these works provide all the three key features together as illustrated in Table 6.1. In addition to our key features we added also cross-layer support as an additional key feature, since when it it comes to challenging case of cross-layer coordination testing on real testbeds, there is no framework that provides the degree of support that we provide.

6.3.6 Future Work

The remote configuration feature allows to add, remove and modify cross-layer coordination algorithms at runtime. In the interactive mode we also enabled controlling of other helper programs. Unfortunately, the control of helper programs is limited to starting the programs with their arguments or stopping them. It would be beneficial for developers if they could provide optional arguments or influence the input and output of programs with regular expressions similar to command line instructions. However, we can log the system output on the console, but more control and filtering mechanisms about what to log and how would be useful.

So far it is only possible to control applications that are compiled or rather installed on the test devices. It would be beneficial to have the ability to also push precompiled (cross-compiled) code the test devices. This will simplify the process of bringing all test devices to certain updated versions.

With respect to automation, the configuration of the tests nodes and their addressing is conducted statically (i.e., using IP addresses) in the configuration file. This requires knowledge about all involved test nodes and their manual configuration. Automatic address configuration, for example, with the Dynamic Host Configuration Protocol (DHCP), is not supported yet. Service discovery could be incorporated for an automatic configuration of the remote servers.

6.3.7 Summary

With the presented extension to CRAWLER, we demonstrated that we are able to conveniently evaluate network scenarios where cross-layer coordination algorithms are involved. In particular, we have enhanced CRAWLER's application support interface that facilities three key features.

First, we proposed remote automation which conveniently allows to centrally automate and execute whole test setups with different settings. For this it is only necessary to provide a configuration that includes the necessary instructions to automate the execution of desired settings including applications, cross-layer coordination algorithms and further helper programs like iperf.

Second, we facilitated remote configuration by enhancing the shared library of CRAWLER to remotely feed cross-layer coordination algorithms into a system. With this feature it is possible to remotely add, remove and modify cross-layer coordination algorithms on a remote device at runtime.

Third, to understand the behavior of algorithms and cross-layer coordination algorithms during and after experimentation, we enabled a remote monitoring feature

which allows to log and monitor internal states of cross-layer coordination algorithms as well as variables in protocols and system components.

To further support the developer while experimenting, we integrated all three key features in a graphical front-end, enabling us to centrally control and monitor distributed experiments conveniently.

6.4 Network Emulation Tool – Fantasy

Developers of software for real-world wireless networks have to cope with a number of difficulties. The wireless medium and the distributed nature of the systems complicate the testing of such software. One typical problem is that test setups come with high requirements in terms of hardware and space. Another limitation is that the repeated execution of a test under the same conditions is not possible. While implementations have to be able to cope with the rapidly and unpredictably changing behavior of the wireless channels, during testing, it is beneficial to precisely control the environment to evaluate the influence of different parameters on the system behavior. Including mobility in tests is cumbersome and comes at the cost of human interaction [GKN+04] or highly sophisticated test setups [DRSC05]. Another problem in this context is that it can be difficult to access relevant system information that is required to evaluate a system. Due to the fact that a number of different systems are involved in such a test, the setup followed by an execution of a test and the collection of results rapidly get quite complex.

Network simulation tools are especially built to solve the aforementioned problems, but have their own disadvantages: Regular simulation tools only allow a very limited use of existing software code. This leads to the problem that the system under test has to be implemented twice – once for the use in a network simulator and a second time for use in real systems. This means that only the system concept can be tested, not the actual implementation. Furthermore, simulation models are often too abstract and do not take into account important effects that are caused by an operating system, such as scheduling and buffering.

Network emulation [Fal99] combines simulation and real world testing to benefit from both worlds. However, this is only partially possible in such a combined setup. For example, the real systems that form part of the network emulation setup are not as transparently accessible as the simulated parts. Furthermore, network emulation still requires complex setup for the testbed machines.

In the following we present FANTASY[3], a new network emulation architecture that facilitates the evaluation of wireless network software. Our main contribution is a centrally controlled system that allows the fully automated setup and execution of experiments, enables convenient access to system information, and automates the collection of test results. The target audience of FANTASY are developers of wireless network software that want to perform tests with the convenience of a

[3]This and subsequent sections are based on the joint work with Hendrik vom Lehn, Cristoph Habets, Florian Schmidt, and Klaus Wehrle published in "FANTASY: Fully Automatic Network Emulation Architecture with Cross-Layer Support", 5th International Conference on Simulation Tools and Techniques (SIMUTools '12), March 2012 [AvLH+12]. Furthermore, the content is also partially based on Christoph Habets' Diploma Thesis [Hab11].

network simulation, but nevertheless require the use of real implementations and full operating systems for the system under test. We alleviate the problem of complicated access to relevant system information by integrating the CRAWLER into FANTASY, which provides a unified interface for system information access. Furthermore, since CRAWLER's original focus is on facilitating cross-layer coordination, FANTASY is especially suited as a rapid prototyping and testing tool for the design of cross-layer coordination algorithms.

The remainder of this Section is organized as follows: Section 6.4.1 presents a system overview and introduces the components that FANTASY is composed of. In Section 6.4.2, we describe our architecture in more detail and explain how we achieve the fully automatic setup and execution of experiments. Implementation details are provided in Section 6.4.3. We evaluate FANTASY in Section 6.4.4 and discuss related work in Section 6.4.5. Finally, we describe future improvements in Section 6.4.6, before concluding the section about FANTASY in Section 6.4.7.

6.4.1 Design Overview

The main goal of FANTASY is to simplify the process of testing wireless network software. Through a combination of suited emulation components and the support of fully automatic experiment setup and execution, developers shall be able to perform experiments with real software prototypes as easily as with a network simulation. This section provides an overview over the overall design of FANTASY and the used components.

In order to enable developers to test arbitrary networking software in its native environment (an operating system), but nevertheless minimize hardware requirements, we opted for the use of virtual machines that execute the systems under test. Because of its widespread use and easy configuration, we have chosen VirtualBox [Wat08] as virtualization software.

With FANTASY, those parts of an emulated setup that do not run in virtual machines are simulated using the ns-3 simulation software [ns3]. ns-3 is well suited for this task since it contains detailed models of the MAC layer [BRN+10, Wif] and already comes with support for network emulation. Part of this is a real-time scheduler that runs the simulation in real time, which is required in order to allow the exchange of network packets. Note that network emulation fails if the simulation is not real-time capable, e.g., due to large scenarios or complex models [WvLW11]. In such a case, FANTASY stops processing. In Section 6.4.4.2 we perform a scalability analysis to show how complex such a scenario can become till overload occurs on a single specific PC.

However, to connect the simulation with virtual machines, FANTASY utilizes two components that have been developed as part of the SliceTime project [WvLW11]. For the emulation of Ethernet devices, a tap device which connects to ns-3 using UDP datagrams is created in the system that runs inside the virtual machine. A wireless emulation driver is used for the emulation of wireless network connections. This device driver creates a virtual network device which provides the same interfaces as a real 802.11 wireless network card, including the wireless extensions [Wir]. Both devices have in common that they provide interfaces to the guest operating system,

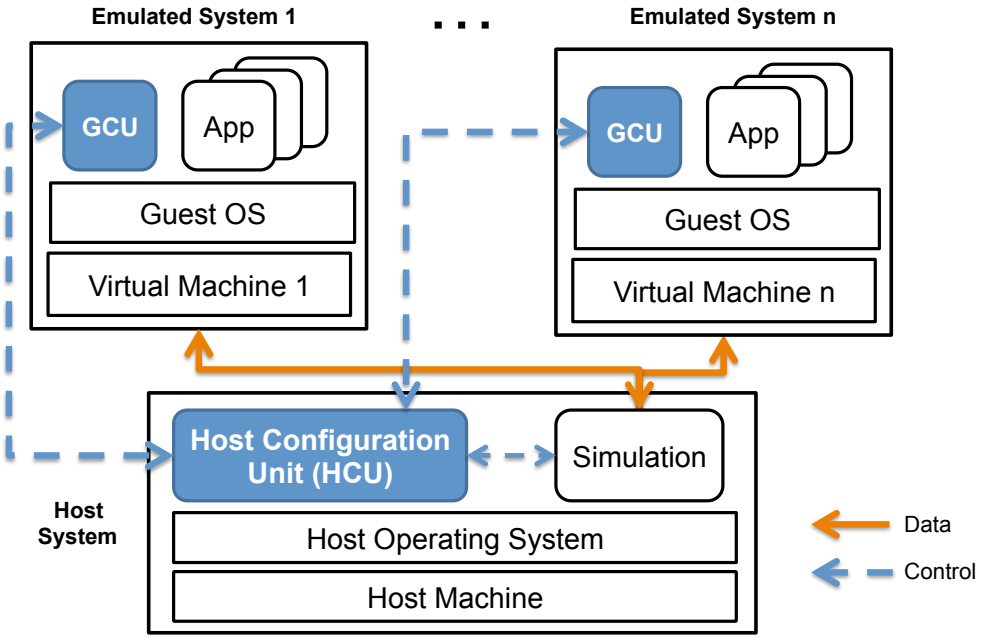


Figure 6.11 Conceptual view of FANTASY.

but forward all sent and received frames to corresponding models inside the ns-3 network simulation.

An advantage of network simulation that is usually lost in case of network emulation [Fal99], is the convenient access to relevant information in the network stack such as packet loss rate at the application layer or signal strength at the PHY layer. Through the incorporation of CRAWLER, FANTASY supports easy access to such information of the systems that run inside the virtual machines. Furthermore, this combination allows to conveniently evaluate the effects of cross-layer coordination algorithms using real-world systems. The only restriction regarding the software that runs inside the virtual machines is that they are constrained to Linux systems, as the wireless emulation driver and CRAWLER have been developed for Linux.

By using the aforementioned components, FANTASY allows the emulation of diverse scenarios on a single computer. Nonetheless in cases of low computational power, FANTASY also allows to use additional computers to run the virtual machines. However, the overall setup and execution of an experiment is still quite complex when compared to the simplicity of a pure network simulation. To further simplify these processes, we developed two components that allow to control the whole setup from a central place. The host configuration unit (HCU) is running on the host computer that accommodates the network simulation and the desired amount of virtual machines. It instantiates the virtual machines even on different physical devices, starts the network simulation and is in charge of the overall setup. As part of the systems that are running inside the virtual machines, the guest configuration unit (GCU) waits for commands from the HCU. It loads the wireless emulation driver, configures CRAWLER, starts processes that are part of the experiment and allows the collection of test results at a central place.

Figure 6.11 gives an overview of the overall setup that is used by FANTASY. The functionality of HCU and GCU are described in more detail in the following section.

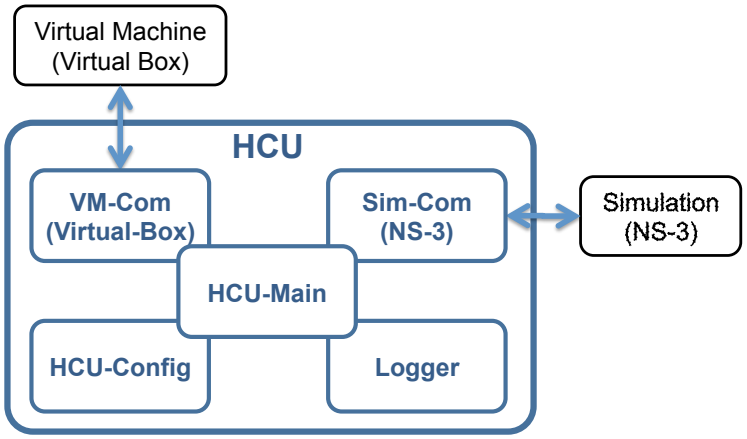


Figure 6.12 Overview over the Host Configuration Unit (HCU), its interfaces, and its subcomponents.

6.4.2 Architectural Details

This section provides more detailed information on how HCU and GCU are used to control whole experiments in an automated fashion. Note that the architectural design of FANTASY in general and the components HCU and GCU in particular are comparable to the client and sever components that enable CRAWLER's remote features, but for an emulation approach additional aspects need to be considered.

6.4.2.1 Host Configuration Unit (HCU)

The Host Configuration Unit (HCU) is executed on the host system and controls the entire experiment. It consists of five subcomponents as shown in Figure 6.12. While designing the HCU we kept it modular to group functionality for usability and maintainability reasons. Moreover, the modularity provides us with exchangeable modules such as the VM-Com subcomponent that is tailored for Virtual Box but can simply be exchanged for another virtual machine software. This holds also for the Sim-Com subcomponent that is tailored for ns-3 but can also be exchanged for another simulator. The HCU-Main subcomponent is interconnected with all subcomponents and controls them.

The initial step to conduct a test setup is done by writing a configuration file. The configuration includes the required parameters for the simulation as well as instructions that are performed on the virtual machines. The configuration file is read and executed by the HCU-Config subcomponent. After the experiment, the logged values for all parameters specified in the configuration are collected from all virtual machines and are given back to the Logger subcomponent. In the following we describe how such values can be configured in order to monitor them and how to setup up a complete experiment.

Configuration

Typically when testing is involved, the experimenter has to adjust parameters to evaluate the effects such as number of involved nodes, different traffic patterns, or used protocols. This also requires to repeat the test several times to have stable and credible experimentation results. It is desirable that both of these steps are very simple to achieve. Therefore, the aim of our configuration is to support a very automated, customizable and easy-to-use testing environment.

In our approach a whole test setup with different settings is described in a configuration file. The configuration contains everything necessary to automate diverse test settings of the simulation as well as application and cross-layer settings in the VM. Repeatability of the test setup is given by re-executing the configuration file which can also be adjusted in the configuration. Among the different configurations we have including Crawler's remote automation feature, we wanted to keep the syntax as similar as possible to decrease the learning effort. Thus, the general syntax of the configuration of Fantasy is similar to Crawler's automation feature: For comparison, we refer the reader to Section 6.3.2.1. A ready-to-use configuration example is shown in Listing 6.6. Here, the configuration is subdivided into sections by using the keywords [ns3], [vm], [schedule] and [logger].

```
[ns3]
# Path to the ns-3 folder
ns3_path: /home/crosslayer/ns-3.7-slicetime

# Path to the ns-3 configuration file;
# leave empty to use default configuration
ns3_config: /ns_3/usecase2.cc

# Offset in seconds to start the HCU after simulation
hcu_start_offset: 0.5

# Simulation duration in seconds
ns3_duration: 50

# (Parameter,value)-pairs to adjust ns-3 config
ns3_param_value: [("sim_node_count",1),("protocol","TCP"),("wlan
    ","a")]

[vm]
# Number of VMs to be started
vm_machines: 2

# Template file for the VMs
vm_templ: ./virtual_machines/templates/default.vdi

[schedule]
wifi:[([1,2], 0.7, "load_wifi_emu")]
crawler:[(1, 1, "load_crawler tcpLayer CLKernelModule/src/test/
    enabletcp.ko")]
iperf:[(2, 4, "iperf -s -p 5001"), (1, 5, "iperf -c 192.168.1.2 -t
     30 -y c -p 5001 -x CMSV -i 1 &> iperf.log")]
tcp_cwnd:[(1, 5, "crawlerapp monitorapp transport.tcp.tcp_out_5001
    .cwnd &> /home/crosslayer/gcu/tcp_cwnd.log")]
fer:[(1, 1.5, "crawlerapp monitorapp wemu0.wireless_stats.qual.fer
     &> /home/crosslayer/gcu/fer.log")]

[logger]
# what should be logged
log:[(1,"iperf.log"),(1,"tcp_cwnd.log"),(1,"fer.log"),(1,"gcu.log
    ")]
# Path to where the log files should be kept
log_path: ./logs
```

Listing 6.6 A simple configuration file in Fantasy.

In section [ns3] simulation related settings are listed. Line 3 indicates which ns-3 version is used: For example, we used the ns-3 version 3.7 but modified for Slice-time [WSvL+11]. In line 7 the ns-3 configuration, already preconfigured, is loaded. As describing how to provide ns-3 configurations is out of the scope go this thesis, we omit here the description. However, the parameters of the ns-3 configuration can be re-adjusted within the configuration by assigning values to parameters with the use of tuples as shown in line 16.

Virtual machine related settings are listed in section [vm]. For example, the number of emulated nodes is set on line 20 and their VM image is loaded from a path given on line 23.

The main part of the configuration is the [schedule] section (similar to CRAWLER's automation configuration) which determines when to execute which instruction. For example, in line 28 indicated with iperf, we used two tuples. We introduced a three tuple notation `(<vm>, <time>, <command>)`; the first index indicates the VM, the second index the relative time in seconds. Note that the experimentation starting time between all VMs is synchronized, i.e., all VMs are set once to a common time zero. After the simulation has started, the HCU receives a signal and itself sends a signal to all VMs setting them to the common time zero. The third index gives instruction about what should be done on that particular VM. If we come back to the example at line 28, an iperf server is started on VM 2 at time 4 and an iperf client on VM 2 at time 5. Similarly, at line 29 a monitoring application is started on VM 1 at time 5 that uses the CRAWLER shared library to monitor the congestion window (CWND) of TCP. The outcome of the monitored parameter is stored in a log file. The same holds for line 30 where the frame error rate (FER) is observed. Remark, in line 26 we use a special notation by using brackets for the VMs to address multiple VMs at once. Here, the first index has two arguments in brackets. With this notation both values within this brackets are combined with the other two indexes and this will be interpreted as two three tuples. This is very helpful when the time and commands that should be executed is the same for multiple VMs.

To deliver the logged values from the GCU within the VMs to a central place, the [logger] section is used. For example, line 34 indicates which values should be delivered and line 36 indicates where these values should be stored. Results for this particular configuration will be presented in Section 6.4.4.

So far, we have only seen a fixed test setup, a single experiment without changing parameters. What happens if we want to have a slightly different test setup? To relieve the user from having to write an additional configuration file for only minor changes we distinguish between **main** and **custom configuration** files. The main configuration file describes the major test setup while the custom configuration only includes the differences. The custom configuration overwrites the respective values of the main configuration. For example, if we want to use a different simulation configuration as given in the main configuration shown in line 7 of Listing 6.6, we have to add only a single modified line in the custom configuration to the respective section ([ns3]), e.g., `ns3_config:./ns_3/usecaseXX.cc`, that overwrites the main configuration. The main and custom configurations can be simply started with executing the following instruction in the console:

```
$>python hcu.py -mf <main_config> -cf <custom_configs>
```

The `-mf` option expects only one main configuration while the `-cf` option allows to add several custom configurations or a directory including several custom configurations.

However, when many similar tests with only minor changes are supposed to be conducted, the process of creating custom configuration can be very cumbersome. To alleviate this process, we have implemented a configuration generator which is explained next.

Configuration Generator

The configuration generator is an interactive tool that helps to create several custom configurations. It is started in the console as follows:

```
$>python hcu.py -gf <config>
```

The option `-gf` accepts a configuration file of any sort as an argument which is parsed and the parameter assignments such as `ns3_param_value` in line 16 of Listing 3.1 are detected and used as default values. For each of the parameters needed by a main configuration, the configuration generator asks for assignments. If a default value is present for a parameter it can be kept by just pressing enter and moving on to the next parameter.

We have introduced a list notation in brackets that allows to assign several values to a variable. The following example assigns several values to the variable `ns3_param_value`:

```
ns3_param_value=
("sim_node_count", [0,1,2,3,4,5,6,7,8,9,10,11,12]), ("protocol",["
    UDP","TCP"]),("wlan",["a","b"])
```

For each of the parameters within these tuples one value is assigned from the list. In this particular example, the permutation of all these assignments allows to conduct 52 tests, all of which can be started by issuing one command that, for ease of use, is given to the user when the configuration generator finished successfully. This is a shortened example that we also use later in the evaluation in Section 6.4.4.2. The only difference is that we configured more simulated nodes (ranging till 20 nodes), more runs (10) and one additional assignment to another variable (TCP and UDP traffic between emulated nodes) which leads to 1680 different test runs for generating the experimentation results as presented in Section 6.4.4.2. As a result, with the help of the configuration generator, custom configurations are automatically generated and then used to run automated tests without any further interaction from the experimenter.

6.4.2.2 Guest Configuration Unit (GCU)

The HCU gives instructions on what should be done when a certain time arrives such as starting the simulation or starting a real application on a VM. Remember that VMs could not only run on the host computer but also on additional computers if necessary. However, these instructions given by the HCU have to be received and

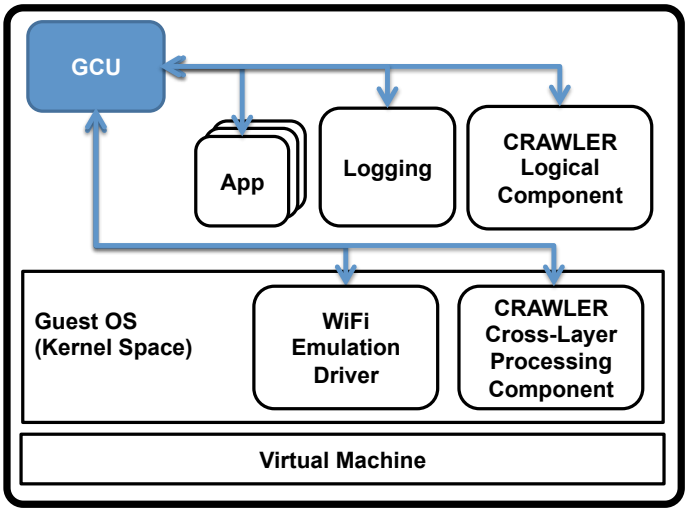


Figure 6.13 Overview over the tasks and responsibilities controlled by the guest configuration unit (GCU).

performed on the VMs by a counterpart. This responsibility is taken over by the Guest Configuration Unit (GCU). Through the GCU, the HCU has the power to fully control the VMs. Within each VM, the GCU is executed when the system is started. This entails that the experimenter must install the GCU software on the VM template which the HCU clones (copies) before starting the experiment.

Figure 6.13 shows the main responsibilities of the GCU. These responsibilities include loading applications that have been determined in the configuration (like iperf in Listing 6.6). The GCU also loads CRAWLER [AOSW10] which allows passive monitoring and active manipulation of protocol and system variables within the VM, as well as the WiFi emulation driver [WvLW11] which couples the VM with the ns-3 simulation.

In a simulation, accessing protocol variables is relatively easy. Most network simulators are designed with ease of access and observation of protocol or system variables in mind. However, in a real system, the access to many of such variables is restricted. This is mainly due to the protocol stack being integrated into the operating system which only provides few limited interfaces mainly because of security concerns. To facilitate access to protocol and system variables in a real machine, we use CRAWLER. FANTASY integrates support in a way that it hides the details of the CRAWLER implementations and relieves the developer from directly configuring the framework. We do this by providing a wrapper called `monitorapp` that translates FANTASY logging instructions into CRAWLER monitoring instructions (cf. Listing 6.6, lines 29 and 30). For details about how the `monitorapp` works, we refer the reader to Section 3.4.2.

Since CRAWLER originated as a cross-layer coordination framework, it is of course also possible to use it for this purpose. Developers who want to test cross-layer coordination algorithms in a VM can do so without any further changes to the standard FANTASY setup. The configuration to define cross-layer coordination algorithms is syntactically very similar to FANTASY's configurations and therefore poses little additional learning effort. Details about CRAWLER's configuration language to design cross-layer coordination algorithms is explained in Section 3.4.1.1.

6.4.3 Implementation

FANTASY is a combination of many different tools and libraries. Of those, two of the vital pieces, CRAWLER and the WiFi emulation driver tightly integrate into the Linux kernel (2.6.32). This means that, while the developer is free to choose a distribution, they are restricted to use Linux as an operating system. For our tests, we used Ubuntu distributions. All components of the HCU and GCU are implemented in Python [pyt].

The communication channel between the HCU and the GCU is realized through a separate virtual network in which the HCU provides IP addresses to the GCUs via DHCP. As a communication protocol between the HCU and GCU we used the Python Remote Objects (Pyro) [dJ] package for Python, which runs a daemon inside each VM allowing the HCU to connect to it. Thus, the HCU is able to call functions of an instance of the GCU as if they were local objects even running on different (physical) machines.

6.4.4 Evaluation

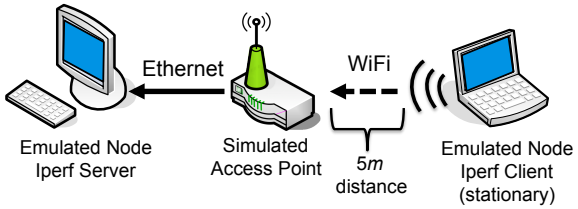
With our evaluation we show different test cases where we emphasize different features of FANTASY. In each test, we show a subset of these features. The highlighted features are: (i) comparability of results between real world tests and FANTASY (ii) cross-layer support for emulated nodes that allows us to passively monitor protocol and system information as well as accessing them actively; (iii) repeatability of experiments; (iv) support of mobile wireless scenarios; (v) automation and rapid testing capabilities. Furthermore, we will also give insight into the scalability (i.e., till the real time demand is violated) of FANTASY when using only a single computer for the whole experimentation.

6.4.4.1 Demonstrating Areas of Application

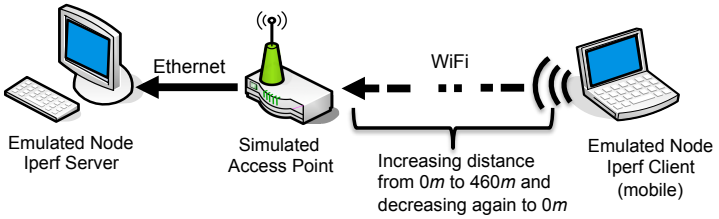
In this section we give two examples to demonstrate the areas of application for FANTASY. In the first example we have conducted real world tests and we compare them with the emulation results achieved with FANTASY. This also highlights the integrated cross-layer support. In the second example, we showcase simulated mobility as well as monitoring of parameters of several protocols in the emulated node's Linux kernel. In both test cases, the physical layer and channel are simulated with ns-3's channel model. We used the two-ray ground propagation loss model that is part of ns-3 WiFi model [Wif] for our simulated wireless channel.

Test Case 1: Comparability, Cross-Layer Coordination Support

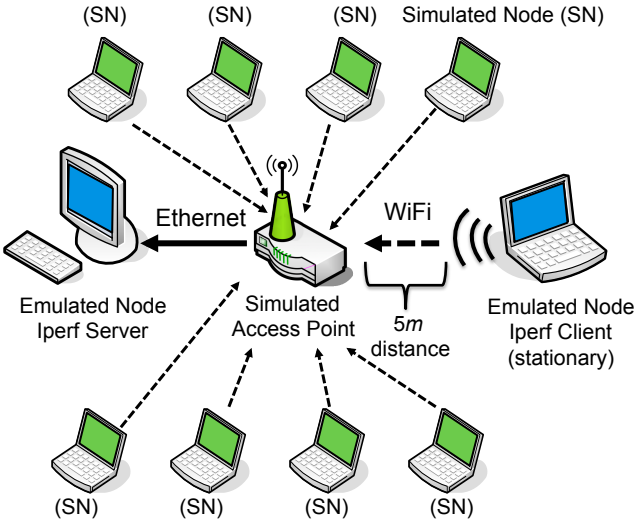
Inspired by our use-cases presented in Chapter 4 where we already collected real-world experiments, we decided to reevaluate the TCP congestion control use-case from Section 4.2 with FANTASY. In this use-case, the idea is to change TCP's congestion control algorithm depending upon the underlying network conditions. The brief motivation behind the change is as follows: TCP CUBIC [HRX08] is the standard



(a) Test case 1: comparison of real world tests and emulated results with FANTASY



(b) Test case 2: monitoring several parameters across several protocol layers of a mobile emulated node

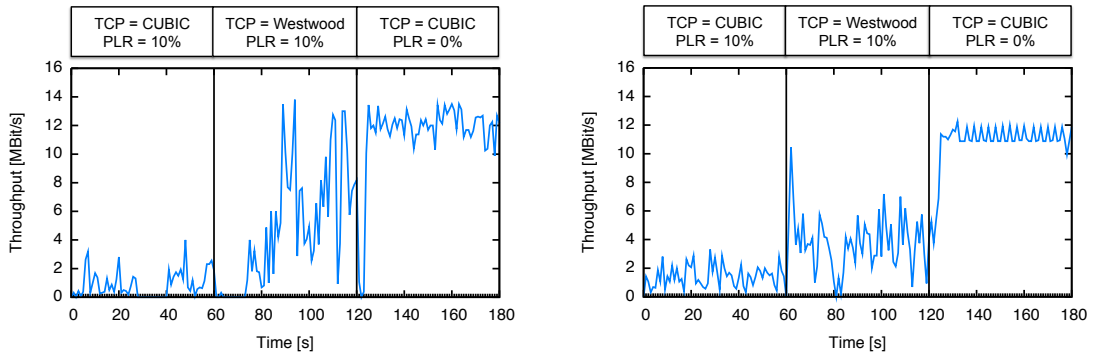


(c) Test case 3: testing the scalability by adding simulated nodes until the simulation's real-time capability is lost.

Figure 6.14 Topology of the three test case scenarios that were used for evaluation.

congestion control algorithm in the Linux kernel since version 2.6.19 due to its superior performance and fairness properties under most network conditions. However, TCP Westwood [MCG+01], specifically developed for wireless communications (such as in WiFi), provides better throughput in changing network conditions with high packet loss rates. The cross-layer coordination idea is to switch between different congestion control algorithms at runtime without reinitializing the TCP connection, based on the observed network conditions. For more details about the coordination idea we refer the reader to Section 4.2.

In the real world testbed, our test setup consists of two PCs and one 802.11g access point. One PC runs the cross-layer coordination configured in CRAWLER, and is connected to the access point via WiFi. On this PC we use iperf [TQD+04] to create TCP traffic, and netem [Hem05] to create different packet loss rates (PLRs). The other PC serves as the destination for iperf traffic; it is connected to the access point



(a) Testbed measurements

(b) Emulated measurements with FANTASY

Figure 6.15 Comparison of throughput measurements for a cross-layer coordination scenario. TCP's congestion control algorithm is changed based on the changing packet loss ratio (PLR) and the received signal strength indicator (RSSI).

via Ethernet. In FANTASY, the two PCs are emulated virtual machine nodes that use the emulated network drivers to connect to a simulation setup. This simulation models an 802.11a access point since 802.11g functionality is not given in the used ns-3 version; but for our test setup 802.11a should largely show the same behavior as it only differs in its PHY layer. The test setup is depicted in Figure 6.14(a).

Figure 6.15 shows the results of our experiments; Figure 6.15(a) shows the real-world results and Figure 6.15(b) shows the results gained with FANTASY. For the first 120 seconds, we have set the PLR to 10% which is a very significant PLR for TCP. For the first 60 seconds, the coordination is not active, as depicted by the low TCP throughput achieved during this time. The coordination is activated at 60 seconds which triggers the switch from CUBIC to Westwood and subsequently improves the throughput. At 120 seconds, we set the PLR to 0%, so that TCP switches back to CUBIC and thus achieves a consistently higher throughput. Although the results obviously are not perfectly similar, they show the same tendency. The differences can be mainly attributed to channel effects; the emulated setup uses a simple path loss model and does not account for small-scale fading or interference from other nearby machines that use a wireless connection, but are not part of the test setup.

This test case demonstrates the following features: We have gained comparable results between real world tests and our architecture. Furthermore, we showed an example of executing a cross-layer coordination and continuously monitoring a protocol parameter.

Test Case 2: Monitoring, Mobility, Repeatability

As FANTASY also supports mobility and monitoring of diverse parameters across protocol layers and system components, we want to give an additional example where we demonstrate these features. For our second test case (cf. Figure 6.14(b)), we use an emulated setup similar to the previous one. However, we use a standard TCP setup without any cross-layer coordination algorithms. Instead, we introduce mobility by simulating a constant speed movement. The node that is connected to the AP via WiFi starts at a distance of $0\,m$ to the access point and moves to a distance of $460\,m$ at a speed of $1\,m/s$. Upon reaching that point, it moves back to

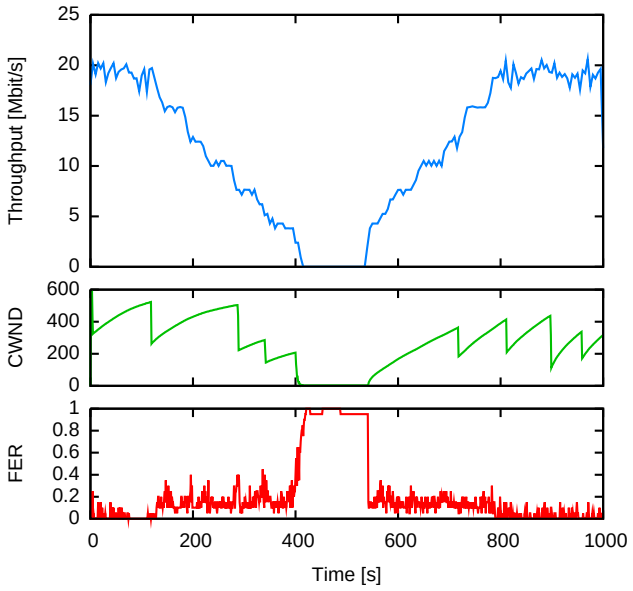


Figure 6.16 Monitoring of three metrics (TCP throughput, TCP congestion window, and frame error rate) at an emulated node over time. The node starts at a distance of 0 m to the access point, moves away at a constant speed up to a distance of 460 m, and turns back until it reaches 0 m again.

0 m at the same speed. To showcase the monitoring capabilities of FANTASY, we logged several key metrics throughout the experiment. The setup is the result of the configuration shown in Listing 6.6 and discussed in Section 3.4.1.1.

The monitored parameters are shown in Figure 6.16. From the application layer we logged the throughput measurements gained by iperf, from the transport layer the congestion window (CWND), and from the WiFi emulation device driver the frame error rate (FER). We chose these parameters for two reasons. First, they are good candidates to demonstrate the effects of mobility and packet loss. Second, we show that we are able to monitor different protocol layers or system components within an emulated node.

As can be seen in Figure 6.16, the throughput is reduced with increasing distance. The step pattern of the throughput curve shows the effect of rate adaptation. At around 400 s, the distance becomes too large for any meaningful communication: virtually all frames are dropped due to errors, and no TCP packets are received any more. After the node reaches its maximum distance and slowly returns back to the AP, the communication recommences at around 550 s, and throughput gradually increases afterwards.

To investigate whether we can produce repeatable test setups with FANTASY, we ran this scenario several times with identical simulation settings. As can be seen in Figure 6.17, the throughput is almost exactly the same over all runs. Other measured parameters were also highly similar to each other.

With this test case, we demonstrate FANTASY's capabilities to model mobile wireless scenarios. We also give an example of monitoring and logging of system and protocol parameters inside a virtual machine, and we use this to show how FANTASY produces very similar results over several runs with identical simulator setups.

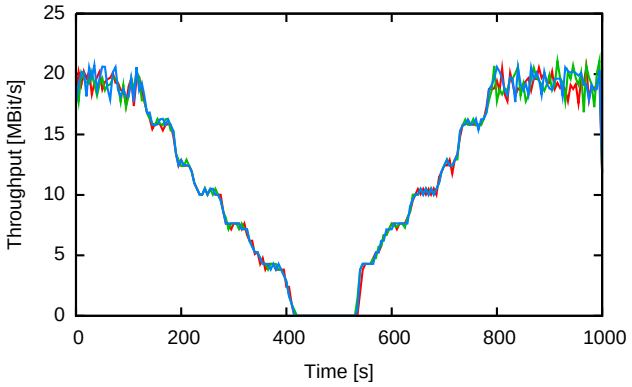


Figure 6.17 Monitored TCP throughput from three simulation runs with the same parameters. Repeated runs produce highly similar results.

6.4.4.2 Demonstrating Scalability, Automation and Rapid Testing

If the simulation consists of complex scenarios (e.g., too many nodes or computational intensive models) and accordingly needs to process too many events, it is not able to follow the real time demands needed by the real part of the emulation. In such a case, simulation overload introduces artifacts that strongly distort the results. FANTASY stops the experimentation in such a case. We therefore want to determine how complex the simulation can get before simulation overload occurs. This of course strongly depends on the performance of the computer running the simulator and the virtual machines. Evidently, a stronger machine allows more complex simulations and more emulated nodes. For our tests, we used a Dell OptiPlex 960 with 4GB Ram and Intel Core2 Quad CPU Q9400, each processor running at 2.66GHz. Ubuntu 10.04 was installed on an external USB 2TB hard disc.

The test setup is very similar to the previous ones but without simulating mobility. The emulated WiFi node is kept at a fixed distance of 5m and sends netperf [JCS] traffic to the emulated Ethernet node. With each run, we add an additional simulated node to the simulation setup. We continued this until the simulation experienced overload and canceled the evaluation. Figure 6.14(c) shows the set for this third test case. We placed the simulated nodes equidistantly around the access point. The cross-traffic created by the simulated nodes always adds up to 8 Mbit/s, so that, for example, with four nodes, each simulated node sends with 2 Mbit/s. This way, the channel is always kept roughly equally busy from traffic by the simulated nodes, with netperf using up the remaining capacity.

Apart from the number of simulated nodes ranging from 0 to 20, we varied three settings for this evaluation setup: (a) we used either TCP or UDP for the netperf traffic between the emulated nodes, (b) we used either TCP or UDP for the traffic between the simulated nodes, and (c) we used either 802.11a or 802.11b as the WLAN standard. We also conducted each test 10 times, resulting in 1680 experimental runs. How these diverse tests were generated has been shown in Section 6.4.2.1. The reasons for these different tests were all related to simulator performance. UDP produces less overhead than TCP due to the less complex protocol. 802.11b produces less load on the simulator than 802.11a because of the lower maximum speed of the wireless network (11 Mbit/s and 54 Mbit/s, respectively).

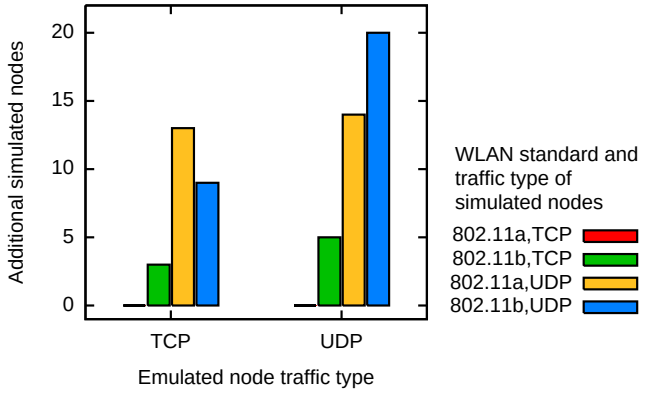


Figure 6.18 Number of nodes that could be simulated using FANTASY before simulation overload occurred.

The results of our tests are summarized in Figure 6.18. In general, the results confirmed our expectations. Generating UDP traffic between the emulated nodes allowed us to add more simulated nodes before a simulation overload is achieved (compare same-colored bars with each other) compared to the case of generating TCP traffic. Likewise, UDP cross-traffic allowed us to add more nodes than TCP cross-traffic (compare first pair of bars in each set with second pair). 802.11b allowed us to add more nodes than 802.11a (compare first bar with second, and third with fourth in each set). The exception from these rules are 802.11a with TCP cross-traffic, in which both experiments overloaded after adding a single cross-traffic node. The setup with just the emulated nodes running netperf completed successfully for both TCP and UDP traffic.

In this section, we have demonstrated that we are able to set up a mix of emulated and simulated nodes. The complexity of running the scenario depends on the used protocols, the used MAC layer models, and the used hardware to run the network emulation setup. We will discuss concepts to improve the performance and allow larger emulated networks in Section 6.4.6.

6.4.5 Related Work

Since network simulation and emulation play an important role in protocol evaluation, there are different projects that offer functionality similar to FANTASY. Table 6.2 gives an overview about these proposals.

From these proposals arguably the most elaborate is *Emulab* [Emu], a project that aims at combining network simulation, network emulation and real networks to create a complex testbed. From the first implementation at the University of Utah, several additional Emulab testbeds at different locations were spawned, demonstrating the concept's success. The original Emulab consists of several hundred PCs with additional hardware connecting them. Later additions to the testbed include stationary wireless nodes, as well as *Mobile Emulab* [JSF+06] which uses mobile robots to carry sensor nodes in a room dedicated to this purpose. Clearly, this setup allows testing in wireless networks with a lot of realism and support of repeatability, but access to existing testbeds is limited and costs for the setup of a new testbed are high. With emulation, Emulab supports another way to perform tests in wireless

Testbed	Cost	Realism	Accessibility	Usability	Versatility	Cross-layer support
Emulab [Emu]	–	+	–	–	+	–
MobEmu [JSF+06]	– –	++	– –	–	++	– –
NEPI [LFH+10]	+	++	+	– –	++	–
CLSO [LCG+09]	++	– –	++	++	–	+
ARE [Dör07]	+	–	++	+	– –	+
FANTASY	+	+	++	+	+	++

Table 6.2 Comparison of the testbeds: ++ very good, + good, – bad, – – very bad.

networks. However, the emulation features in general are more limited and simple than the functionality that FANTASY offers through the incorporation of ns-3, especially monitoring and cross-layer coordination across all layers of the simulated and real-world part.

An approach similar to FANTASY is given by the *Network Experiment Programming Interface (NEPI)* [LFH+10] which tries to offer a single user interface for testing with many different tools. For example, NEPI can use *ns-3*, *PlanetLab*, *Emulab* and *ORBIT* in its tests to create diverse setups and topologies. Since NEPI tries to offer an uniform interface for this diverse set of tools, the interface of NEPI itself is already quite complex. If this combination of different tools is not required, it is therefore easier to learn the use of a single tool such as FANTASY.

Another similar concept is given with *Ad-hoc Routing/Emulation* [Dör07] where a cross-layer implementation is tested on virtual machines. These virtual machines represent robots that have to navigate through a virtual world created by a simulation. However, the virtual machines do not interact with the simulation but only get status information about the virtual world therefrom. This approach targets a rather tailor-made cross-layer solution as it was designed specifically to test the robots' navigation. This approach can not be applied to other cross-layer design ideas and thus using this setup for cross-layer testing in general is not possible.

6.4.6 Future Work

As the scalability tightly depends on the computing power of the test machine, the power needed to be real-time capable rapidly grows with the simulation complexity. In other words, for any amount of computing power, it is possible to create network emulation setups that will overload the simulation and therefore be too complex to be run in real-time. Therefore, we are considering to solve this problem that FANTASY suffers from like all other network emulation frameworks more fundamentally by relieving it from that real-time constraint.

One solution to do so is SliceTime [WSvL+11], which continuously synchronizes virtual machines and network simulation with each other to counter the time drift that originates from simulation overload. We are currently investigating the complexity of integrating SliceTime with FANTASY. Since SliceTime is tightly integrated with Xen [BDF+03, Cit] instead of VirtualBox, this will mean a change in the VM software used. Due to the modular design of FANTASY, this will be mainly a question of creating a new VM-Com module (see Section 6.4.2.1) for the HCU.

6.4.7 Summary

We presented FANTASY, a new network emulation architecture that allows the fully automated setup and execution of an experiment, enables the convenient access to system information and the collection of test results. With the integration of the cross-layer architecture CRAWLER, we demonstrated that we are able to monitor parameters in a mobile emulated node across protocol layers and to evaluate network emulation scenarios where cross-layer coordination is involved.

Our fully automated emulation architecture is already an essential part for testing of cross-layer coordination algorithms within the CRAWLER project [AOSW10]. As we believe that our fully automatic network emulation architecture will be useful to other researchers and developers in the area of wireless networks, we have made the source code[4] available to the public.

6.5 Conclusion

We proposed two different approaches to support developers while experimenting with cross-layer coordination algorithms.

Our first approach is an enhancement to our CRAWLER's application support interface and allows the experimenter the remote automation, configuration and monitoring of experiments where cross-layer coordination algorithms are involved. The focus of our remote evaluation enhancements is the real-world testing. In particular, our enhancements extend CRAWLER by enabling the following three key features: (i) The central automation and execution of whole test setups with different settings, (ii) the remote injection, removal, and modification of cross-layer coordination processes, and (iii) the central and remote monitoring and logging of specified set of system variables and states of cross-layer coordination algorithms.

Our second approach, called FANTASY, is a network emulation architecture that complements our first approach by enabling a method to isolate and control influencing factors. FANTASY uses network emulation as an approach to gain more control about the wireless environment which is running in a simulation environment. The part (layer one and two) that is being simulated in FANTASY allows to provide more control to a developer about factors such as wireless and mobile effects. The ability to control or isolate these factors support developers in understanding and validating the behavior of their coordination algorithms. Moreover, the integration of CRAWLER allows the same convenience and power to monitor variables inside the "real part" of the experiment as possible as in simulation.

Concept-wise both approaches are comparable, as both consist of two components that are functionally similar. While in FANTASY the controlling entity is the HCU and instructs the GCUs residing in VMs, the controlling entity for our remote evaluation enhancements of CRAWLER is the client and the server is the entity that receives and realizes the instructions on a real computer.

[4] All source files are available at:
`http://www.comsys.rwth-aachen.de/research/projects/crawler`

7

Summary and Conclusions

Although the cross-layer design paradigm has demonstrated its high potential to improve adaptability and performance, it has not been able to leverage its utility beyond few promising yet concentrated research efforts. Among other reasons, this can partly be attributed to the naïve utilization of the design paradigm resulting in solution-oriented implementations. A direct consequence of its naïve use is the missing support of software engineering principles such as maintainability and reuse, which are necessary for successful proliferation of software.

As a result of this observation, a few static cross-layer architectures have been proposed that support a developer in the design process of cross-layer coordination algorithms. In particular, static architectures facilitate integration of several coordination algorithms while satisfying software engineering principles. We have identified three major challenges that limit real utilization of static architectures.

First, static cross-layer architectures neither support dynamic adaptation, i.e., addition, removal, and modification of cross-layer coordination algorithms nor context adaptation, i.e., automatically loading the adequate set of coordination algorithms at runtime based on the detection of underlying environmental changes.

Second, while designing a single cross-layer coordination is already a complicated task, the process gets significantly more complex for multiple cross-layer coordination algorithms. Problems such as contradicting coordination goals and redundancy of coordination algorithms appear. Static architectures do not support developers to deal with these problems.

Third, testing cross-layer coordination algorithms in general is a tedious task. On the one hand, setting up an experiment requires huge effort to install the relevant software involving cross-layer coordination algorithms and their coordination among different devices. On the other hand, to obtain credible results an experiment usually requires many test runs since the volatile nature of the wireless medium complicates the interpretation of evaluation results. In either of these cases, static architectures do not support a developer.

As a result of missing flexibility in particular and support for developers while experimenting with different sets of cross-layer coordination algorithms in general, we have identified and tackled three main research questions:

Question 1 (Q1) - How to enable a generic and runtime flexible cross-layer architecture that facilitates convenient system monitoring, cross-layer design and experimentation?
We designed CRAWLER, a generic and runtime flexible cross-layer architecture that answers Q1 in Chapter 3. In addition, we demonstrated the practical use of CRAWLER on several use-cases in Chapter 4.

Question 2 (Q2) - How to handle problems caused by multiple cross-layer coordination algorithms?
We have investigated the problems of running multiple cross-layer coordination algorithms in Chapter 5. We have identified and tackled the two distinct problems of cross-layer conflicts and redundancy of coordination algorithms, respectively.

Question 3 (Q3) - How to improve the evaluation of cross-layer coordination algorithms?
We support developers by providing two different frameworks (emulation approach vs. real-world testbed) that answers this question in Chapter 6. These frameworks are extensions to CRAWLER and allow central automation of experiments (including cross-layer coordination algorithms), their central monitoring and logging.

In the following, we summarize the major contributions of this thesis in detail.

7.1 Contributions

This thesis makes four contributions that address the aforementioned questions. Figure 7.1 visualizes the relationship among the research questions and our contributions. While trying to provide answers to the research questions, we have identified further challenges and our key features tackle these challenges. In the following, we briefly summarize our four contributions and their key features.

Contribution C1 – CRAWLER: A Generic and Flexible Cross-Layer Architecture

Our contribution (C1) answers our first research questions (Q1) by providing the following key features:

Feature F1 - Manageability
With CRAWLER cross-layer coordination algorithms are easily maintainable and usable for application and system developers as it does not require to deal with system details and architectural requirements. This is achieved with

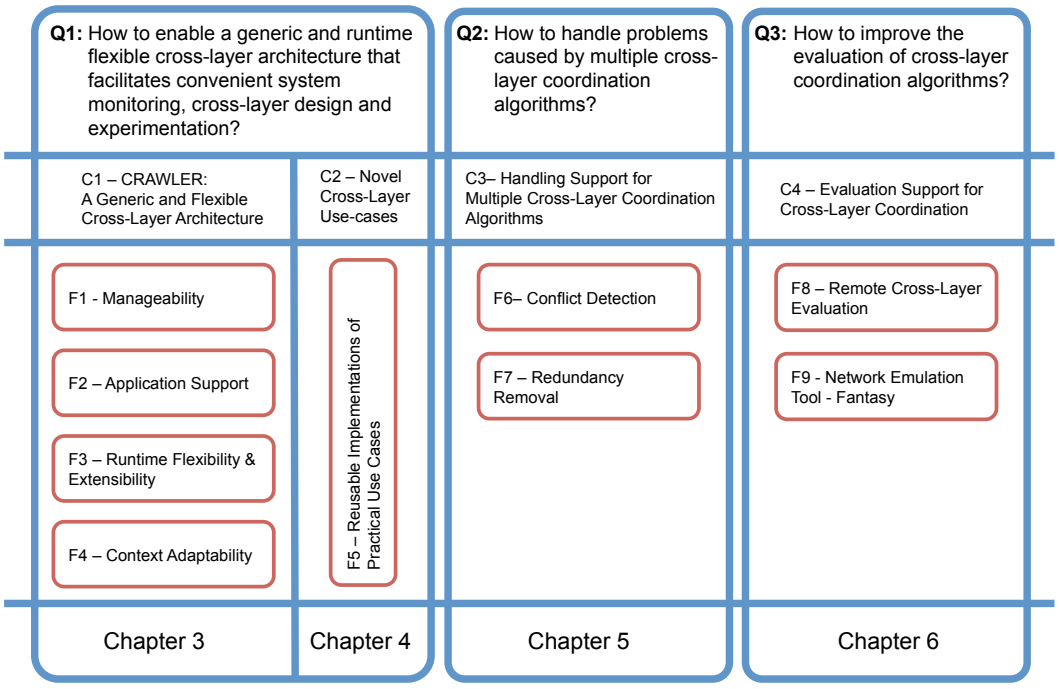


Figure 7.1 Overview presenting the relationship between research questions, contributions and CRAWLER's key features.

the declarative approach, which allows specifying cross-layer coordination algorithms at a high level of abstraction. None of the existing architectures simplify the specification of cross-layer coordination algorithms to a degree where even developers who are not experts in system or cross-layer development can describe cross-layer coordination algorithms.

Feature F2 - Application Support

Unlike existing approaches, CRAWLER provides a unified interface for application developers to (i) specify and add their own monitoring needs and cross-layer coordination algorithms into the system, and (ii) bundle these coordination algorithms with their applications, without needing to deal with OS level details. Moreover, it simplifies the process of accessing protocol and system information, typically placed in the OS, which today is limited to only a few interfaces and thus requires manual inspection and adaptation of the very large OS code base.

Feature F3 - Runtime Flexibility and Extensibility

CRAWLER offers flexibility which is essential for adjusting and experimenting with different sets of cross-layer coordination algorithms, and further, extensibility for involving a wide range of possible protocols and system components. In particular, when designing a cross-layer coordination, the exchange of information among any number of layers and system components, and the composition of any number of specific cross-layer coordination algorithms are adaptable at runtime.

Feature F4 - Context Adaptability

CRAWLER offers the ability (i) to detect underlying environmental changes, and (ii) to react to these changes by automatically loading the prespecified

set of cross-layer coordination algorithms at runtime. For example, exchanging a energy saving coordination algorithm with a high performing coordination algorithm if the device is plugged-in to a power supply. When specified, Crawler allows the detection of the defined conditions (e.g., plugged-in to power), unloading the set of unnecessary coordination algorithms (e.g., energy-aware) and loading the adequate set of coordination algorithms (e.g., better performing but energy-consuming).

Contribution C2 – Novel Cross-Layer Use-Cases

Our contribution of novel cross-layer use-cases (C1) demonstrates the applicability of Crawler in diverse fields of networking and also provides answers to our first research question Q1 by providing the following key feature:

Feature F5 - Reusable Implementations of Practical Use Cases

By using Crawler we showed its utility as framework for the networking research community. We implemented five use-cases from different research fields. The code-basis of the use-cases can be partially used to develop completely new use-cases or to improve the existing use-cases. In particular, we conducted experiments and proposed cross-layer coordination algorithms to (i) manipulate TCP's sliding window and application behavior in wireless environments, (ii) switch TCP congestion control algorithm based on wireless conditions, (iii) adapt the used VoIP codec to given network conditions, (iv) enable a framework for jamming detection and reaction and (v) enable an advanced machine learning-based jamming detection approach. In each of the use-cases, we first highlighted Crawler's monitoring capability to analyze and understand protocol and application behavior under given network conditions. Based on the observations, we showed how to conveniently formulate a cross-layer coordination algorithm on a very abstract level by using Crawler's configuration language. Moreover, dependent on the use-case, we presented how to experiment with cross-layer coordination algorithms and utilize Crawler's monitoring power to demonstrate the achieved flexibility or the relative performance improvement.

Contribution C3 – Handling Support for Multiple Cross-layer Coordination Algorithms

Our contribution C3 enhances Crawler to support developers to tackle problems involved with multiple running cross-layer coordination algorithms and accordingly answers our second research question Q2 by providing the following key features:

Feature F6 - Cross-Layer Conflict Detection

Although each single cross-layer coordination improves the system performance, running multiple cross-layer coordination algorithms in parallel could lead to contradicting goals resulting in severe performance degradations. We have classified cross-layer conflicts and proposed a solution to give developers feedback about conflicting cross-layer coordination algorithms early in the

experimentation phase. In this context, CRAWLER's monitoring capability further supports developers to analyze and understand cross-layer coordination algorithms and their effects on the system.

Feature F7 – Cross-Layer Redundancy Removal

Giving application developers the freedom to add their own cross-layer coordination algorithms into the system without knowledge about existing ones might lead to redundancy of cross-layer coordination algorithms. However, when many developers implement such cross-layer coordination algorithms independently, the overall resulting cross-layer coordination algorithms can become suboptimal as some of the added algorithms are already in the system and have redundant instructions utilizing more CPU and memory than necessary. CRAWLER facilitates to automatically detect and resolve such redundancies without developer interaction.

Contribution C4 – Cross-layer Evaluation Support

The research question Q3 is answered by our contribution C4, which extends CRAWLER to support developers in the experimentation phase of their cross-layer coordination algorithms by providing the following key features:

Feature F8 - Remote Cross-Layer Evaluation

Testing and validating cross-layer coordination algorithms typically require excessive effort due to the following reasons. First, installing the considered cross-layer coordination algorithms and further helper programs (e.g., iperf and tcpdump) is tedious. Second, the timely coordination of cross-layer coordination algorithms, applications, and helper programs on different devices requires manual developer intervention. Third, implementing mechanisms to monitor and log different parameters in the system (preferably while running) is difficult due to operation system restrictions. To support a developer in this process, we have extended CRAWLER to remotely (i) allow the automation of test runs on different devices, (ii) add, remove and modify cross-layer coordination algorithms at runtime and (iii) monitor and log different parameters in the running system.

Feature F9 - Network Emulation Tool

Testing cross-layer coordination algorithms in wireless environments usually requires many test runs. One reason, amongst others, is that the gain of the specific cross-layer coordination is difficult to identify as the performance difference might be influenced, for example, by the novel developed cross-layer coordination algorithm and / or due to changing channel conditions. Therefore, we coupled CRAWLER with a simulation framework (ns-3) resulting in a network emulation tool that allows the automated setup and execution of an experiment in a controllable simulation environment to improve the monitoring and analysis of cross-layer coordination algorithms and their behavior.

In the following we discuss possible future directions to enhance CRAWLER.

7.2 Future Work

Based on the achieved state of Crawler, we discuss limitations and suggest potential extensions of the contributions made in this work.

7.2.1 Increasing the Toolbox of Reusable Functional Units and Stubs

Crawler's abstraction language allows to describe cross-layer coordination algorithms on a high level of abstraction. The power of Crawler's abstract language is determined by the amount of functional units (FUs) and stubs. Stubs provide read and write access to protocol and system information. They act as a glue element between the cross-layer coordination algorithms and the operating system. Crawler currently offers about 20 FUs and 100 stubs, with the numbers growing with every new sample scenario. In Appendix B we give an overview about available stubs and FUs. Due to the reusable nature of FUs and stubs and the fact that it enriches Crawler's abstraction language, developers will highly benefit from increasing the number and variations of FUs.

7.2.2 Conflict Resolution

Crawler supports developers during their experimentation with cross-layer coordination algorithms. We especially enabled a feature for developers to detect cross-layer conflicts in the system. We classified the problem in direct and indirect conflicts. The former case is rather simple and we provide already few techniques to tackle the problem, but the latter is complex and requires further investigation to support developers to explore the right set of cross-layer coordination algorithms. In our approach, after detection of an indirect conflict, we inform the applications about the presence of a misbehavior. Although such a notification is helpful, ideally conflicts should be resolved automatically. We believe that without additional semantic knowledge this problem becomes more challenging. It is similar to expecting from a debugger to find programming faults without interaction and to fix them accordingly. Similar to program analysis, resolution of conflicts requires a deep understanding of the semantic of all running cross-layer coordination algorithms in the system. On the other hand, providing semantic knowledge is tedious and will slow down the development process. As we opt for a practical solution, we believe that providing hints about misbehavior, similar to what debuggers do, is the right way towards proper developer support.

7.2.3 Timing Constraints

The monitoring of variables in a (remote) system and the exchange of information between applications and Crawler require several steps of processing which influence the processing time. The crucial factors that influence the processing time are highlighted in Figure 7.2 where we monitor variables on a remote system.

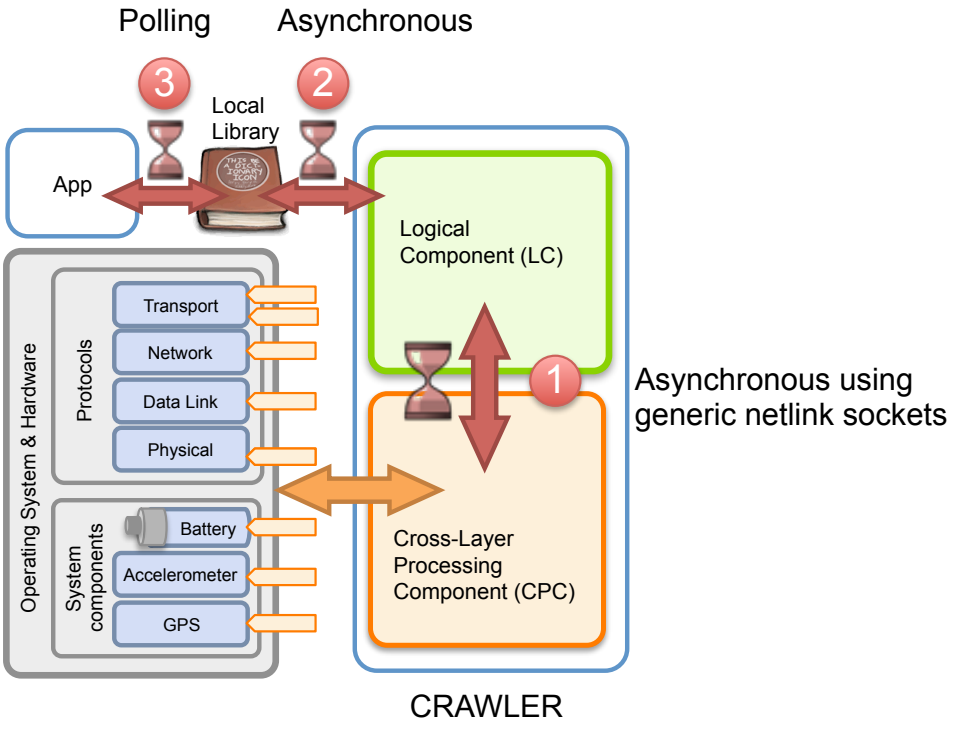


Figure 7.2 Three potential sources for delay when interacting with CRAWLER: (i) the asynchronous communication (caused by the generic netlink sockets) between the LC (user space) and the CPC (kernel space) as indicated by ①, (ii) the communication between the shared library and CRAWLER's LC as indicted by ②, and (iii) the interaction between the shared library and the application as indicated by ③.

The requested variables (e.g., TCP's congestion window) for monitoring or exchange have to be accessed in the kernel space of the system. In CRAWLER the requested information is accessed by stubs which provide the information to the cross-layer processing component (CPC) running in the kernel space. Subsequently, this information is provided to the user space daemon of CRAWLER which we refer to as logical component (LC). In CRAWLER's implementation, information between the CPC and the LC is exchanged using generic netlink sockets (cf. ① in Figure 7.2). Generic netlink sockets exchange information asynchronously resulting in a potential delay. The delay depends on the workload and the computational power of the device. We opted for generic net link sockets due to their potential to exchange any type of information and its adaptability to our needs. Other few alternatives were not able to offer the same degree of adaptability [NAGL10]. However, we suggest to avoid synchronous communication between the CPC (running in kernel space) and the LC (running in user space) as it can lead to blocking of components and thus avoids their processing causing performance degradation. Another source for delay occurs for the asynchronous communication between the daemon and the (local) shared library (cf. ② in Figure 7.2). Establishing synchronous communication could be an option. In a next step, the event timer in the shared library polls for application variables (cf. ③ in Figure 7.2). The polling intervals are set individually for each application and its variables are parameterized on application registration. If not set properly this could lead to either too much computational overhead or high delays. Accordingly, providing an event-based mechanism for the interaction between the shared library and the application will be beneficial. However, in a

local Ethernet network (for the use-case as described in Section 6.3.4), we used our remote monitoring feature to remotely monitor the RSSI and observed a delay of around one to two seconds including the network delay.

7.2.4 Potential Use-Cases

We have realized different use-cases from diverse fields of networking and with each use-case we have further extended Crawler, its FU toolbox and the number of stubs.

For example, we have started to realize an energy saving use-case. The idea is to control the on and off state of a wireless device depending on the availability of application traffic. We distinguish between two types of application traffic where one type is delay-tolerant and the other type is delay-sensitive. Applications can inform Crawler about the type of traffic they are currently generating. If all running applications are in the delay-tolerant mode, Crawler powers off the wireless device to save energy until enough data has accumulated in the network stack for transmission. Once enough delay-tolerant data has accumulated, Crawler signals the applications to hand over their data to the network stack. Similarly, if any of the applications is in delay-sensitive mode, the wireless device is not powered off to ensure that data is transmitted immediately. Consequently, the delay-tolerant data from other applications also benefits from the powered device and is sent out immediately. This example shows that applications and the system can make joint adaptation decisions by exchanging information. Using this coordination algorithm, it is possible to save energy over longer periods of time.

Another area of research using the cross-layer paradigm is mobility. Device mobility could affect multiple layers, for instance, layer two and three where not only the location or association to an access point is changed (affects layer 2) but also the domain which necessitates a new IP address assignment (affects layer 3). In such a case, layer two and three protocols have to be coordinated. Many research effort has been put to tackle the mobility related problems. It would be interesting to analyze these proposals and to enhance the coordination process with Crawler.

We have investigated the research field of jamming detection and reaction in IEEE 802.11 networks. However, the design of our machine-learning-based jamming detection approach is not coupled a particular wireless technology. Thus, it should be easily applicable to other wireless technologies such as LTE networks. It would be interesting to explore and integrate novel metrics gained from LTE drivers into Crawler and to experiment with their reaction to jamming.

To conclude, monitoring and experimentation with Crawler in already known networking fields and also in completely new areas by using established and new wireless technologies would be intresting.

7.2.5 Realization on Further Platforms

With the emergence of new operating system for smart phones such as iOS and Android, many new applications have appeared. Due to the demand of these applications, operating systems have extended their APIs to access system component

information such as to sensor information (e.g., acceleration and orientation) and to localization information (e.g., GPS coordinates). Since smartphones are increasingly used in the networking research community, it would be beneficial to implement Crawler for an operating system such as Android. In particular, it would be interesting to analyze how Crawler could complement the existing APIs by also providing access to protocol information.

Glossary

ANI	Ambient Noise Immunity
API	Application Programming Interface
CBR	Channel Busy Ratio
CCA	Clear Channel Assessment
CDF	Cumulative Distribution Function
CPC	Cross-Layer Processing Component
CPU	Central Processing Unit
CRC	Cyclic Redundancy Checksum
CWND	Congestion Window
FU	Functional Unit
GCU	Guest Configuration Unit
GPS	Global Positioning System
HCU	Host Configuration Unit
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
IPC	Inter-process Communication
IT	Inactive Time
LC	Logical Component
MAC	Medium Access Control
MTU	Maximum Transmission Unit
NIC	Network Interface Card
OFDM	Orthogonal Frequency-Division Multiplexing
OS	Operating System
PCM	Pulse Code Modulation
PDR	Packet Delivery Ratio
PHY	Physical Layer
PLCP	Physical Layer Convergence Protocol
PLR	Packet Loss Rate
QoE	Quality of Experience
QoS	Quality of Service
RFC	Request for Comments
RSS	Received Signal Strength
RSSI	Received Signal Strength Indicator
RTP	Real-Time Transport Protocol
RTT	Round Trip Time
SIP	Session Initiation Protocol
TCP	Transmission Control Protocol
TN	True Negative
TP	True Positive
TTL	Time To Live
UDP	User Datagram Protocol
VM	Virtual Machine
VoIP	Voice over IP
WLAN	Wireless Local Area Network

Bibliography

[AAS+14] Ismet Aktas, Muhammad Hamad Alizai, Florian Schmidt, Hanno Wirtz, and Klaus Wehrle, *Harnessing cross-layer-design*, Ad Hoc Networks (2014), pp. 444–461.

[Abi13] Gloria Abidin, *Machine Learning-Based Jamming Detection in WLAN*, Master Thesis, Communication and Distributed Systems, RWTH Aachen University, November 2013.

[ACLL06] Hervé Aïache, Vania Conan, Jérémie Leguay, and Mikaël Levy, *Xian: Cross-layer interface for wireless ad hoc networks*, Mediterranean Ad-hoc Networking Workshop (Med-Hoc-Net), 2006, pp. 1–8.

[AHA+14] Ismet Aktas, Martin Henze, Muhammed Hamad Alizai, Kevin Möllering, and Klaus Wehrle, *Graph-based Redundancy Removal Approach for Multiple Cross-Layer Interactions*, Proceedings of the 6th International Conference on Communication Systems and Networks (COMSNETS), January 2014, pp. 1–8.

[AOSW10] Ismet Aktas, Jens Otten, Florian Schmidt, and Klaus Wehrle, *Towards a Flexible and Versatile Cross-Layer-Coordination Architecture*, Proceedings of the 29th International Conference on Computer Communications (INFOCOM), March 2010, pp. 1–5.

[APS+14a] I. Aktas, I. Punal, F. Schmidt, M.H. Alizai, T. Druner, and K. Wehrle, *Machine learning-based jamming detection for IEEE 802.11: Design and experimental evaluation*, IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014, pp. 1–9.

[APS+14b] Ismet Aktas, Oscar Punal, Florian Schmidt, Tobias Drüner, and Klaus Wehrle, *A Framework for Remote Automation, Configuration, and Monitoring of Real-World Experiments*, 9th ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization (WINTECH), September 2014, pp. 1–8.

[AR99] AV Arunachalam and JH Reed, *Quality of service (QoS) classes for bwa*, A contribution to the IEEE 802, 1999, pp. 1–10.

[ASA+12] I. Aktas, F. Schmidt, M.H. Alizai, T. Druner, and K. Wehrle, *Crawler: An experimentation platform for system monitoring and cross-layer-coordination*, IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM), June 2012, pp. 1–9.

[ASW+12] I. Aktas, F. Schmidt, E. Weingärtner, C.J. Schnelke, and K. Wehrle, *An adaptive codec switching scheme for sip-based voip*, Internet of Things, Smart Spaces, and Next Generation Networking, August 2012, pp. 347–358.

[ath] *Ath9k - Linux Wireless: Official Website*, `http://wireless.kernel.org/en/users/Drivers/ath9k`, Last visit 28-7-2013.

[AVA06] Ian F Akyildiz, Mehmet C Vuran, and Ozgur B Akan, *A cross-layer protocol for wireless sensor networks*, Information Sciences and Systems, 2006 40th Annual Conference on, IEEE, 2006, pp. 1102–1107.

[AvLH+12] Ismet Aktaş, Hendrik vom Lehn, Christoph Habets, Florian Schmidt, and Klaus Wehrle, *Fantasy: fully automatic network emulation architecture with cross-layer support*, Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques (SimuTools), 2012, pp. 57–64.

[AXM04] Ian F Akyildiz, Jiang Xie, and Shantidev Mohanty, *A survey of mobility management in next-generation all-ip-based wireless systems*, IEEE Wireless Communications, 2004, no. 4, pp. 16–28.

[BB02] Sabine Bachl and Franz-Josef Brandenburg, *Computing and drawing isomorphic subgraphs*, Graph Drawing (Michael Goodrich and Stephen Kobourov, eds.), Lecture Notes in Computer Science, vol. 2528, Springer Berlin / Heidelberg, 2002, 10.1007/3-540-36151-0_8, pp. 74–85.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Adrew Warfield, *Xen and the art of virtualization*, Proc. of SOSP, Bolton Landing, NY, USA, ACM, October 2003.

[BES05] D. Ben-Eli and I. Sutskover, *Transmitter operations for interference mitigation*, July 21 2005, WO Patent App. PCT/US2004/043,039.

[BFH03] R. Braden, T. Faber, and M. Handley, *From protocol stack to protocol heap: role-based architecture*, ACM SIGCOMM Computer Communication Review, 2003, no. 1, 17–22.

[BHS07] F. Buschmann, K. Henney, and D.C. Schmidt, *Pattern-oriented software architecture: On patterns and pattern languages*, John Wiley & Sons Inc, 2007.

[BKL+08] E. Bayraktaroglu, C. King, X. Liu, G. Noubir, R. Rajaraman, and B. Thapa, *On the Performance of IEEE 802.11 under Jamming*, Proc. IEEE INFOCOM, 2008.

[BMS+06] Didier Bourse, Markus Muck, Olivier Simon, Nancy Alonistioti, Klaus Moessner, Eric Nicollet, David Bateman, Enrico Buracchini, Gemini Chengeleroyen, and Panagiotis Demestichas, *End-to-end reconfigurability (e2r ii): Management and control of adaptive communication systems*, IST Mobile Summit, 2006.

[BP95] Lawrence S. Brakmo and Larry L. Peterson, *Tcp vegas: End to end congestion avoidance on a global internet*, IEEE Journal on Selected Areas in Communications, 1995, no. 8, pp. 1465–1480.

[BPY09] Wafa Berrayana, Guy Pujolle, and Habib Youssef, *Xlengine: a cross-layer autonomic architecture with network wide knowledge for qos support in wireless networks*, Proceedings of the 2009 International Conference on Wireless Communications and Mobile Computing: Connecting the World Wirelessly, ACM, 2009, pp. 170–175.

[Bre01] L. Breiman, *Random Forests*, Tech. report, University of California at Berkeley, 2001.

[BRN+10] Nicola Baldo, Manuel Requena, Jose Nunez, Marc Portoles, Jaume Nin, Paolo Dini, and Josep Mangues, *Validation of the ns-3 IEEE 802.11 model using the EXTREME testbed*, Proceedings of SIMUTools Conference, 2010, March 2010.

[BSAK95] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H Katz, *Improving tcp/ip performance over wireless networks*, MobiCom, vol. 95, Citeseer, 1995, pp. 2–11.

[BSK95] Hari Balakrishnan, Srinivasan Seshan, and Randy H. Katz, *Improving reliable transport and handoff performance in cellular wireless networks*, Wireless Networks, 1995, pp. 469–481.

[CC08] Ben-Jye Chang and Jun-Fu Chen, *Cross-layer-based adaptive vertical handoff with predictive rss in heterogeneous wireless networks*, IEEE Transactions on Vehicular Technology, 2008, no. 6, 3679–3692.

[CDMM00] C. Casetti, J. C. De Martin, and M. Meo, *Framework for the analysis of adaptive voice over ip*, IEEE International Conference on Communications, June 2000, pp. 821 – 826.

[CGM+02] Claudio Casetti, Mario Gerla, Saverio Mascolo, MY Sanadidi, and Ren Wang, *Tcp westwood: end-to-end congestion control for wired/wireless networks*, Wireless Networks, 2002, no. 5, pp. 467–479.

[Cit] Citrix Systems, Inc., *Xen hypervisor, the powerful open source industry standard for virtualization.*, `http://www.xen.org/`, (accessed Nov 14, 2011).

[CMTG04] M. Conti, G. Maselli, G. Turi, and S. Giordano, *Cross-layering in mobile ad hoc network design*, Computer, 2004, no. 2, pp. 48–51.

[CRRP04] G. Carneiro, J. Ruela, M. Ricardo, and I. Porto, *Cross-layer design in 4G wireless terminals*, IEEE Wireless Communication, 2004, no. 2, pp. 7–13.

[CSN02] K. Chen, S.H. Shah, and K. Nahrstedt, *Cross-layer design for data accessibility in mobile ad hoc networks*, Wireless Personal Communications, 2002, no. 1, pp. 49–76.

[DCY93] Antonio DeSimone, Mooi Choo Chuah, and On-Ching Yue, *Throughput performance of transport-layer protocols over wireless lans*, Global Telecommunications Conference, 1993, including a Communications Theory Mini-Conference. Technical Program Conference Record, IEEE in Houston. GLOBECOM'93., IEEE, IEEE, 1993, pp. 542–549.

[Den09] Dominik Dennisen, *Integration of a WiFi Tweaking component into the X-Layer Architecture*, Bachelor Thesis, Communication and Distributed Systems, RWTH Aachen University, October 2009.

[Den12] Dominik Dennissen, *Coping with Jamming Attacks with the Help of Cross-Layer Design*, Master Thesis, Communication and Distributed Systems, RWTH Aachen University, August 2012.

[dJ] Irmen de Jong, *Pyro 3.x - Python remote objects*, `http://www.xs4all.nl/~irmen/pyro3/`, (accessed Nov 14, 2011).

[Dör07] Kay Dörnemann, *Ad-Hoc Routing/Emulation - Philipps-Universität Marburg - Verteilte Systeme (AG Freisleben)*, `http://www.uni-marburg.de/fb12/verteilte_systeme/forschung/pastproj/adhoc_routing_emul`, 2007, (accessed Nov 14, 2011).

[DRSC05] P. De, A. Raniwala, S. Sharma, and T. Chiueh, *MiNT: a miniaturized network testbed for mobile wireless research*, Proc. IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies INFOCOM 2005, vol. 4, 2005, pp. 2731–2742 vol. 4.

[Drü10] Tobias Drüner, *Reconfigurability of Application-driven Cross-Layer Optimizations*, Bachelor Thesis, Communication and Distributed Systems, RWTH Aachen University, July 2010.

[Drü13] Tobias Drüner, *Remote Cross-Layer Evaluation*, Master Thesis, Communication and Distributed Systems, RWTH Aachen University, December 2013.

[DSB+06] Panagiotis Demestichas, Vera Stavroulaki, Dragan Boscovic, Al Lee, and John Strassner, *m@angel: autonomic management platform for seamless cognitive connectivity to the mobile internet*, IEEE Communications Magazine, 2006, no. 6, pp. 118–127.

[EBPC05] Wolfgang Eberle, Bruno Bougard, Sofie Pollin, and Francky Catthoor, *From myth to methodology: cross-layer design for energy-efficient wireless communication*, Design Automation Conference, 2005. Proceedings. 42nd, IEEE, 2005, pp. 303–308.

[Emu] *Emulab.Net - Emulab - Network Emulation Testbed Home*, `http://www.emulab.net/`, (accessed Nov 14, 2011).

[Fal99] Kevin R. Fall, *Network emulation in the Vint/NS simulator*, 4th IEEE Symposium on Computers and Communication, 1999.

[FGA08] Fotis Foukalas, Vangelis Gazis, and Nancy Alonistioti, *Cross-layer design proposals for wireless mobile networks: a survey and taxonomy*, Communications Surveys & Tutorials, IEEE, 2008, no. 1, pp. 70–85.

[FL03] Cheng Peng Fu and Soung C Liew, *Tcp veno: Tcp enhancement for transmission over wireless access networks*, IEEE Journal on Selected Areas in Communications, 2003, no. 2, pp. 216–228.

[For02] Behrouz A Forouzan, *Tcp/ip protocol suite*, McGraw-Hill, Inc., 2002.

[GFTW06] Xiaoyuan Gu, Xiaoming Fu, Hannes Tschofenig, and Lars Wolf, *Towards self-optimizing protocol stack for autonomic communication: initial experience*, Autonomic Communication, Springer, 2006, pp. 186–201.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides, *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley, 1995.

[GK02] P. Godefroid and S. Khurshid, *Exploring very large state spaces using genetic algorithms*, Tools and Algorithms for the Construction and Analysis of Systems, 2002, pp. 71–82.

[GKN+04] Robert S. Gray, David Kotz, Calvin Newport, Nikita Dubrovsky, Aaron Fiske, Jason Liu, Christopher Masone, Susan McGrath, and Yougu Yuan, *Outdoor experimental comparison of four ad hoc routing algorithms*, Proceedings of the ACM/IEEE International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM), 2004.

[GLS+13] D. Giustiniano, V. Lenders, J. Schmitt, M. Spuhler, and M. Wilhelm, *Detection of Reactive Jamming in DSSS-based Wireless Networks*, Proc. ACM WiSec, 2013.

[GMPG00] T. Goff, J. Moronski, D.S. Phatak, and V. Gupta, *Freeze-tcp: a true end-to-end tcp enhancement mechanism for mobile environments*, INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, vol. 3, mar 2000, pp. 1537 –1545 vol.3.

[GWCA11] S. Götz, C. Wilke, S. Cech, and U. Aßmann, *Runtime variability management for energy-efficient software by contract negotiation*, Proceedings of the International Workshop on Models@ run. time, 2011.

[GWGS07] Ramakrishna Gummadi, David Wetherall, Ben Greenstein, and Srinivasan Seshan, *Understanding and Mitigating the Impact of RF Interference on 802.11 Networks*, Proc. ACM SIGCOMM, 2007.

[Hab11] Jan Christoph Habets, *Fully Automated Network Emulation Setup for Protocol Evaluation*, Diploma Thesis, Communication and Distributed Systems, RWTH Aachen University, July 2011.

[HBOM09] Ali Hamieh, Jalel Ben-Othman, and Lynda Mokdad, *Detection of Radio Interference Attacks in VANET*, Proc. of GLOBECOM, 2009.

[Hem05] S. Hemminger, *Netem-emulating real networks in the lab*, Proc. Linux Conference Australia, 2005.

[HHT+02] Matti Hamalainen, Veikko Hovinen, Raffaello Tesi, Jari HJ Iinatti, and Matti Latva-aho, *On the uwb system coexistence with gsm900, umts/wcdma, and gps*, IEEE Journal on Selected Areas in Communications, 2002, no. 9, pp. 1712–1721.

[HP91] N.C. Hutchinson and L.L. Peterson, *The x-kernel: An architecture for implementing network protocols*, IEEE Transactions on Software Engineering, 1991, no. 1, pp. 64–76.

[Hro03] Juraj Hromkovic, *Theoretical computer science : Introduction to automata, computability, complexity, algorithmics, randomization, communication, and cryptography*, Springer, November 2003.

[HRX08] Sangtae Ha, Injong Rhee, and Lisong Xu, *CUBIC: a new TCP-friendly high-speed TCP variant*, SIGOPS Oper. Syst. Rev., 2008, no. 5, pp. 64–74.

[IBW97] J. Inouye, J. Binkley, and J. Walpole, *Dynamic network reconfiguration support for mobile computers*, Proc. MobiCom, ACM New York, NY, USA, 1997, pp. 13–22.

[IKSF04] Luigi Iannone, Ramin Khalili, Kave Salamatian, and Serge Fdida, *Cross-layer routing in wireless mesh networks*, Wireless Communication Systems, 2004, 1st International Symposium on, IEEE, 2004, pp. 319–323.

[ITU01] *ITU-T P.862 : Perceptual evaluation of speech quality (pesq): An objective method for end-to-end speech quality assessment of narrow-band telephone networks and speech codecs*, 2001.

[JCS] R. Jones, K. Choy, and D. Shield, *Netperf*, `http://www.netperf.org`, (accessed Nov 14, 2011).

[JSF+06] David Johnson, Tim Stack, Russ Fish, Daniel Montrallo Flickinger, Leigh Stoller, Robert Ricci, and Jay Lepreau, *Mobile Emulab: A Robotic Wireless and Sensor Network Testbed*, Proceedings of the 25th Conference on Computer Communications (IEEE INFOCOM 2006), April 2006.

[KCH+08] Ahmed Khattab, Joseph Camp, Chris Hunter, Patrick Murphy, Ashutosh Sabharwal, and Edward W Knightly, *WARP: A Flexible Platform for Clean-Slate Wireless Medium Access Protocol Design*, ACM SIGMOBILE Mobile Computing and Communications Review, 2008, no. 1, pp. 56–58.

[KHZ+03] Wuttipong Kumwilaisak, Yiwei Thomas Hou, Qian Zhang, Wenwu Zhu, C-CJ Kuo, and Ya-Qin Zhang, *A cross-layer quality-of-service mapping architecture for video delivery in wireless networks*, Selected Areas in Communications, IEEE Journal on, 2003, no. 10, pp. 1685–1698.

[KKT04] Ulas C Kozat, Iordanis Koutsopoulos, and Leandros Tassiulas, *A framework for cross-layer design of energy-efficient communication with qos provisioning in multi-hop wireless networks*, INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies, vol. 2, IEEE, 2004, pp. 1446–1456.

[KKTC05] V. Kawadia, PR Kumar, BBN Technol, and MA Cambridge, *A cautionary perspective on cross-layer design*, IEEE Wireless Communications, 2005, no. 1, pp. 3–11.

[KMC+00] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek, *The click modular router*, ACM Transactions on Computer Systems (TOCS), 2000, no. 3, pp. 263–297.

[KNB+04] Raymond Knopp, Navid Nikaein, Christian Bonnet, Hervé Aiache, Vania Conan, Sandrine Masson, Gregoire Guibe, and CL Martret, *Overview of the widens architecture, a wireless ad hoc network for public safety*, 1st IEEE International Conference on Sensor and Ad Hoc Communications and Networks (SECON), Santa Clara, USA, 2004.

[Koe11] Nikolaus Koemm, *Cross-Layer Conflict Detection*, Diploma Thesis, Communication and Distributed Systems, RWTH Aachen University, December 2011.

[KP09] Stylianos Karapantazis and Fotini-Niovi Pavlidou, *Voip: A comprehensive survey on a promising technology*, Computer Networks, 2009, no. 12, pp. 2050–2090.

[KPS+06] S. Khan, Y. Peng, E. Steinbach, M. Sgroi, and W. Kellerer, *Application-driven cross-layer optimization for video streaming over wireless networks*, IEEE Communications Magazine, 2006, no. 1, pp. 122–130.

[KSHH04] Teemu Karhima, Aki Silvennoinen, Michael Hall, and S-G Haggman, *Ieee 802.11 b/g wlan tolerance to jamming*, Military Communications Conference, 2004. MILCOM 2004. 2004 IEEE, vol. 3, IEEE, 2004, pp. 1364–1370.

[KV13] B Yeshwanth Kumar and T Vishal, *Internet protocol version 6 internet protocol version 6 (ipv6)*, International Journal of Research in Engineering and Advanced Technology (IJREAT), 2013, pp. 1–2.

[LCG+09] Chi Harold Liu, Sara Grilli Colombo, Athanasios Gkelias, Erwu Liu, and Kin K. Leung, *An Efficient Cross-Layer Simulation Architecture for Wireless Mesh Networks*, Proceedings of IEEE UKSim 2009, March 2009, pp. 491–496.

[LFH+10] Mathieu Lacage, Martin Ferrari, Mads Hansen, Thierry Turletti, and Walid Dabbous, *NEPI: Using Independent Simulators, Emulators, and Testbeds for Easy Experimentation*, ACM SIGOPS Operating Systems Review, January 2010, pp. 60–65.

[lin] *Linphone*, [Online; accessed 11-April-2010].

[LKP07] Mingyan Li, I. Koutsopoulos, and R. Poovendran, *Optimal jamming attacks and network defense policies in wireless sensor networks*, INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE, may 2007, pp. 1307 –1315.

[LSS06] X. Lin, NB Shroff, and R. Srikant, *A tutorial on cross-layer optimization in wireless networks*, IEEE Journal on Selected Areas in Communications, 2006, no. 8, pp. 1452–1463.

[LSVW11] Jó Agila Bitsch Link, Paul Smith, Nicolai Viol, and Klaus Wehrle, *Footpath: Accurate map-based indoor navigation using smartphones*, Indoor Positioning and Indoor Navigation (IPIN), 2011 International Conference on, IEEE, 2011, pp. 1–8.

[Ltd08] S. Ltd, *Skype Public API*, 2008.

[MA06] Shantidev Mohanty and Ian F Akyildiz, *A cross-layer (layer 2+ 3) handoff management protocol for next-generation wireless systems*, Mobile Computing, IEEE Transactions on, 2006, no. 10, pp. 1347–1360.

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner, *OpenFlow: Enabling Innovation in Campus Networks*, ACM SIGCOMM Computer Communication Review, 2008, no. 2, pp. 69–74.

[MCC04] Matthew L Massie, Brent N Chun, and David E Culler, *The Ganglia Distributed Monitoring System: Design, Implementation, and Experience*, Parallel Computing, 2004, no. 7, pp. 817–840.

[MCG+01] Saverio Mascolo, Claudio Casetti, Mario Gerla, M. Y. Sanadidi, and Ren Wang, *TCP westwood: Bandwidth estimation for enhanced transport over wireless links*, Proc. MobiCom'01 (New York, NY, USA), ACM, 2001, pp. 287–297.

[MHLS09] Guowang Miao, Nageen Himayat, Ye Geoffrey Li, and Ananthram Swami, *Cross-layer optimization for energy-efficient wireless communications: a survey*, Wireless Communications and Mobile Computing, 2009, no. 4, pp. 529–542.

[Mit00] Joseph Mitola, *Cognitive radio—an integrated agent architecture for software defined radio*, Ph.D. thesis, Royal Institute of Technology (KTH), 2000.

[MK07] W. Mazurczyk and Z. Kotulski, *Adaptive voip with audio watermarking for improved call quality and security*, Journal of Information Assurance and Security, 2007, no. 3, pp. 226–234.

[MLAW99] Jeonghoon Mo, Richard J La, Venkat Anantharam, and Jean Walrand, *Analysis and comparison of tcp reno and vegas*, INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, vol. 3, IEEE, 1999, pp. 1556–1563.

[MLM+05] Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Jorg Hahner, Robert Sauter, and Kurt Rothermel, *Tinycubus: a flexible and adaptive framework sensor networks*, Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on, IEEE, 2005, pp. 278–289.

[MMJ99] Joseph Mitola and Gerald Q Maguire Jr, *Cognitive radio: making software radios more personal*, Personal Communications, IEEE, 1999, no. 4, pp. 13–18.

[Möl11] Kevin Möllering, *Establishing Synergies between Cross-Layer Optimizations*, Bachelor Thesis, Communication and Distributed Systems, RWTH Aachen University, June 2011.

[MPRW06] Petri Mähönen, Marina Petrova, Janne Riihijärvi, and Matthias Wellens, *Cognitive wireless networks: your network just became a teenager*, Proceedings of IEEE INFOCOM, vol. 2006, Citeseer, 2006.

[MS03] Ivo Maathuis and Wim A Smit, *The battle between standards: Tcp/ip vs osi victory through path dependency or by quality?*, Standardization and Innovation in Information Technology, 2003. The 3rd Conference on, IEEE, 2003, pp. 161–176.

[MVP06] Tommaso Melodia, Mehmet C Vuran, and Dario Pompili, *The state of the art in cross-layer design for wireless sensor networks*, Wireless Systems and Network Architectures in Next Generation Internet, Springer, 2006, pp. 78–92.

[MWH01] Martin Mauve, Jorg Widmer, and Hannes Hartenstein, *A survey on position-based routing in mobile ad hoc networks*, IEEE Network, 2001, no. 6, pp. 30–39.

[NAGL10] Pablo Neira-Ayuso, Rafael M Gasca, and Laurent Lefevre, *Communicating between the kernel and user-space in linux using netlink sockets*, Software: Practice and Experience, 2010, no. 9, pp. 797–810.

[ncu13] *NCurses (new curses) library*, `http://www.gnu.org/software/ncurses/ncurses.html`, 11 2013.

[net] *Netem (network emulation)*, `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`.

[NHS05] See Leng Ng, S. Hoh, and D. Singh, *Effectiveness of adaptive codec switching voip application over heterogeneous networks*, Mobile Technology, Applications and Systems, 2005 2nd International Conference on, November 2005, pp. 7.

[Noi] *Method and System for Noise Floor Calibration and Receive Signal Strength Detection (Atheros Patent)*, `http://www.patentstorm.us/patents/7245893/description.html`, Last visit 19-12-2013.

[ns3] *ns-3 Website*, `http://www.nsnam.org/`, (accessed Nov 14, 2011).

[Ott09] Jens Otten, *Design and Evaluation of a Dynamically Reconfigurable and Extendable Cross-Layer Signaling Architecture*, Diploma Thesis, Communication and Distributed Systems, RWTH Aachen University, May 2009.

[Pen00] Kostas Pentikousis, *Tcp in wired-cum-wireless environments*, Communications Surveys & Tutorials, IEEE, 2000, no. 4, pp. 2–14.

[pes] *Pesq*, `http://www.pesq.org/`.

[PIK11] K. Pelechrinis, M. Iliofotou, and S.V. Krishnamurthy, *Denial of service attacks in wireless networks: The case of jammers*, IEEE Communications Surveys Tutorials, 2011, no. 2, pp. 245–257.

[PnAG12] Oscar Puñal, Ana Aguiar, and James Gross, *In vanets we trust?: characterizing rf jamming in vehicular networks*, Proceedings of the ninth ACM international workshop on Vehicular inter-networking, systems, and applications (New York, NY, USA), VANET '12, ACM, 2012, pp. 83–92.

[PPS+09] H. Pham, J.M. Paluska, U. Saif, C. Stawarz, C. Terman, and S. Ward, *A dynamic platform for run-time adaptation*, Pervasive and Mobile Computing, 2009, no. 6, pp. 676–696.

[pyt] *Python Programming Language, Official Website*, `http://www.python.org/`, (accessed Nov 14, 2011).

[QK04] Liang Qin and Thomas Kunz, *Survey on mobile ad hoc network routing protocols and cross-layer design*, Tech. report, Carleton University Systems and Computer Engineering, 2004.

[Rai06] Vijay Thakurdas Raisinghani, *Cross layer feedback architecture for mobile device protocol stacks*, Ph.D. thesis, INDIAN INSTITUTE OF TECHNOLOGY, BOMBAY, 2006.

[RDN06] Mohammad A Razzaque, Simon Dobson, and Paddy Nixon, *A cross-layer architecture for autonomic communications*, Autonomic Networking, Springer, 2006, pp. 25–35.

[RDN07a] Mohammad Abdur Razzaque, Simon Dobson, and Paddy Nixon, *Context awareness through cross-layer network architecture*, Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on, IEEE, 2007, pp. 1076–1081.

[RDN07b] Mohammad Abdur Razzaque, Simon Dobson, and Paddy Nixon, *Cross-layer architectures for autonomic communications*, Journal of Network and Systems Management, 2007, no. 1, pp. 13–27.

[RI04a] V.T. Raisinghani and S. Iyer, *Cross-layer design optimizations in wireless protocol stacks*, Computer Communications, 2004, no. 8, pp. 720–724.

[RI04b] V.T. Raisinghani and S. Iyer, *ECLAIR: An efficient cross layer architecture for wireless protocol stacks*, Proc. World Wireless Congress, 2004.

[RI06] Vijay T Raisinghani and Sridhar Iyer, *Cross-layer feedback architecture for mobile device protocol stacks*, IEEE Communications Magazine, 2006.

[RNS13] SV Rani, P Narayanasamy, and J John Shiny, *A cross layer approach for improving TCP performance using channel access information*, Communications and Signal Processing (ICCSP), 2013 International Conference on, IEEE, 2013, pp. 587–591.

[RSC+02] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, *SIP: Session Initiation Protocol*, RFC 3261 (Proposed Standard), June 2002, Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630, 5922, 5954, 6026, 6141.

[RSI02] Vijay T Raisinghani, Ajay Kr Singh, and Sridhar Iyer, *Improving tcp performance over mobile wireless environments using cross layer feedback*, Personal Wireless Communications, 2002 IEEE International Conference on, IEEE, 2002, pp. 81–85.

[SB01] P. Sudame and BR Badrinath, *On providing support for protocol adaptation in mobile wireless networks*, MONET, 2001, no. 1, pp. 43–55.

[Sch11] Caj-Julian Schnelke, *An Adaptive Codec Switching Scheme for SIP-based VoIP*, Bachelor Thesis, Communication and Distributed Systems, RWTH Aachen University, May 2011.

[Sch13] Caj-Julian Schnelke, *Tackling Jamming Attacks in 802.11 Ad-hoc-networks*, Master Thesis, Communication and Distributed Systems, RWTH Aachen University, June 2013.

[SDN06] Paul Sutton, Linda E Doyle, and Keith E Nolan, *A reconfigurable platform for cognitive networks*, Cognitive Radio Oriented Wireless Networks and Communications, 2006. 1st International Conference on, IEEE, 2006, pp. 1–5.

[SDv10] Mario Strasser, Boris Danev, and Srdjan Čapkun, *Detection of reactive jamming in sensor networks*, ACM Trans. Sen. Netw., 2010, no. 2, 16:1–16:29.

[Shy06] DJ Shyy, *Military usage scenario and ieee 802.11 s mesh networking standard*, Military Communications Conference, 2006. MILCOM 2006. IEEE, IEEE, 2006, pp. 1–7.

[SKC05] C.M. Sadler, L. Kant, and W. Chen, *Cross-layer self-healing mechanisms in wireless networks*, Proc. WWC, vol. 254, 2005.

[SM05] V. Srivastava and M. Motani, *Cross-layer design: a survey and the road ahead*, IEEE Communications Magazine, 2005, no. 12, pp. 112–119.

[SPKW07] M. Schinnenburg, R. Pabst, K. Klagges, and B. Walke, *A software architecture for modular implementation of adaptive protocol stacks*, MMBnet Workshop 2007, Hamburg, Germany, Sep 2007, pp. 94–103.

[SRK03] S. Shakkottai, TS Rappaport, and PC Karlsson, *Cross-layer design for wireless networks*, IEEE Comm. Magazine, 2003, no. 10, pp. 74–80.

[SSC11] Chowdhury Shahriar, Shabnam Sodagari, and T Charles Clancy, *Physical-Layer Security Challenges of DSA-Enabled TD-LTE*, Proc. ACM CogART, 2011.

[ST09] M. Salehie and L. Tahvildari, *Self-adaptive software: Landscape and research challenges*, ACM Transactions on Autonomous and Adaptive Systems (TAAS), 2009, no. 2, 14.

[Stå00] Mika Ståhlberg, *Radio jamming attacks against two popular mobile networks*, Helsinki University of Technology Seminar on Network Security, 2000.

[STE97] WR STEVENS, *Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms*, Internet Draft RFC 2001 (1997), pp. 1–5.

[SYH+04] Daniel Grobe Sachs, Wanghong Yuan, Christopher J Hughes, Albert Harris, Sarita V Adve, Douglas L Jones, Robin H Kravets, and Klara Nahrstedt, *GRACE: A Hierarchical Adaptation Framework for Saving Energy*, Tech. report, University of Illinois at Urbana-Champaign, 2004.

[tbf] *Token bucket filter*, `http://linux.die.net/man/8/tc-tbf`.

[tes] *Audio testfile for ITU-T recs P.862*, `http://www.itu.int/rec/T-REC-P.862-200511-I!Amd2/en`.

[TFDM06] Ryan W Thomas, Daniel H Friend, Luiz A DaSilva, and Allen B MacKenzie, *Cognitive networks: adaptation and learning to achieve end-to-end performance objectives*, Communications Magazine, IEEE, 2006, no. 12, pp. 51–57.

[TFDM07] Ryan W Thomas, Daniel H Friend, Luiz A DaSilva, and Allen B MacKenzie, *Cognitive networks*, Springer, 2007.

[TMS11] G. Thamilarasu, S. Mishra, and R. Sridhar, *Improving reliability of jamming attack detection in ad hoc networks*, International Journal of Communication Networks and Information Security (IJCNIS), 2011, no. 1, pp. 57–66.

[TQD+04] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs, *Iperf: The TCP/UDP bandwidth measurement tool*, `http://iperf.sourceforge.net/`, 2004.

[TS07] Geethapriya Thamilarasu and Ramalingam Sridhar, *Exploring cross-layer techniques for security: Challenges and opportunities in wireless networks*, Military Communications Conference, 2007. MILCOM 2007. IEEE, IEEE, 2007, pp. 1–6.

[TW08] Alberto Lopez Toledo and Xiaodong Wang, *Robust detection of mac layer denial-of-service attacks in csma/ca wireless networks*, IEEE Transactions on Information Forensics and Security, 2008, no. 3, pp. 347–358.

[TYCH05] Chien-Chao Tseng, Li-Hsing Yen, Hung-Hsin Chang, and Kai-Cheng Hsu, *Topology-aided cross-layer fast handoff designs for ieee 802.11/mobile ip environments*, IEEE Communications Magazine, 2005, no. 12, pp. 156–163.

[VDS+05] Mihaela Van Der Schaar et al., *Cross-layer wireless multimedia transmission: challenges, principles, and new paradigms*, Wireless Communications, IEEE, 2005, no. 4, pp. 50–58.

[Wal06] S. Waldbusser, *RFC 4502 - Remote Network Monitoring Management Information Base Version 2*, May 2006.

[WAR03] Q. Wang and M.A. Abu-Rgheff, *Cross-layer signalling for next-generation wireless systems*, IEEE WCNC, vol. 2, 2003, pp. 1084–89.

[Wat08] Jon Watson, *Virtualbox: bits and bytes masquerading as machines*, Linux Journal, 2008, no. 166, 1.

[wbe] *Wbest: a bandwidth estimation tool for ieee 802.11 wireless networks*, `http://web.cs.wpi.edu/~claypool/papers/wbest/`.

[WBLO99] G. Wu, Y. Bai, J. Lai, and A. Ogielski, *Interactions between TCP and RLP in Wireless Internet*, Proc. GLOBECOM, 1999, pp. 661–666.

[Weh01] K. Wehrle, *An open architecture for evaluating arbitrary quality of service mechanisms in software routers*, Networking—ICN 2001 (2001), pp. 117–126.

[Wif] *ns-3 Wifi Models*, `http://www.nsnam.org/docs/release/3.7/doxygen/group___wifi.html`, (accessed July 09, 2014).

[Wil08] A. Willig, *Recent and emerging topics in wireless industrial communications: A selection*, IEEE Transactions on Industrial Informatics, 2008, no. 2, pp. 102–124.

[Wir] *Wireless Tools for Linux*, `http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html#wext`, (accessed Nov 14, 2011).

[Wód11] Michał Wódczak, *Aspects of cross-layer design in autonomic cooperative networking*, Cross Layer Design (IWCLD), 2011 Third International Workshop on, IEEE, 2011, pp. 1–5.

[WSNB06] Rolf Winter, Jochen H Schiller, Navid Nikaein, and Christian Bonnet, *Crosstalk: Cross-layer decision support based on global knowledge*, IEEE Communications Magazine, 2006, no. 1, pp. 93–99.

[WSvL+11] Elias Weingärtner, Florian Schmidt, Hendrik vom Lehn, Tobias Heer, and Klaus Wehrle, *SliceTime: A platform for scalable and accurate network emulation*, Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11), March 2011.

[WvLW11] Elias Weingaertner, Hendrik vom Lehn, and Klaus Wehrle, *Device-driver enabled wireless network emulation*, Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques (SIMUTools 2011), ICST, 3 2011.

[XTZW05] Wenyuan Xu, Wade Trappe, Yanyong Zhang, and Timothy Wood, *The feasibility of launching and detecting jamming attacks in wireless networks*, Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing (New York, NY, USA), MobiHoc '05, ACM, 2005, pp. 46–57.

[XWY06] Mingbo Xiao, Xudong Wang, and Guangsong Yang, *Cross-layer design for the security of wireless sensor networks*, Intelligent Control and Automation, 2006. WCICA 2006. The Sixth World Congress on, vol. 1, IEEE, 2006, pp. 104–108.

[YCLT08] Y. C. Yee, K. N. Choong, Andy L. Y. Low, and S. W. Tan, *Sip-based proactive and adaptive mobility management framework for heterogeneous networks*, J. Netw. Comput. Appl., 2008, pp. 771–792.

[YLA02] Wing Ho Yuen, Heung-no Lee, and Timothy D Andersen, *A simple and effective cross layer networking system for mobile ad hoc networks*, Personal, Indoor and Mobile Radio Communications, 2002. the 13th IEEE International Symposium on, vol. 4, IEEE, 2002, pp. 1952–1956.

[Yu04] Xin Yu, *Improving tcp performance over mobile ad hoc networks by exploiting cross-layer information awareness*, Proceedings of the 10th annual international conference on Mobile computing and networking, ACM, 2004, pp. 231–244.

[ZZ08] Qian Zhang and Ya-Qin Zhang, *Cross-layer design for qos support in multihop wireless networks*, Proceedings of the IEEE, 2008, no. 1, pp. 64–76.

A

Syntax of CRAWLER's Configuration Language

In order to provide an abstract configuration to declaratively describe cross-layer coordination algorithms, we have created an own rule-based configuration. The configuration language is inspired by the graphical representation of functional unit compositions. Table A.1 gives the grammar of our configuration language in Extended Backus-Naur Form (EBNF).

Init	→	{ *Section* { *Identifier Rule* | *Trigger* } }
Section	→	"`[init]`" | "`[manual]`" | "`[contextEnter]`" | "`[contextExit]`"
Identifier	→	*IDName* ":"
IDName	→	*Name* { "." *Name* }
Name	→	*Alpha* { *AlphaNum* }
Rule	→	*Name* "(" [*ParameterList*] ")"
ParameterList	→	*Parameter* { "," *Parameter* }
Parameter	→	[*Identifier*] *Value* | [*Identifier*] *Rule* | *IDName*
Value	→	*Boolean* | *Integer* | *Character*
Boolean	→	*Bool* | ["`b`"] "`[`" [*Bool* { "," *Bool* }] "`]`"
Bool	→	"`true`" | "`false`"
Integer	→	*Int* | ["`i`"] "`[`" [*Int* { "," *Int* }] "`]`"
Int	→	typical integer representation
Character	→	*Char* | "`"`" { *Char* } "`"`"
Trigger	→	*IDName* "->" *IDName* { "," *IDName* } ";"

Table A.1 Syntax of our configuration language in EBNF notation. `Alpha` represents uppercase and lowercase letters including "_"; `AlphaNum` represents `Alpha` characters and numbers.

The meaning of the terminal and non-terminal symbols are as follows:

Section is divided into three parts and each part contains a list of rules or triggers.

[init] contains all rules that should be loaded on Crawler's start.

[manual] contains rules that are predefined but not yet loaded into Crawler. These rules can later be managed in the section contextEnter and contextExit.

[contextEnter] this section loads a set of predefined rules as soon as a condition is satisfied. This allows Crawler to react when certain context is available, e.g. if the link is below a threshold then predefined rules (i.e., that are hold in manual section) are automatically loaded.

[contextExit] this section unloads a set of predefined rules as soon as a condition is satisfied. Similar to contextEnter it allows to react when a certain context is not available anymore, e.g., if the link is again above a threshold a rule is unloaded .

Rule is a function with a function name and a list of parameters. The function name is mapped to the corresponding functional unit with the same name. ParameterList represents the query list for the corresponding function unit instance. Every list entry is either a constant value, another rule or an identifier. We enforce the root of every rule to have always a unique name, so that other rules can refer to it.

Trigger represents a notification from one rule to multiple other rules. The first identifier is the identifier of the rule that sends the notification. The following list of identifiers maps to interested rules.

Parameter can be one of the following types:

Value is a constant value with a special type, e.g. integer or boolean. It can be a single value or an array of values.

Rule is a new function for creating a nested composition of functions.

Identifier refers to an already composed rule with the given identifier name. With the help of identifiers much shorter descriptions of desired configurations are possible. It allows utilizing already defined (parts of) rules.

B

Available Stubs and FUs in CRAWLER

B.1 Stubs

TCP

Readable TCP Stubs

Stubs for incoming packets, i.e., packet being received from lower layer and delivering it to higher layer:

`transport.tcp.tcp_in_5001.cwnd.get`	The sending congestion window on port 5001.
`transport.tcp.tcp_in_5001.nxt.get`	The next sequence we send on port 5001.
`transport.tcp.tcp_in_5001.srtt.get`	The smoothed round trip time (<<3) on port 5001.
`transport.tcp.tcp_in_5001.ssthresh.get`	The slow start size threshold on port 5001.
`transport.tcp.tcp_in_5001.una.get`	The first Byte we want an ack for on port 5001.
`transport.tcp.tcp_in_5001.wnd.get`	The window we expect to receive on port 5001.

Stubs for outgoing packets, i.e., packet received from higher layer and delivering it to lower layer:

`transport.tcp.tcp_out_5001.cwnd.get`	The sending congestion window on port 5001.
`transport.tcp.tcp_out_5001.nxt.get`	The next sequence we send on port 5001.
`transport.tcp.tcp_out_5001.srtt.get`	The smoothed round trip time (<<3) on port 5001.
`transport.tcp.tcp_out_5001.ssthresh.get`	The slow start size threshold on port 5001.
`transport.tcp.tcp_out_5001.una.get`	The first Byte we want an ack for on port 5001.
`transport.tcp.tcp_out_5001.wnd.get`	The window we expect to receive on port 5001.

Writable Stubs

Stubs for incoming packets, i.e., packet being received from lower layer and delivering it to higher layer:

`transport.tcp.activate.incomingPacketsPort.set`	Activate a tuning layer for incoming packets at the given port (add port as variable).
`transport.tcp.deactivate.incomingPacketsPort.set`	Deactivate a tuning layer for incoming packets at the given port (add port as variable).

Stubs for outgoing packets, i.e., packet received from higher layer and delivering it to lower layer:

`transport.tcp.activate.outgoingPacketsPort.set`	Activate a tuning layer for outgoing packets at the given port (add port as variable).
`transport.tcp.deactivate.outgoingPacketsPort.set`	Deactivate a tuning layer for outgoing packets at the given port (add port as variable).
`transport.tcp.tcp_out_5001.cwnd.set`	Setter for the sending congestion window on port 5001.
`transport.tcp.tcp_out_5001.nxt.set`	Setter for the next sequence we send on port 5001.
`transport.tcp.tcp_out_5001.srtt.set`	Setter for the smoothed round trip time (<< 3) on port 5001.
`transport.tcp.tcp_out_5001.ssthresh.set`	Setter for the slow start size threshold on port 5001.
`transport.tcp.tcp_out_5001.una.set`	Setter for the first Byte we want an ack for on port 5001.
`transport.tcp.tcp_out_5001.wnd.set`	Setter for the window we expect to receive on port 5001.

Usage Hints for the TCP stubs

- Make sure the flag tcpLayer := yes in CLKernelModule/Makefile.kernel is set when you compile.
- Load the tcp layer: insmod CLKernelModule/layer/tcpLayer2.ko
- Write a port number, that should be monitored or modified via an application to the variables: `transport.tcp.activate.incomingPacketsPort.set` or `transport.tcp.activate.outgoingPacketsPort.set`
- Respectively deactivation of packets port for deactivation
- For port 5001, there exist a kernel module for easy and fast handling: CLKernelModule/src/test/enabletcp.ko, which has to be loaded after tcpLayer2.ko

IP

Stubs for incoming packets, i.e., packet being received from lower layer and delivering it to higher layer:

`ip.incoming.netdevice.name`	Associated device (network layer) of the current incoming ip packet.
`ip.incoming.version`	Version of the current incoming ip packet.
`ip.incoming.header_length`	Header length of the current incoming ip packet (multiple of 32 bit).
`ip.incoming.id`	Identification of the current incoming ip packet.
`ip.incoming.tos`	The type of service (class) of the current incoming ip packet.
`ip.incoming.tos.precedence`	The type of service (precedence) of the current incoming ip packet.
`ip.incoming.total_length`	The length of the current incoming ip packet.
`ip.incoming.can_fragment`	Can the current incoming ip packet be fragmented?
`ip.incoming.is_last_fragment`	Is the current incoming packet the last or a following fragment?
`ip.incoming.fragment_offset`	The fragment offset of the current incoming packet.
`ip.incoming.ttl`	The remaining time-to-live (TTL) of the current incoming packet.
`ip.incoming.protocol`	The protocol type which follows after the ip packet.
`ip.incoming.checksum`	Checksum of the current incoming ip packet.
`ip.incoming.address.source`	Source address of the current incoming ip packet.
`ip.incoming.address.source.readable`	Source address of the current incoming packet (readable format).
`ip.incoming.address.destination`	Destination address of the current incoming ip packet.
`ip.incoming.address.destiation.readable`	Destination address of the current incoming packet (readable format).

Stubs for outgoing packets, i.e., packet received from higher layer and delivering it to lower layer:

ip.outgoing.netdevice.name	Associated device (network layer) of the current outgoing packet.
ip.outgoing.version	Version of the current outgoing ip packet.
ip.outgoing.header_length	Header length of the current outgoing ip packet.
ip.outgoing.id	Identification of the current outgoing ip packet.
ip.outgoing.tos	The type of service (corresponding class) of the current outgoing ip packet.
ip.outgoing.tos.precedence	The type of service (corresponding precedence) of the current outgoing ip packet.
ip.outgoing.total_length	The total length of the current outgoing ip packet.
ip.outgoing.can_fragment	Can the current outgoing ip packet be fragmented?
ip.outgoing.ttl	The remaining time-to-live (TTL) of the current outgoing packet.
ip.outgoing.protocol	The protocol type that follows after the current outgoing ip packet.
ip.outgoing.address.source	Source address of the current outgoing ip packet.
ip.outgoing.address.source.readable	Source address of the current outgoing ip packet in a readable format.
ip.outgoing.address.next_hop	IP address of the next hop.
ip.outgoing.address.next_hop.readable	IP address of the next hop in a readable format.

WiFi - Wireless Extensions

Readable Stubs:

wlan0.wireless_stats.status.get	Gives a certain status information - device dependent for now.
wlan0.wireless_stats.qual.rssi.get	Quality information of the link (rssi value).
wlan0.wireless_stats.qual.level.get	Signal level of the link (dBm).
wlan0.wireless_stats.qual.noise.get	Noise level of the link (dBm).
wlan0.wireless_stats.qual.updated.get	Flag to know if quality information updated.
wlan0.wireless_stats.discard.rx_nwid.get	Packet discarded in the wireless device due to wrong nwid/essid.
wlan0.wireless_stats. discard.rx_invalid_cryption.get	Packet discarded in the wireless device because it is unable to code/decode (WEP).
wlan0.wireless_stats. discard.rx_invalid_fragmentation.get	Packet discarded in the wireless device because cannot perform a MAC reassembly.
wlan0.wireless_stats. discard.tx_max_retries.get	Packet discarded in the wireless device because max. MAC retry count reached.
wlan0.wireless_stats.discard.misc.get	Packet discarded in the wireless device due to other cases.
wlan0.wireless_stats.miss.beacon.get	Packet/Time period missed in the wireless adapter due to missed beacons.

WiFi - Cfg802.11

Readable Stubs:

wlan0.cfg80211.signal.max	Maximum of all received signal strengths (in dBm) from all connected stations.
wlan0.cfg80211.signal.min	Minimum of all received signal strengths (in dBm) from all connected stations.
wlan0.cfg80211.signal.avg	Average of all received signal strengths (in dBm) from all connected stations.
wlan0.cfg80211.survey	Measured noise on the channel (in dBm).
wlan0.cfg80211.inactive_max	Maximum of all current inactive times (in ms) from all connected stations.
wlan0.cfg80211.inactive_min	Minimum of all current inactive times (in ms) from all connected stations.
wlan0.cfg80211.inactive_avg	Average of all current inactive times (in ms) from all connected stations.
wlan0.cfg80211.txrate_max	Maximum of all used transmission rates.
wlan0.cfg80211.txrate_min	Minimum of all used transmission rates.
wlan0.cfg80211.rx_busy	Amount of time the channel was sensed as ”busy” (in ms).
wlan0.cfg80211.channel_time	Amount of time spent on the current network channel (in ms).
wlan0.cfg80211.tx_retries	Amount of packet re-transmissions.
wlan0.cfg80211.tx_failed	Amount of packets that failed to be transmitted (hence, amount of all frames that hit the re-transm
wlan0.cfg80211.tx_packets	Amount of transmitted packets.
wlan0.cfg80211.rx_packets	Amount of received packets.
wlan0.cfg80211.tx_bytes	Amount of transmitted bytes.
wlan0.cfg80211.rx_bytes	Amount of received bytes.
wlan0.cfg80211.channel_time_tx	Amount of time spent transmitting (in ms).
wlan0.cfg80211.channel_time_rx	Amount of time spent receiving (in ms).
wlan0.cfg80211.rx_dropped_misc	todo.

Writable Stubs:

wlan0.cfg80211.channel	Sets the currently used WLAN Channel.

Ethernet

Readable Stubs:

ethX.rx_packets	The total number of packets received.
ethX.tx_packets	The total number of packets transmitted.
ethX.rx_bytes	The total number of bytes received.
ethX.tx_bytes	The total number of bytes transmitted.

Systeminformation

Readable Stubs:

sys.uptime	Uptime of the system since bootup (requires kernel 2.6.33).
sys.totalram	Total RAM of the system in kilobytes.
sys.freeram	Free RAM of the system in kilobytes.
sys.loadaverage.1min	Current [http://de.wikipedia.org/wiki/Load Load] average (last 1 minute).
sys.loadaverage.5min :	Current [http://de.wikipedia.org/wiki/Load Load] average (last 5 minutes).
sys.loadaverage.15min	Current [http://de.wikipedia.org/wiki/Load Load] average (last 15 minutes).
sys.systemname :	Current name of the system.
sys.systemversion	Current version of the system (Kernel version number).
sys.systemrelease	Current release of the system (Date of the setup).
sys.idletime	Idle time of the system in seconds.
sys.cpu.utilization	CPU utilization in percentage since system start.

B.2 FUs

and	A boolean and-operator comparison of boolean types.
or	A boolean or-operator comparison of boolean types.
avg(composite)	Returns the average of a composite. Only works for integers.
bigger(value1,value2)	Returns true if value1 is bigger then value2. Otherwise it returns false. Works only for integers.
history(query,x)	Stores the last x values of e.g. a get-query
if(condition, left_branch, right_branch)	Uses if left branch if condition is met. Otherwise it uses the right branch.
less(value1,value2)	Returns true if value1 is smaller then value2. Otherwise it returns false. Works only for integers.
max(composite?)	Returns the maximum value of a composite(?). Only works for integers.
min(composite?)	Returns the minimum value of a composite(?). Only works for integers.
once(query)	Stores the result of up to 4 queries till calling FU returns true. At this point all stored values can be overwritten by the current results of the queries. Only use it for values that you want to change at the SAME point of time. Only works for integers.
onceother(query)	Same as once but for up to 10 values.
oncelong(query)	Same as once only for longs.
percentdiv(value1,value2)	Returns (value1∗100)/value2. Works for longs and integers.
pollingtimer(x)	Polls every x jiffies.
print(query1,...)	Prints the output into the kernel ring buffer.
printdep	Prints the output into the kernel ring buffer but only if querying FU returned true.
ringcounter	Basic ringcounter.
sub(value1, value2)	Subtracts value2 from value1. Works for longs and integers.
sum(value1, value2)	Adds value2 to value1. Works for longs and integers.

C

Configuration of Machine Learning-based Jamming Detection

In Section 4.5 we presented a machine learning-based jamming detection approach. By using Crawler we monitored several metrics and forwarded the values to the machine learning algorithm running in the user space. As the complete Crawler configuration required too many rules for monitoring and computation of the divers metrics, we moved it to this appendix.

```
chainStreamGet << "\n\
  [init]\n\
  signal_max_"<<this<<":get(\""<<wlanDevName<<".cfg80211.signal.max\")\n\
  signal_min_"<<this<<":get(\""<<wlanDevName<<".cfg80211.signal.min\")\n\
  noise_value_"<<this<<":get(\""<<wlanDevName<<".cfg80211.survey\")\n\
  rx_packets_"<<this<<":get(\"app."<<strategyName<<"."<<receivedPacketsVarName<<"\")\n\
  estimate_"<<this<<":get(\"app."<<strategyName<<"."<<estimateVarName<<"\")\n\
  inactive_time_max_"<<this<<":get(\""<<wlanDevName<<".cfg80211.inactive_max\")\n\
  rx_busy_"<<this<<":get(\""<<wlanDevName<<".cfg80211.rx_busy\")\n\
  channel_time_"<<this<<":get(\""<<wlanDevName<<".cfg80211.channel_time\")\n\
  \n\";

chainStreamSet << "\n\
  [init]\n\
  set_pdr_"<<this<<":set(\"app."<<strategyName<<"."<<pdrVarName<<"\",pdr_value_"
    <<this<<")\n\
  set_tx_dum_"<<this<<":set(\"app."<<strategyName<<"."<<pdrVarName<<"\",tx_retry_value_"
    <<this<<")\n\
  set_tx_dum1_"<<this<<":set(\"app."<<strategyName<<"."<<pdrVarName<<"\",tx_failed_value_"
    <<this<<")\n\
  jammed_"<<this<<":set(\"app."<<strategyName<<"."<<jammedVarName<<"\",0)\n\
  set_dummy_"<<this<<":set(\"app."<<strategyName<<"."<<dummyVarName<<"\",cbr_value_"
    <<this<<")\n\
  set_noise_"<<this<<":set(\"app."<<strategyName<<"."<<noiseVarName<<"\",do_noise_"
    <<this<<")\n\
  set_cbr_"<<this<<":set(\"app."<<strategyName<<"."<<cbrVarName<<"\",do_cbr_"
    <<this<<")\n\
  set_maxit_"<<this<<":set(\"app."<<strategyName<<"."<<maxitVarName<<"\",do_maxit_"
    <<this<<")\n\
  set_realpdr_"<<this<<":set(\"app."<<strategyName<<"."<<realpdrVarName<<"\",do_realpdr_"
    <<this<<")\n\
  set_minsig_"<<this<<":set(\"app."<<strategyName<<"."<<minsigVarName<<"\",do_minsig_"
    <<this<<")\n\
  set_maxsig_"<<this<<":set(\"app."<<strategyName<<"."<<maxsigVarName<<"\",do_maxsig_"
```

```
35      <<this<<")\n\
36    \n\";
37
38  chainStreamVarFunc1 << "\n\
39    [init]\n\
40    noise_avg_"<<this<<":avg(noise_history_"<<this<<")\n\
41    noise_history_"<<this<<":history(noise_value_"<<this<<",10)\n\
42    signal_max_avg_"<<this<<":avg(signal_max_history_"<<this<<")\n\
43    signal_max_history_"<<this<<":history(signal_max_"<<this<<", 10)\n\
44    signal_min_avg_"<<this<<":avg(signal_min_history_"<<this<<")\n\
45    signal_min_history_"<<this<<":history(signal_min_"<<this<<", 10)\n\
46    do_noise_"<<this<<":onceother(noise_avg_"<<this<<")\n\
47    do_minsig_"<<this<<":onceother(signal_min_avg_"<<this<<")\n\
48    do_maxsig_"<<this<<":onceother(signal_max_avg_"<<this<<")\n\
49    \n\";
50
51  chainStreamVarFunc2 << "\n\
52      [init]\n\
53      pdr_value_"<<this<<":percentdiv(rx_packets_init_"<<this<<", estimate_"<<this<<")\n\
54      rx_packets_init_"<<this<<":sub(rx_packets_"<<this<<", do_once2_"<<this<<")\n\
55      do_once2_"<<this<<":once(rx_packets_"<<this<<")\n\
56      do_realpdr_"<<this<<":onceother(pdr_value_"<<this<<")\n\
57      \n\";
58
59  chainStreamVarFunc3 << "\n\
60      [init]\n\
61      inactive_time_max_avg_"<<this<<":avg(inactive_time_max_history_"<<this<<")\n\
62      inactive_time_max_history_"<<this<<":history(inactive_time_max_"<<this<<", 10)\n\
63      do_maxit_"<<this<<":onceother(inactive_time_max_avg_"<<this<<")\n\
64      \n\";
65
66  chainStreamVarFunc4 << "\n\
67      [init]\n\
68      cbr_value_"<<this<<":percentdiv(rx_busy_init_"<<this<<",channel_time_init_"<<this<<")\n\
69      rx_busy_init_"<<this<<":sub(rx_busy_"<<this<<", do_once3_"<<this<<")\n\
70      channel_time_init_"<<this<<":sub(channel_time_"<<this<<", do_once4_"<<this<<")\n\
71      do_once3_"<<this<<":oncelong(rx_busy_"<<this<<")\n\
72      do_once4_"<<this<<":oncelong(channel_time_"<<this<<")\n\
73      status_"<<this<<":sum(0,1)\n\
74      do_cbr_"<<this<<":onceother(cbr_value_"<<this<<")\n\
75      \n\";
76
77  chainStreamFunc << "\n\
78    [init]\n\
79    timer_"<<this<<":pollingtimer(10)\n\
80    counter_"<<this<<":ringcounter()\n\
81    timeout_"<<this<<":less(counter_"<<this<<", 1)\n\
82    timeout1_"<<this<<":bigger(counter_"<<this<<", 8)\n\
83    timer_"<<this<<"->counter_"<<this<<";\n\
84    timer_"<<this<<"->noise_history_"<<this<<";\n\
85    timer_"<<this<<"->inactive_time_max_history_"<<this<<";\n\
86    timer_"<<this<<"->signal_max_history_"<<this<<";\n\
87    timer_"<<this<<"->signal_min_history_"<<this<<";\n\
88    timer_"<<this<<"->timeout_"<<this<<";\n\
89    timeout_"<<this<<"->do_once2_"<<this<<";\n\
90    timeout_"<<this<<"->do_once3_"<<this<<";\n\
91    timer_"<<this<<"->set_pdr_"<<this<<";\n\
92    timer_"<<this<<"->set_dummy_"<<this<<";\n\
93    timer_"<<this<<"->timeout1_"<<this<<";\n\
94    timeout1_"<<this<<"->do_noise_"<<this<<";\n\
95    timeout1_"<<this<<"->do_cbr_"<<this<<";\n\
96    timeout1_"<<this<<"->do_minsig_"<<this<<";\n\
97    timeout1_"<<this<<"->do_maxsig_"<<this<<";\n\
98    timeout1_"<<this<<"->do_maxit_"<<this<<";\n\
99    timeout1_"<<this<<"->do_realpdr_"<<this<<";\n\
100   timeout1_"<<this<<"->set_noise_"<<this<<";\n\
101   timeout1_"<<this<<"->set_cbr_"<<this<<";\n\
102   timeout1_"<<this<<"->set_minsig_"<<this<<";\n\
103   timeout1_"<<this<<"->set_maxsig_"<<this<<";\n\
104   timeout1_"<<this<<"->set_maxit_"<<this<<";\n\
105   timeout1_"<<this<<"->set_realpdr_"<<this<<";\n\
106   \n";
```

Listing C.1 CRAWLER configuration of Machine Learning-based Jamming Detection.

Publications

International Journals

Title:	Harnessing Cross-Layer-Design
Authors:	Ismet Aktas, Muhammad Hamad Alizai, Florian Schmidt, Hanno Wirtz, and Klaus Wehrle
Journal:	Elsevir Ad-Hoc Networks, 2013

International Conferences

Title:	CRAWLER: An Experimentation Platform for System Monitoring and Cross-Layer-Coordination
Authors:	Ismet Aktas, Florian Schmidt, Muhammad Hamad Alizai, Tobias Drüner, Klaus Wehrle
Venue:	13th Int. IEEE Symposium on a World of Wireless, Mobile, and Multimedia Networks, 2012 (WoWMoM'12)

Title:	An Adaptive Codec Switching Scheme for SIP-based VoIP
Authors:	Ismet Aktas, Florian Schmidt, Elias Weingärtner, Caj-Julian Schnelke and Klaus Wehrle
Venue:	12th Int. Conf. on Next Generation Wired/Wireless Networking (New2An'12)

Title:	Graph-based Redundancy Removal Approach for Multiple Cross-Layer Interactions
Authors:	Ismet Aktas, Martin Henze, Muhammad Hamad Alizai, Kevin Möllering and Klaus Wehrle
Venue:	6th Int. Conf. on COMunication Systems and NETworkS (COMSNETS'14)

Title:	FANTASY: Fully Automatic Network Emulation Architecture with Cross-Layer Support
Authors:	Ismet Aktas, Hendrik vom Lehn, Christoph Habets, Florian Schmidt and Klaus Wehrle
Venue:	5th ACM Int. Conf. on Simulation Tools and Techniques (SIMUTools'12)

Title:	Machine Learning-based Jamming Detection for IEEE 802.11: Design and Experimental Evaluation
Authors:	Ismet Aktas, Oscar Punal, Caj-Julian Schnelke, Gloria Abidin, James Gross, and Klaus Wehrle
Venue:	15th Int. IEEE Symposium on a World of Wireless, Mobile, and Multimedia Networks, 2014 (WoWMoM'14)

International Workshops

Title:	Towards a Flexible and Versatile Cross-Layer-Coordination Architecture
Authors:	Ismet Aktas, Jens Otten, Florian Schmidt and Klaus Wehrle
Venue:	Work in progress track, 29th Int. Conf. on Computer Communications (INFOCOM'10)

Title:	A Framework for Remote Automation, Configuration, and Monitoring of Real-World Experiments
Authors:	Ismet Aktas, Oscar Punal, Florian Schmidt, Tobias Drüner, Klaus Wehrle
Venue:	9th ACM Int. Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization (WINTECH'14)