Effective Fusion and Separation of Distribution, Fault-Tolerance, and Energy-Efficiency Concerns

Young-Woo Kwon

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Application

June 3, 2014

Blacksburg, Virginia

Keywords: distributed computing, fault-tolerance, energy-efficiency, middleware, mobile applications, program transformation, runtime system, dynamic adaptation, refactoring

Effective Fusion and Separation of Distribution, Fault-Tolerance, and Energy-Efficiency Concerns

Young-Woo Kwon

ABSTRACT

As software applications are becoming increasingly distributed and mobile, their design and implementation are characterized by distributed software architectures, possibility of faults, and the need for energy awareness. Thus, software developers should be able to simultaneously reason about and handle the concerns of distribution, fault-tolerance, and energy-efficiency. Being closely intertwined, these concerns can introduce significant complexity into the design and implementation of modern software. In other words, to develop reliable and energy-efficient applications, software developers must understand how distribution, fault-tolerance, and energy-efficiency interplay with each other and how to implement these concerns while keeping the complexity in check.

This dissertation addresses five technical issues that stand on the way of engineering reliable and energy-efficient software: (1) how can developers select and parameterize middleware to achieve the requisite levels of performance, reliability, and energy-efficiency? (2) how can one streamline the process of implementing and reusing fault tolerance functionality in distributed applications? (3) can automated techniques be developed to help transition centralized applications to using cloud-based services efficiently and reliably? (4) how can one leverage cloud-based resources to improve the energy-efficiency of mobile applications? (5) how can middleware be adapted to improve the energy-efficiency of distributed mobile applications operated over heterogeneous mobile networks?

To address these issues, this research studies the concerns of distribution, fault-tolerance, and energy-efficiency as well as their interaction. It also develops novel approaches, techniques, and tools that effectively fuse and separate these concerns as required by particular software development scenarios. The specific innovations include (1) a systematic assessment of the performance, conciseness, complexity, reliability, and energy consumption of middleware mechanisms for accessing remote functionality, (2) a declarative approach to hardening distributed applications with resiliency against partial failure, (3) cloud refactoring, a set of automated program transformations for transitioning to using cloud-based services efficiently and reliably, (4) a cloud offloading approach that improves the energy-efficiency of mobile applications without compromising their reliability, (5) a middleware mechanism that optimizes energy consumption by adapting execution patterns dynamically in response to fluctuations in network conditions.

This dissertation is based on 5 conference papers, presented at Middleware'09 [82], SCC'10 [84], MESOCA'11 [77], ICDCS'12 [78], and ICSM'13 [80] as well as 4 journal articles, published at Service Oriented Computing and Applications [83], Information and Software Technology [79], IEEE Computer [152], and Automated Software Engineering [81].[1]

[1]This research received support from the National Science Foundation through the Grant CCF-1116565 (Automated Refactoring Techniques for Efficient and Reliable Distributed Execution).

To my family:

My wife, Eunmi Kim

My father, Gaphyun Kwon

My mother, Okgeum Bae

My son, Aiden Yoonsang Kwon

ACKNOWLEDGEMENTS

My graduate school experience has been an exciting journey and it has provided me with a wonderful opportunity to learn. Any accomplishments I have achieved during this long journey would not have been possible without many individuals. I would like to sincerely thank all those who have helped me complete my Ph.D.

First, I would like to express my deepest appreciation to my advisor, Dr. Eli Tilevich, for his valuable support, prompt guidance, and effective encouragement throughout my graduate program. He has supported me on every aspects of my Ph.D. dissertation research, in particular with emphasis on guiding me to the true research process. He always made himself available whenever I needed his advice, selflessly introducing substantial amount of time guiding this research. I was able to lean how research ideas should be selected and developed to publish at premier venues. Also, I would like to thank him for his advice and invaluable feedback to improve my ability to write research papers. I am very fortunate to have an excellent mentor like him and I would like to thank him for enriching Ph.D. experience.

I would also like to thank my committee members, Patrick Eugster, Dennis Kafura, Binoy Ravindran, and Barbara Ryder, for carefully reading my dissertation and presenting constructive feedbacks and valuable insight that made it possible for me to improve the overall quality of the dissertation research.

My special thanks go to Barbara Ryder, who took time from her busy schedule to provide me with constructive advice in improving my presentation skill and enhancing my dissertation research. I am also thankful to her for her feedback on my dissertation document and my job talk.

During my job search, I greatly thank Eli Tilevich, Binoy Ravindran, and Barbara Ryder, for their help regarding to their recommendation letters and fruitful advice. Without their invaluable guidance and supports, I would not be able to start my academic career at Utah State University.

I am grateful to all CS@VT friends, Software Innovations Lab mates, Korean students for their friendship and countless help. I could develop many research ideas and improve my communication skill based on their comments raised during seminars and technical discussions.

My graduate journey would have not been possible without my family. I would like to thank my family for their love, support, and encouragement. I would like to thank my wife, Eunmi Kim, for everything. Her encouragement and supports have helped me overcome many obstacles for the last 6 years. Also, I would like to thank my parents. I probably would not be writing this dissertation if it were not for my parents who have constantly supported me to complete my Ph.d. Finally, I thank my son, Aiden, who is the joy of my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The last several years have seen a fundamental shift in how the average user takes advantage of computing resources. Traditional desktop software applications are being gradually replaced by a computation model dominated by cloud computing [18, 164, 131, 30, 39, 150]. By embracing cloud computing, industrial enterprises and research establishments reap tangible benefits, including reduced costs, increased automation, greater flexibility, and enhanced mobility [15]. This transition fundamentally has changed how we develop software. Because traditional software development has been tailored toward centralized applications for so long, many centralized applications need to be adapted for distributed execution.

Because the computing resources offered by cloud infrastructures are typically orders of magnitude more powerful than those of mobile devices, the distributed application is also likely to execute more performance efficiently. As a result, distribution has become a versatile optimization mechanism. Through distribution, programmers can minimize energy consumption, maximize performance, or maximize both of them as required by particular software development scenarios. However, a combination of naïve implementation practices and the extreme heterogeneity of mobile computing platforms makes it hard to ensure that distributed applications are always reliable and energy-efficient.

Centralized and distributed applications have different failure modes, so that simply rendering a subset of a centralized application remote does not preserve the original semantics. Distributed applications are subject to partial failure, in which its different components (client, server, or network) may fail independently from each other. Although one cannot handle all the possible failures in a distributed application, some failures have well-known handling strategies. Thus, to better preserve the original execution semantics, programmers must change code to transition applications

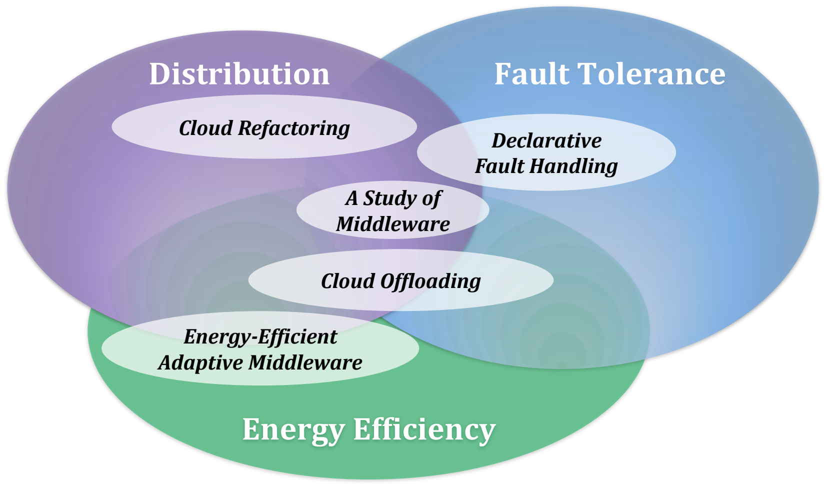


Figure 1.1: Concerns and major research contributions.

to use cloud-based services and add proper fault handling functionality to any application that invokes services remotely.

In addition, as mobile devices are rapidly becoming the primary means of accessing computing resources, energy-efficiency—fitting an energy budget and maximizing the utility of applications under given battery constraints—has become an important software design consideration. One can offload a mobile application's functionality to cloud to reduce its energy consumption. Furthermore, because network communication is one of the largest sources of energy consumption [118], the choice and parameterization of middleware can reduce the amount of energy consumed by a distributed application. However, because mobile applications commonly run on a variety of mobile devices over mobile networks with divergent characteristics, optimizing a mobile application to reduce its energy consumption is non-trivial. Thus, to maximize energy savings, mobile execution should be continuously adapted in response to the fluctuations in the mobile networks.

All in all, to develop reliable and energy-efficient applications, software developers should be able to simultaneously reason about and handle the concerns of distribution, fault-tolerance, and energy-efficiency, as well as their interaction. Figure 1.1 shows the relationship of distribution, fault-tolerance, and energy-efficiency concerns as well as the major research contributions of this dissertation. Being closely intertwined, these concerns can introduce significant complexity into the design, implementation, and maintenance of modern software. In this dissertation, we study how distribution, fault-tolerance, and energy-efficiency interplay with each other and how to implement these concerns while keeping the complexity in check. In the following discussion, we briefly introduce this research's individual parts.

1.1 Major Research Contributions and Scope

This dissertation makes several diverse contributions to address the issues described above. Nevertheless, these contributions form a logical, if not always perfectly chronologically ordered, cohesive narrative. In particular, the first contribution focuses on assessing middleware platforms with respect to their reliability, energy-efficiency, and performance to determine how they interplay and affect the quality of service of distributed applications. Based on this assessment, we provided a set of practical guidelines to help software developers select an appropriate middleware platform.

As discovered by our assessment of middleware platforms, fault-tolerance abstractions constitute an essential part of a middleware platform. An important open issue in distributed system engineering remains the tailoring of fault-tolerance functionality for individual application scenarios, without having to reimplement the fault-handling functionality from scratch. Hence, to improve reliability and facilitate software development, we introduced a declarative fault handling mechanism that is reusable across distributed applications.

Having gained new insights from developing reusable fault-tolerance mechanisms, we then approached the problem of transforming a centralized application to using cloud-based services reliably. To that end, we introduced new refactoring techniques that transform a centralized application into a distributed application. One of these refactoring techniques enhances newly introduced services with reusable fault-tolerance functionality.

The final contribution of this dissertation uses the insights gained from the research thrusts above as guidelines and building blocks. It explores how mobile applications can reduce their energy consumption through distribution. To that end, we introduced a cloud offloading technique by innovating in program analysis, program transformation, and runtime systems. We also developed a new middleware platform that optimizes energy consumption by adapting execution patterns dynamically in response to fluctuations in network conditions.

In the following discussion, we provide an overview of the major contributions made by this dissertation as well as their scope and applicability.

1.1.1 A Study of Middleware

Due to the shift from software-as-a-product (SaaP) to software-as-a-service (SaaS), software components that were developed to run in a single address space must increasingly be accessed remotely across the network. Distribution middleware is frequently used to facilitate this transition. Yet a range of middleware platforms exist, and there are few existing guidelines to help the programmer choose an appropriate middleware platform to achieve desired goals for performance, conciseness, intuitiveness, and reliability. To address this limitation, we first compared and contrasted five middleware implementations in terms of their respective performance, conciseness, complexity, and reliability. Then, as energy-efficiency has become an important software design consideration, their energy consumption characteristics were assessed to help the programmer choose the right abstraction for energy-constrained scenarios.

In Chapter 2, we report on the findings of a systematic study we conducted to compare and contrast major middleware platforms in terms of their performance, conciseness, intuitiveness, reliability, and energy consumption patterns. Based on our findings, we present a set of practical guidelines for the programmer to select an abstraction that satisfies various constraints in place. Our other guidelines can steer future efforts in creating reliable and energy-efficient middleware platforms.

1.1.2 Declarative Fault Handling Against Partial Failure

Refactoring an existing centralized program for distributed executions should preserve its original execution behavior, so that a distributed application would provide the same or equivalent functionality to the user as the original centralized version. In other words, distributing an application should not only improve its performance, availability, scalability, and energy-efficiency, but also furnish the original functionality to the user. However, distributed applications are subject to partial failure, where components of a distributed system can fail independently of each other. The proposed approach focuses on *network volatility* occurring when a network suffers an outage and then shortly becomes operational again. Despite its temporary nature, network volatility is disruptive and detrimental to the user experience. As a result, it is tedious and error-prone to make distributed applications resilient against network volatility.

To address these challenges, in Chapter 3, we report on enhancing middleware with *declarative fault handling*, which is application-level fault handling as compared to the system level approaches to achieve fault tolerance. Programmers can specify and configure fault tolerance strategies to apply by means of a domain-specific language. To counteract partial failures, an extensible hardening framework detects failures as they emerge at runtime, and then handles them by applying dynamically deployed special fault handling components. The framework automatically adds fault-handling ability to an existing middleware system, so that the enhanced distributed application can continue executing in the presence of network volatility without having to change their source code. The fault handling components can be developed by third-party programmers for a variety of distributed applications.

Research Scope and Applicability

This approach has specific engineering objectives, creating pragmatic new technologies that can make distributed service-oriented applications more available, reliable, and secure. One important question concerns whether availability, reliability, and security can be effectively reasoned about and implemented as orthogonal cross-cutting concerns, separate from the core functionality of a given distributed service-oriented application. The scientific consensus has been that it is impossible to achieve this objective in full generality. However, these concerns can be quite effectively separated in certain domains and execution environments.

1.1.3 Automated Refactoring for Reliable Cloud-Based Execution

To alleviate the code transformation hurdles involved in adapting existing centralized applications to take advantage of cloud-based services, we expressed several common program transformations as refactorings, thereby reducing development efforts/costs and increasing programmer productivity. Although the approach is not fully automatic, programmers only determine if the source code should be transformed. The actual transformations are performed by a refactoring engine. However, because centralized and distributed applications have different failure modes, transformed applications are subject to partial failure. Thus, the enhanced refactoring adds the ability to handle partial failures. Then, detected partial failures are handled by fault-tolerant middleware through our declarative fault handling mechanism.

Specifically, in Chapter 4, we report on common program transformations performed when transitioning to cloud-based services, well-amenable to be expressed as a refactoring. In particular, we explored a set of refactoring techniques that facilitate the process of transforming centralized applications to use cloud-based services. These techniques automate the program transformations required to 1) rewrite a class making its methods into remote service methods, 2) partition class methods into service methods and regular methods, rewriting all the communication between the two into remote service calls, 3) re-target all clients of the original class to access its functionality in the cloud by means of remote calls, and 4) add fault handling code to the client. This research introduced *Cloud Refactoring* as a valuable tool in the toolset of software developers charged with the challenges of migrating applications to take advantage of cloud resources.

Research Scope and Applicability

A refactoring may not be a proper approach for transforming all kinds of software applications to cloud-based services. Transforming tightly coupled applications without incurring a significant performance overhead may require deep architectural changes that are not supported by our refactoring techniques. Ensuring good performance requires that remote communication be crude-grained and infrequent. In addition, cloud-based communication is inherently unidirectional: client talks to server but not vice versa. If the original application does not follow this communication pattern, its architecture needs to be changed before our refactoring techniques can be applied.

Our refactoring techniques do not make any provision for a situation when a newly extracted cloud service is used by multiple clients. Then the application logic would have to be modified accordingly to ensure a consistent and efficient access by multiple clients. Furthermore, our fault handling strategies cannot cover all the possible failure cases. In some scenarios, the programmer may need to implement some failure handling strategy by hand, outside the framework provided by our refactoring infrastructure.

1.1.4 Energy-Efficient Mobile Execution

The mobile computing domain is characterized by the applications' energy demands outpacing the devices' battery capacities. Rapid growth in application functionality requires ever greater energy budgets, thus subsuming any advances in battery capacities. A common energy optimization technique for mobile applications is *Cloud Offloading*—placing energy-intensive functionality to run at a remote, cloud-based server. Executing this functionality at a remote server saves the mobile device's battery power.

In Chapter 5, we report on the cloud offloading technique that transforms applications, leveraging static program analysis and program transformation techniques, without destroying their ability to execute locally in the case of network disconnection [78]. Our other contribution to cloud offloading takes advantage of dynamic adaptation by means of a hand-crafted runtime system

[80, 76]. Specifically, this technique automatically enhances a centralized program with the ability to execute across the network, while the local/remote parts are determined dynamically at runtime, as required by the current execution environment.

Research Scope and Applicability

Our approach works with the standard, unmodified hardware/software stack; it employs bytecode engineering to transform programs and a lightweight runtime system to dynamically steer and adapt offloading operations. Our approach makes it possible to keep the maintained version of the mobile application's source code intact, as only the bytecode version is transformed. Cloud offloading requires a minimal programming effort, limited to marking methods as energy hotspots. Our approach makes offloading decisions at runtime by monitoring the execution environment, thus discovering optimal offloading strategies. Finally, the offloading transformations do not preclude the mobile application from executing locally in the case of the network becoming disconnected.

However, our approach is not applicable to all applications. Some mobile applications are written in a monolithic style, in which functionality cross-cuts through traditional modularization program constructs such as classes and methods. Without clear offloading program points, our approach, which operates at the method boundary, would be inapplicable. In addition, we do not take multiple concurrent threads into consideration when determining whether a method can benefit from offloading.

1.1.5 Energy-Efficient Middleware

Because network communication incurs high energy costs in mobile applications, middleware presents a promising target for energy optimizations. Unfortunately, mainstream middleware mechanisms are oblivious to the highly volatile nature of mobile networks, operating over which energy efficiently requires aligning the middleware communication patterns with the network conditions in place.

In Chapter 6, we report on *energy-aware adaptive middleware* that features dynamic adaptation capabilities as well as straightforward configurations that enable the programmer to express a rich set of middleware energy optimizations and the runtime conditions under which these optimizations should be applied. Our results indicate that the technique represents a promising direction in improving the energy efficiency of mobile applications.

Research Scope and Applicability

Our approach provides the generality, separation of concerns, and reusability advantages. Our middleware mechanism is general in that it can be applied to a variety of distributed mobile applications. The middleware enables a greater separation of concerns in that it can change a mobile applications's energy/performance characteristics without affecting its core business logic. The energy optimization strategies and the configurations to apply them are expressed separately from the main source code. This degree of separation also makes it possible to effectively reuse energy optimization strategies and configurations across components and applications.

Although our approach can deliver tangible benefits to the mobile application programmer, it also has some inherent limitations. In particular, the limitations concern its ranges of applicability and usability. The overhead imposed by the middleware runtime makes the approach inapplicable to those distributed mobile applications that use simple, infrequent remote interactions. The runtime overhead is offset if the optimized application spends a substantial amount of energy on remote interactions.

Finally, in Chapter 7, we summarize the technical insights gained from carrying out this research and outline future research directions.

1.2 Broader Impact

By enabling fusion and separation of distribution, fault-tolerance and energy-efficiency concerns, this research will enable software developers to effectively develop reliable and energy-efficient applications. The specific expected broader impacts of this research are as follows. First, the software innovations introduced by this research can reduce cost and improve programmer productivity in adapting centralized applications for distributed execution, which has become required for a large and growing number of application domains. Second, this research can improve the reliability of real-world applications against partial failure via a reusable framework that will eliminate the need to reimplement fault handling functionality, thereby increasing programmer productivity. Finally, this research can help reduce the energy consumption of mobile applications, thus extending battery lives, helping mobile application developers, and benefiting society at large.

1.3 Structure

The rest of this dissertation is structured as follows. Chapter 2 compares multiple middleware abstractions and analyzes assessment results. Chapter 3, 4, 5, and 6 covers motivations, design, implementation, and experimental results of declarative fault handling, cloud refactoring, adaptive cloud offloading, and energy-efficient middleware, respectively. Chapter 7 presents concluding remarks, summarizes the contributions of this research, and discusses future work directions.

Chapter 2

An Assessment of Distributed Programming Abstractions

Due to the shift from software-as-a-product (SaaP) to software-as-a-service (SaaS), software components that were developed to run in a single address space must increasingly be accessed remotely across the network. Furthermore, with battery capacities remaining a key physical constraint for mobile devices, energy-efficiency has become an important software design consideration. However, their energy consumption characteristics are poorly understood. Distributed programming abstractions (e.g., sockets, RPC, messages, etc.) are an essential component of modern software to facilitate this transition. Yet there are few existing guidelines to help the programmer choose an appropriate abstraction to achieve desired goals for performance, conciseness, reliability, and energy efficiency. In this chapter, we compare and contrast multiple distributed programming abstractions and then present a set of practical guidelines for the programmer to select an abstraction that satisfies various constraints in place. We first discuss experimental results with focuses on performance, conciseness, and reliability in Section 2.1, and then focus on energy efficiency in Section 2.2.

2.1 An Assessment of DPAs for Accessing Remote Services

The next couple of years will see a fundamental shift in how the average user takes advantage of computing resources. Traditional shrink-wrapped software applications will move in the direction of a computation model dominated by cloud computing [57, 158]. In this shift, the provisioning of software will evolve from software-as-a-product (SaaP) to software-as-a-service (SaaS). For example, a desktop application could be modified so that much of its execution takes place at a remote server in the cloud, with only the GUI rendered locally. The GUI part is likely to run on a mobile device, for example a smart phone.

Two levels of infrastructure are needed to realize this vision of software services. Firstly, component models are needed to define services and their interfaces. The Open Service Gateway Initiative (OSGi) [107] provides a platform for defining and managing components that can be used as services. It is used by developers to package features as components for separate deployment, and by end users to select components they need.Secondly, middleware infrastructure is needed to allow services to be accessed remotely. There are several different kinds of middleware, and each has different performance, conciseness, complexity, and reliability characteristics. Middleware can be based on messaging, remote procedure calls, or remote evaluation, with the option of asynchronous processing. The trade-offs between these approaches have not been properly examined and, as a result, are poorly understood.

To address this lack of understanding, in this section we describe a case study we have conducted to examine the trade-offs of using different middleware platforms of accessing services remotely. For the case study, we chose a realistic OSGi service that has been integrated into several commercial applications. This service is the Lucene search engine library [145] that provides functionality to index and search text files in Java. For the case study, we implemented a simple dictionary application that can search and return definitions, find synonyms, as well as suggest corrections for misspelled or partially-specified words.

We have implemented three Lucene-based services using five different middleware platforms: TCP sockets, synchronous and asynchronous remote calls in R-OSGi [120], Message Oriented Middleware (MOM) [9], and Remote Batch Invocation (RBI) [61]. For each implementation, we measured: (1) the total number of lines of uncommented code and its cyclomatic complexity, (2) the aggregate latency of invoking remote service methods, and (3) the degree of reliability of remote service methods in the presence of network volatility. The amount of code and its cyclomatic complexity are two standard software engineering metrics most commonly used to assess the complexity and quality of a software artifact [117]. The aggregate latency of invoking a service is a performance metrics that indicates how long it takes for the clients to derive the expected benefits when using the service. This metrics comprehensively assesses the Quality of Service (QoS) from the end user's perspective. Finally, the ability of a remote service to cope with network volatility is critical to maintaining the required QoS in the majority of realistic network environments.

distributed programming abstraction and a middleware system we have recently introduced [61]. In RBI, a batch is a collection of method calls, conditional statements, and loops that is transfered in bulk to the server, which executes the collection and returns the results to be assigned to local variables. Although RBI clients resemble traditional RPC clients, they have a fundamentally different, service-oriented execution model. As such, our implementation of OSGi in RBI is the first non-RPC implementation of the OSGi R4.2 specification, which codifies how OSGi bundles should be accessed remotely.

Based on the results of our case study, the technical contributions of this article are as follows:

- The first non-RPC remote implementation of the OSGi R4.2 specification.
- A comprehensive evaluation of the trade-offs between the performance, conciseness, complexity, and reliability of middleware platforms for accessing services remotely.
- A systematic analysis of the evaluation that can help inform a working programmer about which middleware platform should be used to access a given service remotely.

The rest of this chapter is structured as follows. Section 2.1.1 introduces the concepts and technologies used in this work. Section 2.1.2 describes the implementation of OSGi in RBI. Section 2.1.3 describes our case study and its results. Section 2.1.5 discusses related work, and Section 2.1.6 presents future research directions and concluding remarks.

2.1.1 Technical Background

We first describe the distributed programming abstractions we have evaluated and then introduce the issue of measuring energy consumption in software systems.

Distributed Programming Abstractions (DPAs) Distributed computing coordinates the execution of multiple remote processes. Distributed programming abstractions (DPAs) provide programming and runtime support for one process to execute functionality in a different process. In other words, by eliminating the need for low-level network programming, programming abstractions offer convenient building blocks for constructing distributed systems. Major, widely used DPAs include sockets, messaging, remote procedure/method calls, and remote services.

Remote Procedure Calls Remote Procedure Calls (RPC) serve as a foundation for a wide range of implementations. In this model, the programmer expresses functionality to be invoked in a different process as a regular procedure. However, when such a procedure is invoked, the runtime executes it in a remote process, transferring the parameters and returning the results. RPC has been extended to support object-oriented programming through Remote Method Invocation (RMI); object proxies forward invocations across processes. Representative RMI implementations are Java RMI and XML-RPC.

Message Oriented Middleware To communicate through messages, remote processes can take advantage of Message Oriented Middleware (MOM). MOM commonly supports synchronous and asynchronous interactions through two primary message topologies: point-to-point and publish/-subscribe. With point-to-point, a sender delivers messages to a particular client by depositing them onto a message queue. With publish/subscribe, a sender publishes messages for multiple clients through intelligent broadcasting, called a message topic. To communicate through messages, Java programs can use the standardized API of the Java Message Service (JMS) [100]. In this study, we use a popular JMS-compliant MOM infrastructure called ActiveMQ [148].

Remote Services Service Oriented Architectures (SOA) provide uniform access to a variety of computing resources in multiple application domains. In SOA, software components are provided as services, self-encapsulated units of functionality accessed through a public interface. Services can access each other only via each other's public interfaces. Remote OSGi (R-OSGi) [120] is an RPC-based DPA for OSGi. The R-OSGi distribution infrastructure allows accessing OSGi services remotely through a proxy-based approach, with proxies exposed as standard OSGi bundles. R-OSGi is based on RPC, but both synchronous and asynchronous. The OSGi R4.2 specification codifies the discovery and usage of remote services [107], with Apache CXF DOSGi [149] implementing this specification as SOAP-based Web services.

Remote Batch Invocation As an alternative to RPC, whose unit of distribution is a single procedure call, we introduced Remote Batch Invocation (RBI) [61], whose unit of distribution is a block of code. RBI partitions blocks of code into remote and local parts, while performing all communications in bulk. Batches are specified using a *batch* statement, with the body of a batch statement combining remote and local computation. A batch block looks like a collection of remote method calls but is executed using *remote evaluation* [136], in which all the remote calls are transmitted in a single, compiler-constructed *batch script*. In addition, data is moved in bulk between client and server. RBI differs from RPC in that the unit of distribution is a block of code rather than a single procedure call. The details of RBI are discussed in the following section, which also shows how RBI can be used to provide remote access to OSGi services.

2.1.2 OSGi in RBI

RBI introduces a `batch` statement that executes multiple remote calls using a single remote round trip to the server. Figure 2.1 shows how the Lucene OSGi service can be accessed with RBI. Note that the `batch` block includes looping and conditional statements. The `batch` language extension is transformed into standard Java.[1]

[1]Please refer to our ECOOP 2009 papers for translation details [61].

```
//Get BundleContext object from the Activator class
BundleContext ct = ... ;

// Retrieve the remote service object
ServiceReference sref = ct.getServiceReference(IRSearch.class.getName());
IRSearch rs = context.getService(sref);

// Instantiate Service object for batch
Service service = new Service(rs, IRSearch.class);

//Prepare the search query
Term term = new Term(DEFINITION, word);
Query query = new TermQuery(term);

batch (Lucene ls : service) {
 //Invoke the remote search function
 final TopDocs topDocs = ls.search(query);
 StringBuffer defBuffer = new StringBuffer();

 // Retrieve meanings from the search result
 for (ScoreDoc hits : topDocs.scoreDocs) {
  Document doc = ls.doc(hits.doc);
  if (doc != null) {
   defBuffer.append(doc.getValues(DEFINITION));
  }
 }
}
```

Figure 2.1: Example of batch invocation.

The RBI runtime executes multiple calls (combined with conditional and looping constructs) to a given remote service. Finally, RBI/OSGi does not require any changes to remote service interfaces, which are discovered and bound using a standard OSGi registry.

The runtime architecture of RBI, shown in Figure 2.2, consists of a service consumer, service provider, batch processor, and distribution provider. Once the service provider registers a service in the OSGi framework, the distribution provider instantiates a server that can be accessed remotely. The service consumer discovers and retrieves the remote service, and then the distribution provider creates a proxy for importing the service. Upon the service consumer making remote calls, the batch processor aggregates them into a single descriptor, which is transmitted across the network to the service provider. The service provider's batch processor interprets the descriptor, invoking

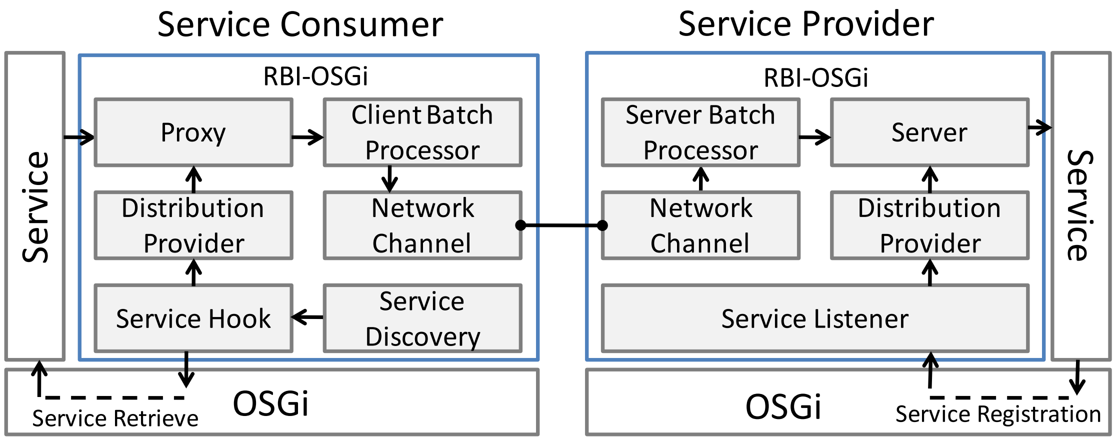


Figure 2.2: RBI/OSGi Architecture.

the appropriate service methods, and sends the results back to the service consumer.

RBI Runtime System

To integrate OSGi with RBI, we connected RBI to the standard OSGi services, `ServiceListener` and `ServiceHook`. Once a `ServiceListener` is registered with OSGi, it starts receiving lifecycle change events for the registered service. The distribution provider uses a `ServiceListener` to determine when a server must be instantiated to process remote requests. The `ServiceHook` service, introduced only in the OSGi R4.2 specification, intercepts service events, raised in response to the service consumer retrieving the remote service, and creates a proxy for accessing services remotely.

The `ServiceHook` service makes it possible to treat local and remote services uniformly, with the only difference concerning their configuration. In other words, switching from using the local version of a service to a remote version and vice verse does not require any source code changes, which are confined to configuration files. Because the OSGi R4.2 specification requires that remote service interfaces be decoupled from their implementations, the `ServiceHook` service accomplishes that by making it possible to switch implementations through a simple configuration file change.

Figure 2.3 demonstrates how straightforward the RBI/OSGi architecture makes it to export a remote service. All it takes to register a remote service is to define its RBI/OSGi properties, including the remote service's interface name, the local address, and the local port number. Specifically, the `service.exported.interfaces` property defines the exported interfaces. The `service.exported.configs` property specifies the available distribution provider such as RBI. Once the local address and port number are specified, the service can be assessed remotely.

```
Dictionary props = new Hashtable();
props.put("service.exported.interfaces", *);
props.put("service.exported.configs", "edu.vt.cs.dosgi.rbi.rs");
props.put("edu.vt.cs.dosgi.rbi.rs.url", local_address);
props.put("edu.vt.cs.dosgi.rbi.rs.port", local_port);

context.registerService(IRSearch.class.getName(), new RSearchImpl(), props);
```

Figure 2.3: Example configuration for exporting remote services.

For a client to import a remote service, an XML configuration file must be provided. Figure 2.4 provides an example of such a configuration file. Mirroring the properties used to export the remote service, the XML configuration file specifies them in the same order, starting with `service.exported.configs`, followed by `service.exported.interfaces`. We are currently implementing a design in which RBI/OSGi server and client modules can use either an XML-based or a hard-coded configuration. This design provides significant flexibility advantages: since RBI/OSGi can work with regular Java interfaces or classes (also known as Plain Old Java Objects or POJOs), any standard OSGi service will be able to export and import RBI/OSGi remote services by means of a configuration file.

2.1.3 Case Study

To compare different middleware platforms, we compared remote access to a set of three services packaged as an OSGi bundle. We chose the Lucene search engine library, which is distributed as an OSGi bundle, thus providing a service interface. The Lucene search services have been used in real-world applications in domains including Web search frameworks (e.g., Nutch [146]) and enterprise systems (e.g., Solr [147]). Using Lucene, we implemented three services to search for

```
<service-descriptions xmlns= "http://www.osgi.org/xmlns/sd/v1.0.0">
  <service-description>
    <provide interface=IRSearch/>
    <property name=service.exported.interfaces>*
    </property>
    <property name=service.exported.configs>
      edu.vt.cs.dosgi.rbi.rs
    </property>
    <property name=edu.vt.cs.dosgi.rbi.rs.address>
      remote_address
    </property>
    <property name=edu.vt.cs.dosgi.rbi.rs.port>
      remote_port
    </property>
  </service-description>
</service-descriptions>
```

Figure 2.4: Example configuration file for importing remote services.

(1) a word's definition, (2) a word's list of synonyms, and (3) a list of spelling suggestions for a misspelled word. Note that service (2) extends the functionality of service (1), and service (3) extends the functionality of service (2). Thus, service (2) includes all the functionality of service (1), and service (3) includes that of services (1) and (2).

For our case study, we examined how these services can be accessed remotely using five different middleware platforms. To that end, we compared each of the five implementations in terms of their respective performance, conciseness, complexity, and reliability.

For the purposes of this study, we define our metrics as follows:

- **Performance:** the total execution time it takes to execute a service, including both network latency and business processing.
- **Conciseness:** the total of Uncommented Lines of Code (ULOC) it takes to write the service.
- **Complexity:** the McCabe cyclomatic complexity (MCC) [92].
- **Reliability:** the ability to withstand temporary network volatility, when the communication network experiences an outage [82].

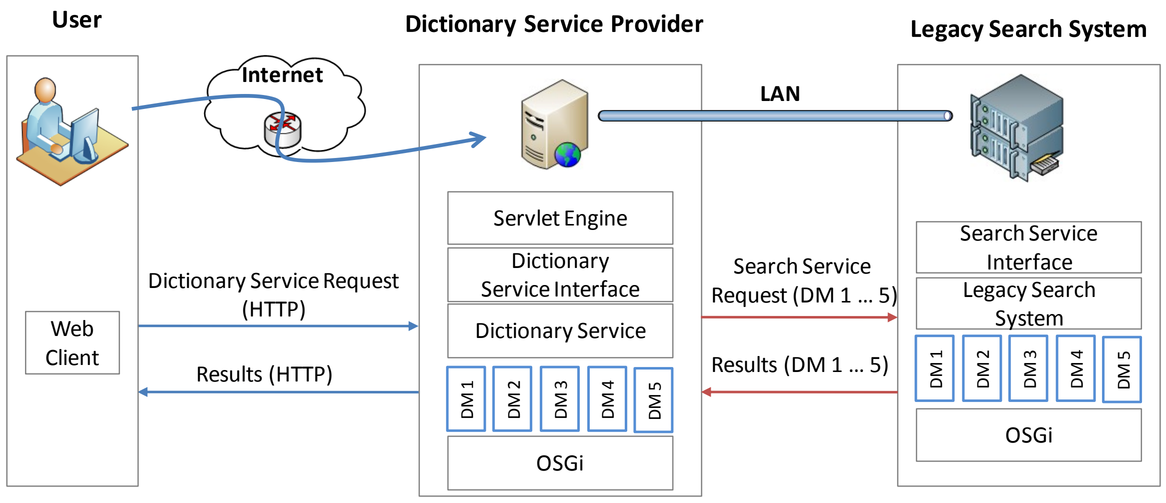


Figure 2.5: Dictionary system.

In this benchmark, we compare these metrics for five middleware platforms: (1) synchronous R-OSGi, (2) asynchronous R-OSGi, (3) Message-Oriented Middleware, (4) raw sockets, and (5) our own RBI implementation to OSGi.

Experimental Setup

All the experiments were conducted on the client machine running 3.0 GHz Intel Dual-Core CPU, 2 GB RAM, Windows XP, JVM 1.6.0 13 (build 1.6.0 13-b03), and the server machine running 1.8 GHz Intel Dual-Core CPU, 2.5 GB RAM, Windows 7, JVM 1.6.0 16 (build 1.6.0 16-b01), connected via a local area network (LAN) with a 100Mbps bandwidth, and 1ms latency. Our results may not be applicable for Wide Area Network (WAN) environments, which are characterized by higher levels of volatility and latency. In fact, some of the middleware mechanisms we have evaluated (e.g., synchronous RPC) are known to have been ineffective in such environments [37].

Figure 2.5 depicts a diagram describing the specifics of our experimental setup. The Lucene OSGi bundle is located on a separate node (server) and is accessed remotely from another node (client). To start the benchmarking of a given setup, we constructed a simple Web client that communicates with the client node through HTTP. By navigating a Web browser to a URL associated with any

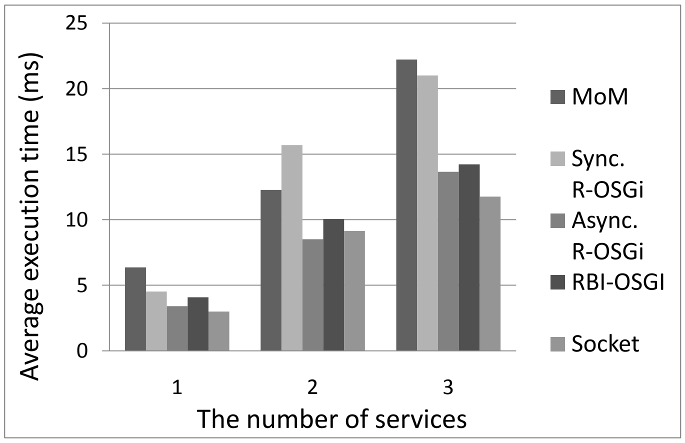


Figure 2.6: Performance Comparison.

of the five middleware implementations, a servlet at the client node invokes its corresponding benchmark method.

Performance

Each benchmark method calls three services in sequence, repeating each service call 1,000 times and then reporting the averaged time. Only the time to invoke the Lucene-based services is taken into account, while the HTTP communication to trigger different benchmarks is omitted.

Figure 2.6 shows the averaged performance for each service. Because each of the three services takes an increasing number of remote roundtrips, for each middleware platform, the total execution time grows for services 2 and 3.

For each service, raw sockets provide the best performance. Asynchronous R-OSGi comes close second. RBI/OSGi using synchronous communication comes quite close to asynchronous R-OSGi. Synchronous R-OSGi is always slower than RBI/OSGi, due to the latter middleware platform aggregating multiple remote calls and invoking them in bulk.

Surprisingly, our MOM-based implementation consistently showed the poorest results across all benchmarks. The reason is because the implementation we used, ActiveMQ, is based on a publish-subscribe rather than a point-to-point communication model. Although MOM-based platforms with point-to-point communication models have been described in the literature [100], the commercial MOM implementations tend to communicate through a publish-subscribe mechanism. While ActiveMQ offers a point-to-point communication option, it is realized as a layer on top of the publish-subscribe infrastructure, with both options offering the same performance results. Publish-subscribe models are beneficial when messages have to be broadcast to a large number of recipients. In our setup, when using MOM for client-server communication, the overhead of involving a message queue was never amortized.

Conciseness and Complexity

Table 2.1 shows the total uncommented lines of code (ULOC) it takes to implement each of the three services using different middleware platforms. The ability to express the same functionality in fewer lines of code has tangible Software Engineering benefits. If the probability of introducing software defects is proportional to the size of a program, fewer lines of code implies a lower defect probability.[2]

Table 2.1: Conciseness and Complexity Comparison.

Middleware platform	**Service**	**ULOC**	**Max. MCC**	**Middleware platform**	**Service**	**ULOC**	**Max. MCC**
Sync. R-OSGi	Service 1	14	7	MOM	Service 1	1172	8
	Service 2	14	10		Service 2	1207	13
	Service 3	14	17		Service 3	1231	23
Async. R-OSGi	Service 1	148	8	Sockets	Service 1	2722	8
	Service 2	170	12		Service 2	2793	13
	Service 3	212	25		Service 3	2839	23
RBI/OSGi	Service 1	23	7				
	Service 2	27	10				
	Service 3	33	17				

The table also shows their McCabe Cyclomatic metric (MCC).[3] The MCC metric is commonly employed to assess the complexity of a codebase. Intuitively, the MCC is indicative of the programming effort required to implement and understand a piece of code. Thus, if a middleware

[2]Our explicit assumption is that the programmer does not try to artificially reduce the ULOC numbers.

[3]We used Metric 1.3.6 http://metrics.sourceforge.net/ for the measurements.

usage scenario produces a lower MCC metric, the complexity will be reduced, as the programmer is likely to exert less effort to produce or modify the code. The ULOC numbers in Table 2.1 combine the client and server portions, while excluding 1918 ULOC that it takes to implement the functional processing part of all the remotely-accessed services.

As expected, our sockets-based implementation is the longest. A programmer has to design and express a low-level communication protocol, which also includes the format for each transferred message. In addition, avoiding deadlocks and ensuring good performance requires that message sending and receiving be handled by different threads. The MOM implementation is the second longest. A programmer has to implement a listener interface and register it with the messaging system and handle messages that arrive out of order. In addition, the programmer must define the messages and process them at the application level. Asynchronous R-OSGi follows next. A programmer also has to implement a listener, but R-OSGi eliminates the need for the programmer to implement messages and setup the communication. The RBI/OSGi implementation takes about an order of magnitude fewer lines of code than the asynchronous R-OSGi one. RBI/OSGi is a method-based middleware mechanisms that does not require the programmer to write any communication-specific code. The synchronous R-OSGi implementation takes about the same amount of code as that of RBI/OSGi. RBI adds a couple of lines of code to setup and express a batch. With respect to complexity, the raw sockets, asynchrononuns R-OSGi, and MOM implementations have high MCC, while synchronous R-OSGi and RBI/OSGi ones have lower MCC.

Reliability

As it turns out, only our MOM-based implementation has built-in fault tolerance capabilities provided by ActiveMQ. It can operate in what is called “persistent mode” that stores every message to be sent in stable storage. Upon disconnection, the undelivered messages are rescheduled for delivery after the network becomes reconnected.

If reliability in the face of network volatility is required, Table 2.2 summarizes how fault handling mechanisms can be adopted in each middleware platform. Even when a middleware mechanism

Table 2.2: Reliability Comparison.

Middleware platform	Fault handling	3rd party solution
Synchronous R-OSGi	N/A	DR-OSGi
Asynchronous R-OSGi	N/A	DR-OSGi
RBI/OSGi	N/A	DR-OSGi
MOM	built-in	N/A
Sockets	N/A	N/A

does not have built-in facilities for dealing with network volatility, our recent research [82] has shown how such facilities can be plugged into a middleware infrastructure, thereby improving reliability in the face of network volatility.

2.1.4 Discussion

Here we discuss some of the implications of the performance, conciseness, complexity, and reliability measurements presented above. In our discussion, we attempt to provide specific recommendation for the developers of serviceoriented applications.

Figure 2.7 depicts the trade-offs between the performance, conciseness, complexity, and reliability guarantees offered by each middleware platform. As it turns out, no platform satisfies all four guarantees. Therefore, programmers should choose an appropriate platform with the immediate needs of their service applications and their deployment environments in mind.

Threats to Validity

The measurements above are subject to both internal and external validity threats. The internal validity is threatened by the way in which we chose to implement our subject services by using different middleware platforms. In our daily programming practices, we do not regularly use all of the five platforms. Therefore, the way we chose to implement our service may not be fully optimal, in terms of using the proven design patterns. We believe, however, that our programming practices are representative of that of the common programmer.

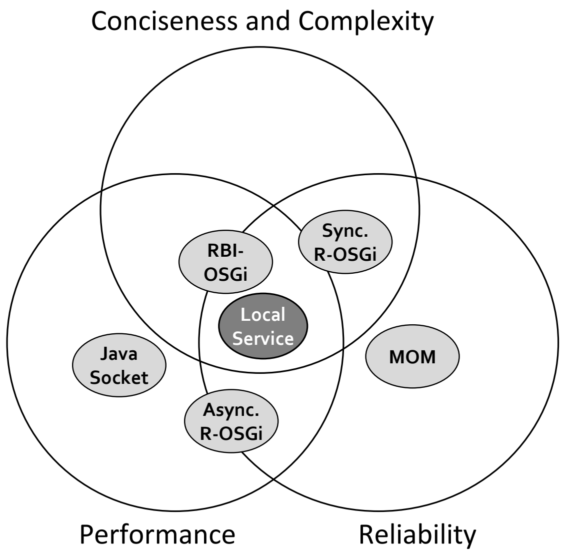


Figure 2.7: Trade-offs between the performance, conciseness, complexity, and reliability levels.

Despite the established practice of using the MCC metric to measure complexity, some experts argue whether complexity always positively correlates with programming effort. If such a correlation turns to be low, the internal validity of our measurements would be further threatened. It is worth noting, however, that defining and measuring programming effort remains hard, as this metric is highly subjective.

The external validity is threatened by our choice of an existing OSGi bundle to be accessed remotely. OSGi public interfaces have been carefully designed to be coarse-grained, and more naively-designed service interfaces can have finer granularity. In that case, the performance disparities between synchronous R-OSGi and the asynchronous alternatives would be even more pronounced.

The external validity of our study is threatened further, as our experiments are not particularly large and varied in terms of the actual services used. Even though few real applications are composed entirely of services, some realistic applications may use more services of different kinds than we have done in our studies. Since not all services are as carefully designed as that of the Lucene search engine, using a more diverse set of services would have likely yielded a greater result variability, particularly with respect to performance and reliability.

Performance

Even coarser grained service interfaces cannot completely eliminate latency concerns. As our measurements show, asynchronous communication leads to better performance. Unfortunately, business logic may require synchronous service calls. Our RBI/OSGi platform can reduce the aggregate latency of multiple remote service calls without asynchronous processing.

Conciseness and Complexity

Despite their performance advantages, asynchronous designs tend to be more complicated, taking more code that is more complex to express. RPC-based abstractions, including our RBI/OSGi, are more straightforward to implement and understand.

Reliability

The reliability of a distributed application is dependent on the reliability of its constituent components, which include both the execution units implementing the application's functionality and the network connecting them. One can argue that the ULOC metrics is inversely proportional to the level of reliability of an individual software component. If the probability of a bug can be expressed in terms of the lines of code and its complexity (e.g., $X\%$ that a software defect exists within N lines of code), then shorter and less complex implementations are less likely to contain bugs. In the light, our ULOC and cyclomatic complexity metrics can also serve a double duty as local reliability metrics.

With respect to distributed execution, the common wisdom of distributed system development suggests that reliability is best implemented on a per-application basis. There is value, however, in handling system-level errors at the middleware level. In that light, using MOM leads to applications that can withstand temporary network disconnections. Such fault-tolerance capacities can be factored into existing systems, as we have demonstrated in our recent research [82].

Price/Performance Ratio

So far, we compared our different middleware platforms using a single metrics. To obtain deeper insights, we introduce a new metrics, *price/performance*, represented by the following

$$PP = \frac{(RULOC/LULOC) \times MCC}{LET/RET}$$

where $RULOC$ and $LULOC$ are local and remote uncommented lines of code, respectively; MCC is a complexity of code (i.e., McCabe complexity number); and LET and RET are local and remote execution times, respectively. The minimum *price/performance* ratio is 1, which can only be achieved when no distribution is present. In other words, the *price/performance* ratio is minimized when its numerator and denominator are approaching 1. Since $LULOC$ and LET are fixed, only RET and $RULOC$ can affect the ratio.

Figure 2.8 depicts the *price (LOC and MCC)/performance* ratio that accounts for both the LOC and MCC values. When MCC is considered, the order of *price/performance* is MOM, sockets, asynchronous R-OSGi, synchronous R-OSGi, and RBI/OSGi, with the smaller value being preferred. Based on this analysis, RBI/OSGi represents a highly-promising alternative to standard middleware, offering a low *price/performance* ratio along with an intuitive programming model.

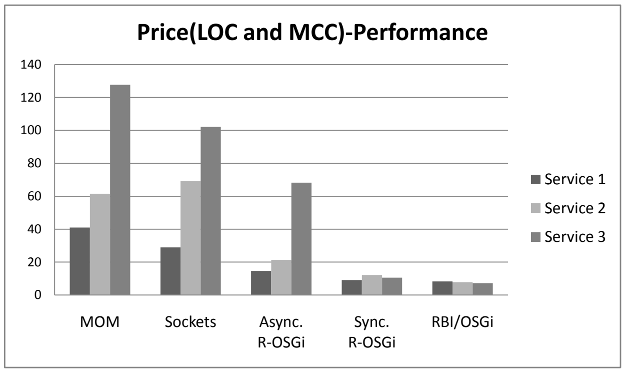


Figure 2.8: The price (LOC and MCC)/performance ratio comparison.

2.1.5 Related Work

The related state of the art includes other studies assessing different properties of middleware platforms as well as a critical assessment of middleware platforms. We describe these two directions next.

Studies of Middleware Platforms

This is not the first effort aimed at comparing and contrasting different middleware platforms. Gokhale et al. [51] assess how the abstraction level of a middleware platform affects its performance. To that end, they measure the overall execution time of micro benchmarks implemented using different middleware platforms ranging in their abstraction level, with sockets being the lowest and CORBA the highest. Their findings confirmed that abstractions incur performance costs in middleware platforms as they do in other computing artifacts. Indeed, lower-level platforms tend to outperform higher-level ones. Nevertheless, abstractions in middleware are necessary to successfully cope with the complexities of constructing modern distributed applications.

Demary et al. [31] compare the round-trip latencies of different configurations of RPC-based middleware platforms, including different versions of CORBA, Java RMI, and XML-RPC implementations. They have found Java RMI to be most efficient and Web services such as XML-RPC incurring a considerable overhead. The performance overhead of Web services often stems from the inefficiencies of XML processing, and various optimization of XML encoding and decoding have been proposed in the literature [37].

Juric et al. [66] have compared RMI, RMI tunneling, and Web services (i.e., SOAP RPC) in terms of their performance characteristics. Mirroring the results of other such studies, this study also found RMI having the best performance in terms of the round-trip latency. Interestingly, this study also found Web services performance to be comparable to that of RMI. Other efforts focused on evaluating MOM and JMS implementations in terms of their respective performance, scalability, and reliability [156, 122].

As compared to these studies, this work focuses on middleware platforms for accessing remote services. In addition to comparing their respective performance, we also investigate their standard software engineering metrics and reliability. By comparing these platform across multiple axes of their properties, we aim at obtaining comprehensive guidelines that can guide programmers needing to satisfy both system and software engineering requirements. These guidelines can help programmers choose an appropriate middleware platform for accessing services remotely.

Middleware Abstractions and Platforms

Remote Procedure Call (RPC) [143] has been one of the most prevalent communication abstractions for building distributed systems. To support distributed object-based applications, RPC has been extended into various distributed object systems, including Common Object Request Broker Architecture (CORBA) [106], the Distributed Component Object Model (DCOM) [14], and Java Remote Method Invocation (RMI) [165]. Despite the ubiquity of RPC, its shortcoming and limitations have been continuously highlighted [141, 161, 125]. Some experts even argue that RPC has been harmful in terms of its influence on distributed systems development [160]. Asynchronous messaging and events, including publish-subscribe abstractions [29], are frequently mentioned as better alternatives to RPC in terms of scalability and reliability.

As confirmed by our study, exposing distributed functionality through a familiar procedure call paradigm of RPC and its object-oriented counterparts provides expressiveness and ease of implementation advantages. Our RBI/OSGi middleware is an attempt to address some of the limitations of RPC, while retaining its advantages without incurring the complexities of asynchronous processing of message- and event-based abstractions.

2.1.6 Conclusion

Due to the advantages provided by services, SaaS has entered the mainstream of commercial software development and a growing percentage of computing functionality is becoming accessible as a service. The programmers who need to access remote services are faced with the challenges of choosing an appropriate middleware platform for the task at hand. To assist the programmers in their decision process, we compared the performance, expressiveness, and reliability of five different middleware platforms for accessing services remotely. Our measurements and analysis not only help the programmers in choosing between different middleware platforms, but also can inform the design of new platforms for accessing services remotely.

2.2 The Impact of DPAs on Application Energy Consumption

Mobile devices have been surpassing stationary computers as the primary means of utilizing computing [47]. As a result, several software design assumptions need to be fundamentally reconsidered to produce applications that use the limited resources of a mobile device optimally. One such resource is energy, provided by constantly improving but always limited batteries. Indeed, energy efficiency has become an important software design constraint [103].

Network communication constitutes one of the largest sources of energy consumption in a distributed application [8]. To streamline the implementation of its network-related functionality, a distributed application can take advantage of programming abstractions (sometimes referred to as *middleware*), which can influence its energy consumption profile. To implement the same application functionality, a software designer can select from a range of distributed programming abstractions, a decision that can significantly impact how much energy the application will consume. Unfortunately, this impact on energy efficiency when choosing one distributed programming abstraction over another has not been studied systematically, thus leaving software designers no choice but to rely on their intuition when reasoning about application energy efficiency.

Unfortunately, relying on one's gut feeling [25] about the energy consumption characteristics of distributed programming abstractions can lead to suboptimal designs that may compromise the application's overall business utility. Although some of the results presented here may seem "obvious," the main contribution of this work is empirical evidence that can support or discredit these

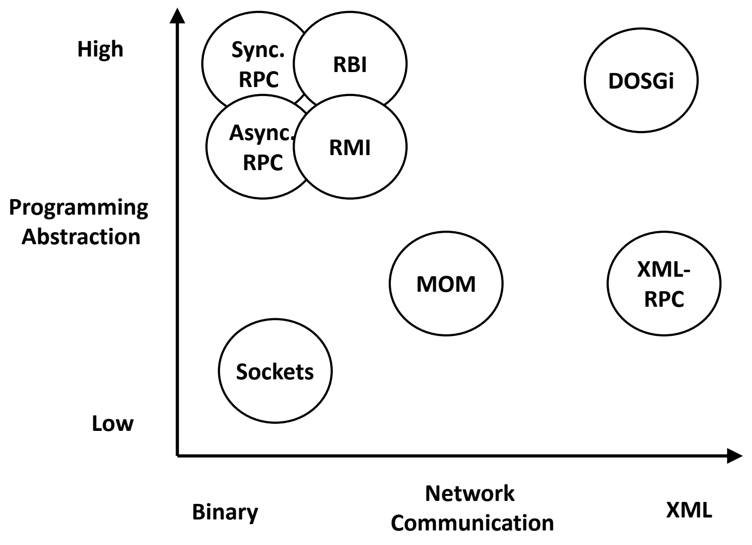


Figure 2.9: Considered DPAs and their classification.

commonly held beliefs about application energy consumption.

To equip the software designer with an informed understanding of application energy consumption, we have focused on the Open Service Gateway Initiative (OSGi), an industry standard for deploying software in multiple domains, including mobile platforms. In particular, we have studied how OSGi bundles (coarse-grained software components) communicate with each other by means of distributed programming abstractions, having considered eight abstractions that differ on two axes: network communication footprint and level of abstraction (See Figure 2.9).

In terms of network communication footprints, we have considered platforms that transfer data in binary and in XML-based formats. In terms of the level of abstraction, we have considered socket-, remote method call-, and message-based platforms. Specifically, we have studied TCP sockets; Message Oriented Middleware (MOM); J2EE RMI; XML-RPC; and OSGi-based remote methods (synchronous and asynchronous), our own Remote Batch Invocation (RBI), and SOAP-based services.

For each considered distributed programming abstraction, our experiments assessed the energy consumption (1) of passing varying volumes of data over networks with different latency/bandwidth characteristics, (2) of marshaling/unmarshaling complex data, and (3) of staying idle. The main contributions of this research include:

- **A systematic study of energy consumption in distributed programming abstraction mechanisms:** We have systematically compared and contrasted the energy consumption of eight distributed programming abstraction mechanisms; by varying the mechanisms while keeping the rest of functionality fixed, we were able to accurately estimate the impact of programming abstractions on the overall application energy consumption.
- **Energy consumption profiles for the aforementioned abstractions:** By analyzing the results of our study, we ranked the abstraction mechanisms by their energy consumption profiles, thus informing programmers needing to choose between different mechanisms.
- **Guidelines for energy-efficient and -aware distributed programming abstractions:** We put forward several guidelines that can guide software engineering researchers, who strive to innovate in the distributed programming abstraction space with energy efficiency in mind.

The rest of this chapter is structured as follows. Section 2.2.1 introduces the studied distributed programming abstractions. Section 2.2.2 describes our experimental study, while Section 2.2.3 analyzes the results. Section 2.2.4 infers energy consumption patterns and proposes new guidelines for both programmers and distributed system designers. Section 2.2.6 discusses our studies and Section 2.2.5 discusses related work. Finally, Section 2.2.7 presents concluding remarks.

2.2.1 Technical Background

We first describe the distributed programming abstractions we have evaluated and then introduce the issue of measuring energy consumption in software systems.

Distributed Programming Abstractions (DPAs)

Distributed computing coordinates the execution of multiple remote processes. Distributed programming abstractions (DPAs) provide programming and runtime support for one process to execute functionality in a different process. In other words, by eliminating the need for low-level network programming, programming abstractions offer convenient building blocks for constructing distributed systems. Major, widely used DPAs including sockets, messaging, remote procedure/method calls, and remote services, have been described in Section 2.1.1.

Measuring Energy Consumption

To measure energy consumption, two primary approaches have been proposed in the literature. One approach leverages specialized hardware (e.g., ACPI[4] or IPMI[5]). These hardware solutions can measure energy consumption quite precisely, but they do not map the consumed energy to the specific application functions or execution phases.

Another approach leverages energy models. For example, Seo at el. [129] put forward a model that divides the total energy consumed by an application into the *functions* of computation, communication, and infrastructure (e.g, JVM garbage collection, implicit OS routines, etc.). Kansal at el. [68] put forward an alternate model that instead focuses on the *phases* of waiting, execution, and idling.

Our measurement model amalgamates features of both of these models. Specifically, we focus on both application *functions* and *phases* by distinguishing between computation and communication, while also differentiating between the phases at which the energy is consumed. It is the amalgamated model that makes it possible for us to infer application energy consumption patterns. By flexibly adjusting our model for the measurement scenario at hand, we are able to infer general energy consumption patterns while ignoring the irrelevant factors. For example, our model omits

[4]Advanced Configuration and Power Interface: `http://www.acpi.info`

[5]Intelligent Platform Management Interface: `http://www.intel.com/design/servers/ipmi/index.htm`

the energy consumed by the infrastructure (i.e., it assumes that software design does not directly affect low-level infrastructure functions such as garbage collection and OS calls).

2.2.2 Measuring Energy Consumption of Distributed Programming Abstractions

In devising our approach to measuring energy consumption of DPAs, we wanted to be able:

1. to understand which components of DPA mechanisms mainly affect their overall energy consumption.
2. to identify temporal patterns in how DPA mechanisms consume energy; these patterns can guide the programmer in search of an abstraction delivering an application-specific energy consumption profile.
3. to infer opportunities for improving the energy efficiency of emerging abstractions.

Next, we first present our energy consumption measurement model. Then we describe our experimental measurements. And finally, discuss the results.

Energy Consumption Model

We estimate total energy consumption by computing the workload incurred by each major piece of functionality. Specifically, the total energy consumption comprises two components—application and DPA:

$$E_{total} = E_{APP} + E_{DPA} = E_{CPU} + E_{mem} + E_{disk} + E_{comm}$$

where E_{APP} is the application-specific energy consumption, which includes E_{CPU}—energy consumed by CPU processing, E_{mem}—energy consumed by memory access, E_{disk}—energy consumed by I/O operations, E_{comm}—energy consumed by network communication.

For our experiments, we have excluded both the disk and memory access components from our measurements. The measured DPAs do not use disk I/O, and one cannot reliably distinguish between the energy consumption incurred by accessing application vs. DPA-specific memory without specialized hardware. Thus, our model considers the energy consumed by a DPA during CPU processing and network communication (i.e., $E_{DPA} := E_{CPU} + E_{comm}$).

Furthermore, our measurements are confined to the client side of all distributed interactions; we assume a client/server communication model, in which server computation and communication do not exhaust battery power. This assumption makes this work inapplicable to energy-conscious server environments or peer-to-peer setups, with mobile devices communicating with each other directly. We plan to extend our measurement model to a broader set of scenarios as a future work direction.

When an application executes, it goes through several phases: initialization, execution, idling, and termination, with the resulting energy consumption divided into four processes:

$$E_{total} = E_{init} + E_{exe} + E_{idle} + E_{term}$$

where the energy consumption for each of these phases is denoted as E_{init}, E_{exe}, E_{idle}, and E_{term}, respectively. For systematic evaluation, one must not only measure the energy consumption of a running application, but also the energy consumption during the application's initialization, idling, and termination phases.

Experimental Setup

Our experimental setup comprised a client and a server. The server machine: 3.0 GHz Intel Dual-Core CPU, 2 GB RAM, Windows 7, and JVM 1.6.0 13 (build 1.6.0_13-b03); the client machine: 2.53 GHz Intel i3 CPU (dual-core), 4 GB RAM, Windows 7, and JVM 1.6.0 16 (build 1.6.0_23-b05). The client and server were connected via a wireless LAN. To create a controlled networking environment with delay and bandwidth limitation, we have used `Network Emulator for Windows Toolkit` [95], a popular network emulator. To measure energy consumption, we have used `pTopW` [34], a process-level power profiling tool that measures energy consumption at the kernel level.

An important goal of this work is to ensure that our results are applicable to distributed applications running on a broad range of mobile computing devices, ranging from laptops to phones. That is why although a laptop is a mobile device, whose energy consumption is an essential issue, we chose our client machine's setup (the CPU, OS, VM) to be as close as possible to the latest models of smartphones and tablets. The somewhat high amount of RAM makes it possible to run the emulator and profiling tools without causing memory paging. Without the RAM taken by our measurement infrastructure, the client machine has about 1 GB left available for running applications, a typical setup for a modern hand-held device.

Because our goal is to determine how the choice of a DPA affects application energy consumption, our measurements focus on application-level energy-consumption patterns rather than on the underlying systems stack (e.g., OS and hardware). We also chose to perform our measurements over a Wi-Fi connection rather than a cellular network such as 3G. The reason is the increasing prominence of Wi-Fi networking, even for hand-held mobile devices. According to CISCO, Wi-Fi networks occupy 36% of the Internet traffic, while cellular networks deliver less than 10 % of traffic [22]. In fact, major US cities, including San Francisco, Washington D.C., Los Angeles, and New York City, have started to provide municipal wireless access through Wi-Fi networks [49]. Therefore, our experimental environment is typical for executing a substantial class of modern distributed applications.

Measurement Methodology

We have based our test suite on the benchmarks originally proposed by the JavaParty project [56] to measure the efficiency of DPAs. These benchmarks comprise remote invocations with varying parameter sizes and types. Similarly, our test suite assumes that a client needs to execute some server methods, each of which takes different parameters. Because the executed server methods are empty, one can reasonably attribute the measured energy consumption to the underlying DPAs.

We have implemented eight versions of the same benchmark that have the same functionality but communicate through different abstractions. The client and server parts of each version are OSGi bundles. We do not measure the energy consumed by the server bundle.

Then, we have experimented with three emulated network conditions that have the following respective round trip time (RTT) and bandwidth characteristics: 2 ms and 50 Mbps, typical for a high-end mobile network; 30 ms and 1 Mbps, typical for a medium-end mobile network; and 30ms and 300 Kbps, typical for a low-end or congested mobile network.

Benchmarks

Benchmark I: Energy Consumed by Initialization

Table 2.3 shows how much energy is consumed by initializing each DPA mechanism, a phase that also includes the initialization of the OSGi framework. DOSGi incurs the highest initialization costs; sockets, R-OSGi, RMI, and XML-RPC all initialize more energy efficiently than either RBI or MOM. DOSGi's high initialization cost are due to its dependence of a high number of third-party OSGi services[6]. The same explanation applies to RBI and MOM, even though their reliance on third-party OSGi bundles is not as significant as that of DOSGi.

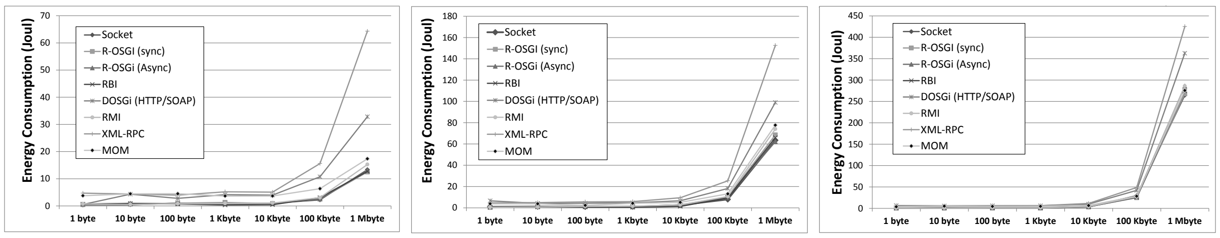
Table 2.3: Energy consumption—initializing DPAs.

Socket	**R-OSGi (sync)**	**R-OSGi (async)**	**RBI OSGi**	**DOSGi (SOAP)**	**RMI**	**XML RPC**	**MOM**
6.5786	7.574	7.574	10.609	72.278	7.395	7.44	11.722

Benchamrk II: Energy Costs of Invoking Remote Functionality

In this experiment, we isolate the energy costs of initiating the execution (i.e., invoking) of various remote methods (i.e., $E_{invoke} = E_{CPU} + E_{comm} - E_{init}$). We measured the aggregate energy consumption of invoking the server method `void ping(byte[])` 100 times in a loop; each experiment was repeated 10 times with the results averaged.

[6]In case of Apache CXF Distributed OSGi, it loads 52 bundles.

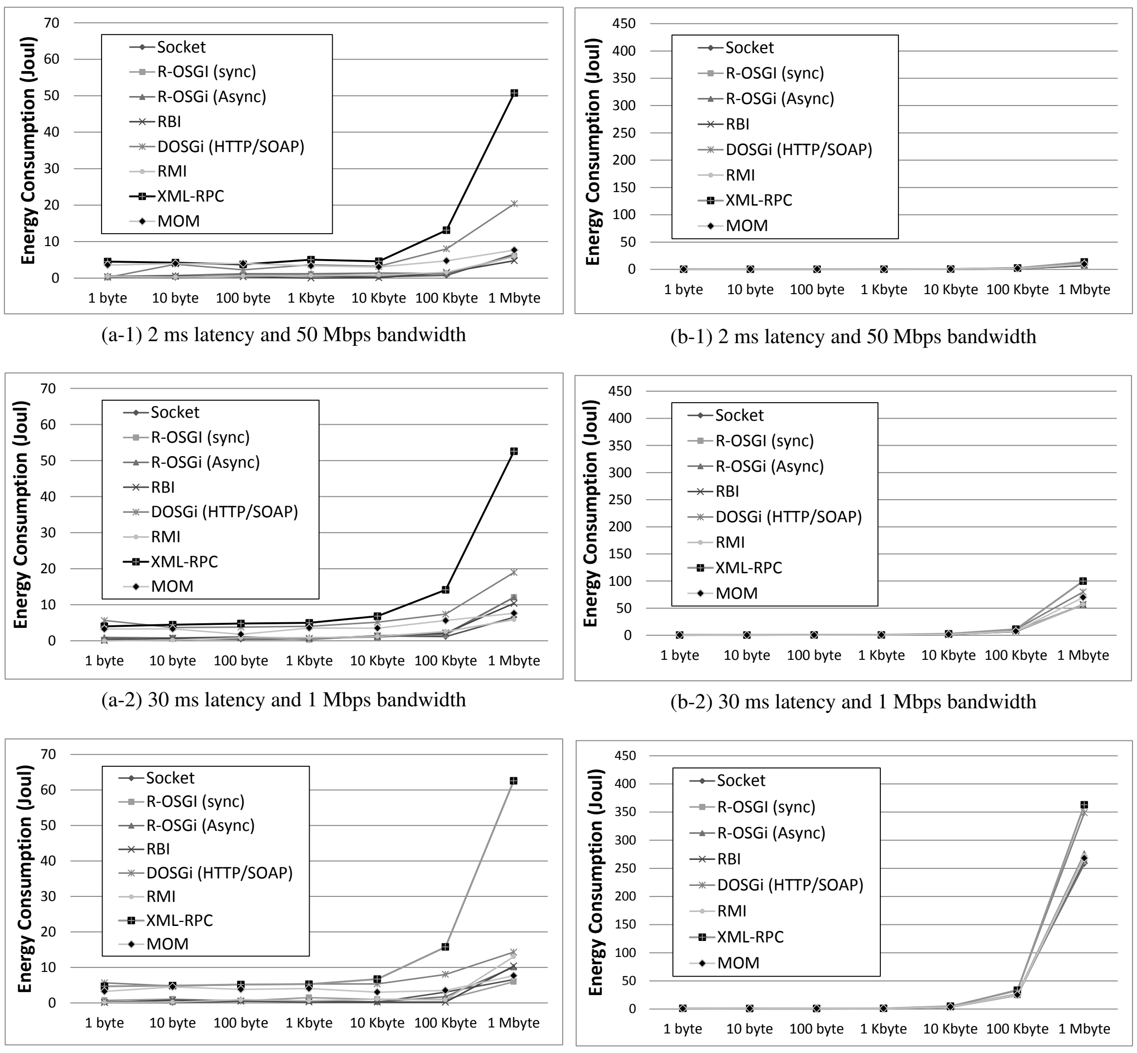


(a) 2 ms latency and 50 Mbps bandwidth (b) 30 ms latency and 1 Mbps bandwidth (c) 30 ms latency and 300 Kbps bandwidth

Figure 2.10: Energy consumption—invoking remote functionality.

Figure 2.10 shows how much energy was consumed by each DPA mechanism. For all platforms, the energy consumption is directly proportional to the increases in latency and transferred data sizes. For example, under the emulated high-end mobile network (i.e., 2 ms latency and 50 Mbps bandwidth), all distributed programming abstractions consume little energy up until the arguments' size reaches 100 Kbytes. Beyond this argument size, the energy consumption begins to increase linearly. Similarly, once the latency goes up to 30 ms and limited the bandwidth goes down to 300 Kbps, the energy increases significantly. The effect is particularly pronounced for DOSGi and XML-RPC, due to their high bandwidth requirements for transferring XML.

Figure 2.11 breaks down the energy consumption between the CPU and network communication portions. The left three figures depict each DPA mechanism's overall CPU energy consumption. The CPU energy consumption is directly proportional to the size of the transferred data. Specifically, when the latency increases and the bandwidth decreases, the energy consumed by CPU processing remains constant. However, the energy consumed by network processing increases significantly, particularly for XML-based DPAs. A surprising result is that asynchronous processing, be it in asynchronous R-OSGi, MOM, or sockets, does not affect the CPU energy consumption. This could be due to the fact that idle CPU cores still consume energy. These results indicate that the network characteristics with respect to the size of the transferred data can significantly influence the overall energy consumption.



(a-1) 2 ms latency and 50 Mbps bandwidth

(b-1) 2 ms latency and 50 Mbps bandwidth

(a-2) 30 ms latency and 1 Mbps bandwidth

(b-2) 30 ms latency and 1 Mbps bandwidth

(a-3) 30 ms latency and 300 Kbps bandwidth

(b-3) 30 ms latency and 300 Kbps bandwidth

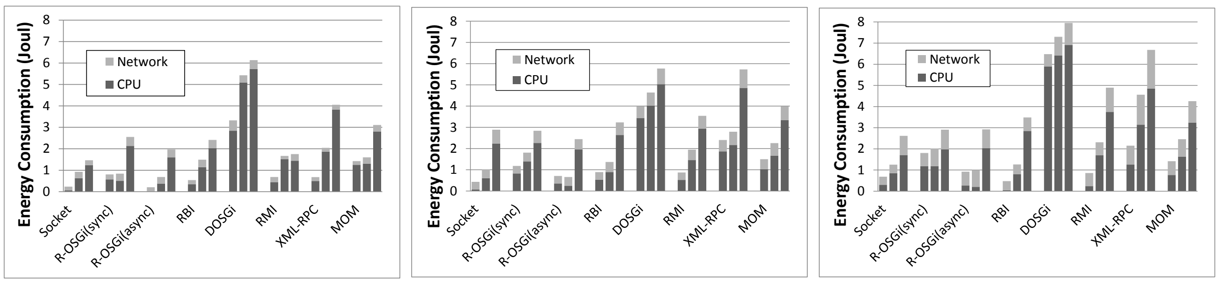
(a) Energy consumption—CPU

(b) Energy consumption—network communication

Figure 2.11: Energy consumption—CPU and network communication.

Benchmark III: Data marshaling/unmarshaling

In this experiment, we isolate the energy costs of marshaling/unmarshaling the data sent as parameters to the invoked remote methods. We have modified the remote methods instead of taking byte buffers to take arguments of different types that the DPA mechanism in place has to marshal/unmarshal. Specifically, we measured the energy consumed by passing (1) an object containing 32



(a) 2 ms latency and 50 Mbps bandwidth (b) 30 ms latency and 1 Mbps bandwidth (c) 30 ms latency and 300 Kbps bandwidth

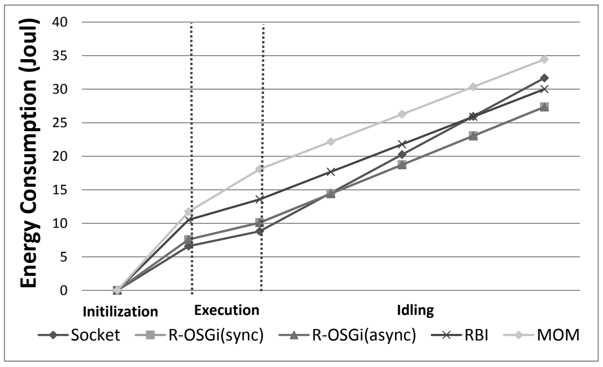
Figure 2.12: Energy consumption of three serialization cases.

`int` fields, (2) 1 non-primitive object which has two other non-primitive objects[7], or (3) a binary tree of 100 nodes, each holding an `int` value and two child recursive references. Each case's transferred data size is as follows: (1) 32 $\times$ 4 bytes = 128 bytes, (2) overall objects sizes are estimated as approximately 420 bytes, and (3) (1 $\times$ 4 bytes + 2 $\times$ 32 bytes) $\times$ 100 = 680 bytes.

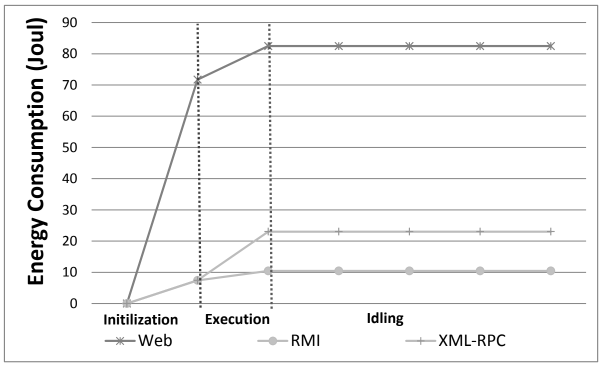
Figure 2.12 shows the energy consumed by each DPA. As expected, XML-based DPA mechanisms consume more energy than those that use either native Java serialization (i.e., RMI, Sockets, and MOM) or optimized serialization mechanisms (i.e., RBI and R-OSGi). Although the inefficiency of Java serialization is well known [114], our experiments did not indicate it to consume significantly more energy than the optimized serialization mechanisms. At least, the difference was not nowhere near as large as that between XML-based and binary serialization formats.

When breaking down the consumed energy into CPU and network processing, XML-based DPAs are particularly vulnerable to limited network conditions, as transferring bulky XML-encoded data can quickly increase the amount of required network transmissions, thereby raising up the energy costs of network communication. Thus, energy-sensitive mobile applications should prefer DPAs that encode data in binary.

[7]This test case is widely used for assessing the efficiency of Java serialization mechanisms. We used revision r128 of JVM serialization benchmark: `http://code.google.com/p/thrift-protobuf-compare/source/detail?r=128`



(a) Socket, R-OSGi(sync), R-OSGi(async), RBI, and MOM



(b) DOSGi (HTTP/SOAP), RMI, and XMP-RPC

Figure 2.13: Energy consumption per execution phases.

Benchmark IV: Energy Consumption per Execution Phases

In this experiment, we measured the aggregate energy consumption of invoking the server method `void ping (byte[100KB])` 100 times in a loop over the emulated network with 2 ms latency and 50 Mbps bandwidth. The benchmark initializes, executes its functionality, idles for one minute, and then exits. We measured how much energy is consumed by each of these phases, with Figure 2.13 showing the results.

Despite its name, the idling phase is important: when an application is long-running, the energy consumed when idling may constitute a significant percentage of the application's energy budget. In fact, our measurements indicate that some DPA mechanisms may consume more energy when idling than when executing remotely, for some application patterns. During idling, it is the open network connections that consume energy. In other words, keeping a network connection open indefinitely (i.e., until the application exits or the connection is interrupted) consumes energy at a constant rate. As we have determined, however, RMI, DOSGi, and XML-RPC consume no energy when idling. Using these DPAs will save energy for long running applications that experience prolonged idle periods.

2.2.3 Result Analysis

Based on the results obtained from the experiments above, we next analyze the results to infer some general energy consumption patterns in DPA mechanisms. Even though we infer the following patterns by analyzing the results obtained from benchmarking the eight DPAs above, we express these patterns in general terms, making them applicable to a wide variety of DPA mechanisms. These patterns should inform software designers charged with the challenges of choosing the right DPA for energy-constrained application scenarios.

Because the same application behavior can be implemented by using any of the equivalent DPAs, being aware of the application's energy consumption patterns becomes an important decision support aid for software designers. By matching these patterns with the intended application behavior, a software designer can make an informed choice when deciding which of the available DPA mechanisms should be applied to a given application scenario.

- Transferring increasing volumes of data over limited networks (characterized by diminishing bandwidths and growing latencies) causes a direct increase in energy consumption. Therefore, when an application is likely to be executed over a limited network, software designers should favor binary-based DPA mechanisms over XML-based ones, as the former ones encode the transferred data more concisely, which reduces the bandwidth requirements.
- In the case of high latency networks, asynchronous DPA mechanisms should be used to avoid having to block remote communications while waiting for a response. Our results indicate that asynchrony does not incur additional energy costs, and as such constitute a viable software building block in the presence of high latency.
- Even though marshaling/unmarshaling can be computationally complex, these functionalities tend to consume more energy on network communication than on CPU processing. Transferring large, complex object graphs across a network requires high bandwidth. At the same time, encoding objects into concise binary representations requires substantial CPU

processing. Therefore, for high-throughput networks, simple serialization protocols can yield an acceptable energy consumption, as it would reduce CPU processing and would transfer larger volumes of data without causing an energy consumption spike due to insufficient bandwidth. However, if the underlying network is limited (high latency and low bandwidth as in a congested network), designs that employ CPU-intensive routines to serialize the transferred data concisely should be preferred, as they would reduce the energy consumed by network processing, a dominant energy consumption ingredient for these types of network.

- Applications with long idle periods should prefer DPA mechanisms that do not consume energy when idling. Sophisticated DPA features, such as issuing heartbeat messages and alternate service discovery, do consume energy even if no core DPA-related functionality is utilized. Therefore, for applications with prolonged idle cycles the energy costs of idling DPA mechanism should be taken into account. For example, using DPA mechanisms that follow stateless communication protocols (e.g., Web services) can reduce the overall energy consumption, as these mechanisms do not maintain any state between remote interactions.

- When both high performance and low energy consumption are equally at stake, no high-level DPA can outperform raw sockets, which has the highest energy consumption/performance ratio. However, asynchronous and batched RPC come close second, while offering convenient programming abstractions to the programmer. When large data volumes are to be transferred across the network, binary DPA mechanisms offer a higher ratio than XML-based ones.

2.2.4 Energy Consumption Patterns and Guidelines

Based on our results, we next present several guidelines for designing energy efficient DPAs.

Energy Consumption Patterns

Designing Energy-Efficient DPAs

Minimize network interactions and transferred data size As network communication incurs the largest costs in the overall energy budget, an energy efficient DPA mechanism should strive to transmit small data volumes over high-throughput networks. Because limited network conditions are hard to avoid, system designs should aim at minimizing the frequency of network interactions and reducing the transferred data size. To achieve the first objective, energy-sensitive designs should minimize state exchange messages (e.g., service discovery message, heartbeat, etc.) to an absolute minimum. To achieve the second objective, binary protocols should be favored over XML-based ones, data compression and advanced serialization (e.g., kryo, protobuf, etc)[8] should be used, and delta should be applied whenever possible. Because algorithmically intensive data compression can increase the CPU energy consumption, the right trade-offs should be sought between transferring smaller data and encoding it into compressed formats.

Share core DPA components across different applications If more than one mobile application can share the same DPA mechanism, the device's aggregate energy consumption may be reduced for two reasons: (1) the initialization phase in a DPA mechanism can consume substantial energy and should be amortized across multiple applications whenever possible; (2) because when idle, a DPA mechanism can still incur energy costs, sharing the infrastructure across applications will reduce its idling time. In the OSGi framework, multiple applications can share common components, realizing the benefits outlined above.

Monitor energy consumption levels and handle outliers For controlling fine-grained energy consumption at the application level, DPA mechanisms should provide energy-monitoring APIs. Such APIs can monitor energy consumption and report potential usage outliers (e.g., attempting to

[8]Various versions of serialization tools have been tested and discussed here: https://github.com/eishay/jvm-serializers/wiki/

send a large data volume over a limited network). The programmer can then implement functionality to handle such abnormalities by either postponing the remote interaction or even replacing it with some local computation. Energy monitoring should not, however, consume additional energy. Estimating energy consumption rather than measuring it directly provides a pragmatic trade-off.

Provide different connection management mechanisms Connection management policies differ: a connection can be terminated after each remote interaction or reused a varying number of times. Reusing connections can both waste and save energy, if managed flexibly. To provide such flexibility, an API can provide methods to select an appropriate connection management policy for a given application's characteristics. For example, for an application rarely invoking remote methods, establishing a connection at every request saves energy. However, for an application frequently invoking remote methods, it is reusing a connection that saves energy. The programmer should have the flexibility to specify the desired policy on a per-application basis.

Flexibly adapt at runtime Based on our measurements, the underlying network environment determines whether and after which threshold the transferred data should be compressed to save energy. A DPA tuned for a particular network through a set of static optimizations is unlikely to consume an optimal amount of energy when operating over networks with fluctuating bandwidth/latency characteristics. An effective energy consumption behavior requires that the DPA switch optimizations on and off dynamically in response to such fluctuations. Although it is the application's business logic that determines what data needs to be transferred across the network, the DPA in place can cluster, encode, and compress the transferred data by means of adaptive optimization.

2.2.5 Related Work

To the best of our knowledge, this work is the first attempt to assess the energy consumption characteristics of DPA mechanisms. However, several prior efforts have informed and inspired this work. These efforts fall into three major categories: studies assessing different properties of DPAs, measuring software energy consumption, and energy saving strategies for various computer system layers.

Studies of DPA mechanisms

Different properties of distributed programming abstractions have been assessed, including performance, scalability, reliability, and programming effort. Gokhale et al. [51] assess how the abstraction level of a distributed programming abstraction affects its performance. Other efforts focused on evaluating MOM and JMS implementations in terms of their respective performance, scalability, and reliability [156, 122]. Our prior work [83] compares DPA mechanisms in terms of performance, reliability, and programming effort. This work complements these studies by assessing the energy consumption of major DPA mechanisms.

Energy Consumption Measurement

Because energy efficiency has become an important consideration in software design, several recent research efforts have focused on creating effective approaches to measuring energy consumption. Three primary approaches have been described in the literature: at the architecture, network, and application levels. An example of an architecture level energy measuring approach is PowerPack [48], which physically connects to the CPU, disk, memory, and mother board component to measure and analyze the energy consumption of high-performance applications. Then, it maps the measured energy consumption patterns to the application's source code, making it possible to analyze energy consumption both at the hardware and source code levels. An example of a network level energy measuring approach is described in reference [8], which measures energy consumption of the general network activity for 3G, GSM, and WiFi networks. Examples of an application level measurement approach are described in reference [54], which measures how VoIP applications consume energy, and in reference [167], which measures how video streaming applications consume energy; both of these focus on mobile phones as their execution environment. JALEN monitors runtime energy consumption by injecting the monitoring code into Java bytecode [104]. As compared to the architecture, network, and application levels measurements, the focus of this work is on DPAs or middleware, a software layer that is situated in between hardware and software layers. Nevertheless, our measurement methodology is an example of an application level approach.

Energy-Saving Techniques

Extending the battery life of mobile devices by reducing the energy consumption of mobile applications has been the focus of multiple complimentary research efforts: energy-efficient design patterns and programming languages [103]), offloading energy-intensive functions to a remote server [78], using specialized-network protocols [8], or switching different algorithms according to pre-defined energy consumption scenarios [59]. While the majority of these efforts focused on one particular system layer (i.e., mainly the network), advanced techniques have been proposed to utilize multiple levels of system information, a technique called a cross-layer approach. A cross-layer approach can effectively control energy consumption by leveraging the information provided by multiple system layers. DYNAMO [99] is a middleware platform that adapts power optimization strategies across various system layers, including applications, middleware, OS, network, and hardware, to optimize both performance and energy. The focus of DYNAMO is on reducing energy consumption for video streaming applications. Our application energy consumption patterns can provide empirical results to efforts such as DYNAMO, which can interpret and apply them to specific application domains.

A recent language-based approach to energy-aware programming is ET [24], a new object-oriented programming language that enables the programmer to write energy-aware code by specifying *phases*, which represent distinct program workloads, and *modes*, which represent required energy states, such as high and low energy consumption. A language like ET can be a useful tool for distributed application programmers who want to take advantage of our DPA energy consumption patterns.

2.2.6 Discussion

The benchmark results presented above gave rise to the following two insights: 1) the latency/bandwidth characteristics of mobile networks can heavily affect the energy consumption of a mobile application and 2) adapting the execution behavior of a DPA in response to changes in latency/bandwidth can reduce the overall energy consumption. Based on these two basic insights and the presented guidelines, DPA designers should be able to create novel, energy-aware DPAs. In the following discussion, we give a concrete example of how such an energy-aware DPA can handle adaptive energy optimization.

Example: Adaptive Data Marshaling

Data marshaling refers to the process of encoding program data into a format that can be transferred across the network. For example, an integer value can be encoded as a byte buffer. The unmarshaling process reverses the marshaling encodings. Multiple marshaling strategies can be applied to the same program data. With respect to energy consumption, one can consider the trade-off between CPU processing and network transfer. Marshaling the data into a smaller byte buffer will reduce network transfer, but will be more computationally intensive, thus requiring additional CPU processing. Marshalling the data into a larger byte buffer will result in transferring more data over the network, but it will require less CPU processing. Which of the strategies will consume less energy depends on the runtime conditions in place.

For example, if the underlying network is limited (high latency and low bandwidth as in a congested network), transferring data concisely should be preferred, as it would reduce the energy consumed by network processing. In case of high-throughput networks, simple serialization protocols can reduce the overall energy consumption, as it would require less CPU processing while transferring larger volumes of data without the energy consumption spikes due to insufficient bandwidth. Therefore, the right trade-offs can only be determined at runtime, as it depends on the current network conditions. Whether to compress the transferred data presents another trade-off between data size vs. processing overhead. Similar to basic marshaling, algorithmically intensive data compression or delta calculation is computationally intensive, while reducing the amount of data transferred across the network.

In summary, the discussion above presented several high-level guidelines that can be applied to designing energy-aware DPAs. A key insight is that because each mobile application has different execution patterns and environments, applying a single energy-optimization strategy to all execution patterns for all network conditions is ill-advised. Thus, mobile application programmers must understand the execution patterns of mobile applications to be able to implement and configure application-specific, dynamic energy optimization strategies.

2.2.7 Conclusion

Due to the advantages provided by services, SaaS has entered the mainstream of commercial software development and a growing percentage of computing functionality is becoming accessible as a service. The programmers who need to access remote services are faced with the challenges of choosing an appropriate DPA for the task at hand. To assist the programmers in their decision process, in this work, we described a case study that compared the performance, expressiveness, and reliability of five different middleware platforms for accessing services remotely. Furthermore, we measured the energy consumption in DPA mechanisms. By systematically measuring and analyzing the constituent parts of major DPA mechanisms, we have identified their energy consumption patterns. Our measurements and analysis not only help the programmers in choosing between different DPAs, but also can inform the design of new DPAs for software designers and distributed system researchers.

Chapter 3

Hardening Distributed Applications Against Partial Failure

Refactoring an existing centralized program for distributed executions should preserve its original execution behavior, so that a distributed application would provide the same or equivalent functionality to the user as the original centralized version. In other words, distributing an application should not only improve its performance, availability, scalability, and energy efficiency, but also furnish the original functionality to the user. However, distributed applications are subject to partial failure, where components of a distributed system can fail independently of each other. In this chapter, we introduce a novel approach to hardening distributed applications in a disciplined and systematic fashion. First, in Section 3.1 we present a systematic approach to hardening distributed components to become resilient against network volatility. Then, in Section 3.2 we present a declarative approach to improving the reliability of distributed applications.

3.1 Hardening Distributed Applications with Network Volatility Resiliency

As the world is becoming more interconnected, our daily existence depends on a variety of network-enabled gadgets. Smart phones, PDAs, GPSs, netbook computers, all run network applications. Many of these gadgets are connected to a wireless network such as Wi-Fi. Despite the significant progress made in improving the reliability of wireless networks in recent years, real-world wireless environments are still subject to *network volatility*—a condition arising when a network becomes temporarily unavailable or suffers an outage. Usually the network becomes operational again within minutes of becoming unavailable.

Volatility is a permanent presence of many network environments for several reasons. For one, Wi-Fi networks transmit radio signals, which are volatile, often making it impossible to reach a 100% reliability. Another condition causing network volatility is congestion, which occurs when radio channels interfere with each other or multiple data is transmitted concurrently over the same radio link [60]. Furthermore, wireless networks are rapidly becoming available in emerging markets (e.g., such as in rural or remote areas), which cannot always rely on the existence of an advanced networking infrastructure [170].

Despite its temporary nature, network volatility can prove extremely disruptive for those distributed applications that are built under the assumption that the underlying network is highly-reliable, and network outages are a rare exception rather than a permanent presence. This could happen, for example, when a distributed application, built for a LAN, is later executed in a wireless environment.

Distribution middleware provides a set of abstractions through a standardized API that hide away various complexities of building distributed systems, including the need for low-level network programming. Distributed component systems such as DCOM [94], CORBA CC [106], and R-OSGi [120] expose network volatility as system-level exceptions that are handled by the programmer in an application-specific fashion. Thus, the programmer writes custom exception-handling code that is difficult to keep consistent, maintain, and reuse.

If the underlying network is expected to be volatile during the execution of a distributed system, a consistent strategy can be beneficial for handling the cases of network outages. Manually written outage handling code makes it difficult to ensure that a consistent strategy be applied throughout the application. Since the outage handling code is also scattered throughout the application, it can create a serious maintenance burden. Finally, the expertise developed in handling outages in one distributed application becomes difficult to apply to another application, with a copy-and-paste approach being the only option.

This paper argues that it is both possible and useful to handle network outages systematically, in a consistent and reusable way. Although software architecture researchers have outlined approaches to continue distributed application execution in the presence of network outages, these approaches are difficult to implement, apply, and reuse.

This work builds upon these approaches to define *hardening strategies*, which are exposed as reusable components that can be seamlessly integrated with an extant distributed component infrastructure. These reusable and customizable components can be added to an existing distributed component application, thereby hardening it against network volatility.

As our experimental platform, we use R-OSGi—a state-of-the-art distributed computing infrastructure that enables service-oriented computing in Java. We have created an extensible framework—DR-OSGi—which can harden any R-OSGi application, enabling it to cope with network volatility. DR-OSGi provides programming abstractions for expressing hardening strategies, which can also be reused across applications. The programmer selects a hardening strategy that is most appropriate for a given R-OSGi application and its deployment environment. DR-OSGi then handles all the underlying machinery required to harden the R-OSGi application with the selected strategy.

In our experiments, we have executed several realistic R-OSGi applications in a simulated networking environment to which we injected periodic network outages. By comparing the execution of the original and hardened versions of each application, we have assessed their respective ability to complete the execution, the total time taken to arrive to a result, and the overhead of the hardening functionality. Our results indicate that it is feasible and useful to systematically harden existing distributed component applications with the ability to cope with network volatility. Based on our results, the technical contributions of this paper are as follows:

- A clear exposition of the challenges of treating the ability to cope with network volatility as a separate concern that can be expressed modularly.
- An approach for hardening distributed component applications with resiliency against network volatility.
- A proof of concept infrastructure implementation—DR-OSGi—which demonstrates how

existing distributed component applications can be hardened against network volatility.

In the following discussion, we first look at network volatility from the networking perspective. Then we outline the concepts and technologies used in implementing our framework.

3.1.1 Background

Network Volatility

Modern computing networks are sophisticated multi-component systems whose reliability can be affected by hardware and software failure. These failure conditions include random channel errors, node mobility, and congestion. The reliability of a wireless network can be additionally afflicted by the contention from hidden stations and frequency interference [46, 62].

To improve the performance and reliability of modern networks, researchers have investigated various solutions, including congestion control, error control, and mobile IP. Most of these solutions improve various parts of the actual networking infrastructure. This work, by contrast, is concerned with solutions that treat network volatility as an unavoidable presence to be accommodated in software at the application level.

Software Components

A software component is an abstraction that improves encapsulation and reusability, thus reducing software construction costs. Typically a component encapsulates some unit of functionality that is accessed by outside clients through the component's interface. Component interfaces tend to remain stable, evolving infrequently and systematically. This reduced coupling between a component and its clients makes it possible to change the component's underlying implementation without having to change its clients. Examples of software component architectures include COM [94], CORBA CC [106], CCA [1], and OSGi [107].

OSGi For our reference implementation, we have chosen a mature software component platform for implementing service oriented applications called OSGi [107]. Among the reasons for choosing OSGi is its wide adoption by multiple industry and research stakeholders, organized into the OSGi Alliance [107]. OSGi is used in large commercial projects such as the Spring framework and Eclipse, which uses this platform to update and manage plug-ins. The OSGi standard is currently implemented by several open-source projects, including Apache Felix, Knopflerfish, Eclipse Equinox, and Concierge[119]. OSGi provides a platform for implementing services. It allows any Java class to be used as a service by publishing it as *a service bundle*. OSGi manages published bundles, allowing them to use each other's services. OSGi manages the lifecycle of a bundle (i.e., moving between install, start, stop, update, and delete stages) and allows it to be added and removed at runtime.

R-OSGi Despite its versatility, OSGi only allows inter-bundle communication within a single host. To support distributed services via OSGi, the R-OSGi distributed component infrastructure was introduced [120]. R-OSGi enables proxy-based distribution for services, providing proxies also as standard OSGi bundles. An R-OSGi distribution proxy redirects method calls to a remote bundle via a TCP channel, supporting both synchronous and asynchronous remote invocations. R-OSGi also provides a distributed service registry, thus enabling the treatment of remote services uniformly with local services.

Thus, R-OSGi introduces distribution transparently, without modifying the core OSGi implementation. It can even enable remote access to an existing regular OSGi bundle, transforming the bundle into a remote service. The transformation employs the concept of *the surrogation bundle*, which registers the service and redirects remote calls to the original bundle.

With respect to network volatility, R-OSGi treats it similarly to other distributed component infrastructures. Specifically, in response to a network disconnection, a client accessing a remote R-OSGi service will receive an exception. The programmer can then write custom code to handle the exception.

Hardening Strategies to Cope with Network Volatility

When the underlying network fails, a distributed application will typically signal an error to the end user, who can then decide on how to proceed. The user, for example, could choose to check the network connection and restart the application. The purpose of hardening strategies is to enable a distributed application to continue executing when the underlying network becomes unavailable. In a recent publication, Mikic-Rakic and Medvidovic classify disconnected operation techniques as well as how they can be applied to improve the overall system dependability [97]. Next we outline these techniques and discuss how they can be applied to harden a distributed component application to cope with network volatility.

Caching—This strategy employs caching techniques to store a subset of remote data locally, so that it could be retrieved and used by remote service requests when the network becomes unavailable. The effectiveness of this strategy depends strongly on the hit rate of the caching scheme in place. That is, since the size of any cache is always limited, the main challenge becomes to cache the remote data that is most likely to be needed by a service invocation when the network is unavailable. This strategy can in effect fail completely if there is a cache miss.

Hoarding—This strategy prefetches all the remote data needed for successfully completing any remote service invocation. It assumes, however, that data alone is sufficient for invoking a remote service. Unfortunately, this assumption fails for any resource-driven distribution—collocating hardware resources with the code and data they use. For example, a remote sensor has to operate at a remote location from which it is collecting data; hoarding any amount of the sensor's output data will fail to provide up-to-date sensor information upon disconnection. Thus, a hoarding-based strategy can be effective only when computation is distributed for performance reasons, and computation with a given data input yields the same results on any network node. These execution properties are often exhibited by high-performance cluster environments that use distribution to improve performance.

Queuing—This strategy intercepts and records remote requests made to an unreachable remote service. The recorded requests are then replayed when the service becomes available. This technique can only work if the results of a remote call are not immediately needed by the client code (e.g., to be used in an `if` statement). Otherwise, the client code will block, not being able to benefit from this strategy. Queuing is also poorly applicable for realtime applications.

Replication—This strategy maintains a local copy of a remote component. When the remote component becomes unreachable, the local copy is used. If the replicated component is stateful, then the states of the local and remote copies have to be kept consistent. When the network is available, client requests can be multiplexed to both local and remote copies. Alternatively, a consistency protocol can be used. Upon reconnection, the remote copy has to be synchronized with the local copy. This strategy has the same applicability preconditions as hoarding.

Multi-modal components—This strategy employs several of the strategies above and can apply them either individually, based on some runtime condition, or together, combining some features of individual strategies. For example, both caching and queuing can be used, depending on which remote service method is invoked. Similarly, replication can be applied to remote components while hoarding the data used by the replicated components.

Aspect-oriented Programming and JBoss AOP

This work aims at treating network volatility resiliency as a distributed cross-cutting concern. A powerful methodology for modularizing cross-cutting concerns is aspect oriented programming (AOP)[71]. We believe that network volatility resiliency is similar to other cross-cutting concerns such as logging, persistence, and authentication—essential functionality, but not directly related to the business logic.

AOP modularizes cross-cutting concerns and weaves them into the application at compile-time, load-time, or runtime. Major AOP infrastructures include AspectJ[70], Spring AOP[134], and JBoss AOP[64]. Some AOP technologies have even been applied to OSGi, including the Eclipse Foundation's AspectJ plug-in and Equinox. For our purposes, we needed to weave in the outage handling functionality at runtime, which typically requires modifying the JVM or rewriting the bytecode. We also needed the ability to modify the parameters of a remote service method. Among the major AOP systems, only JBoss AOP provides all the required capabilities. Another draw of JBoss AOP is that it does not either introduce a new language, thus flattening the learning curve, or changes the JVM, thus ensuring portability.

3.1.2 DR-OSGi: Treating Symptoms of Network Volatility

Our reasoning behind the name DR-OSGi—our reference implementation of an infrastructure for systematic handling of network volatility—is our skeptical view of the power of modern medicine. Despite all its impressive accomplishments, modern medicine can only treat some of the symptoms of the majority of known diseases—it cannot eliminate the disease itself. Take common cold as an example. They say that "If you treat a cold, it takes seven days to recover from it, but if you do not, it takes a week." When a cold is concerned, modern medicine can only help eliminate its symptoms, such as fever, sneezing, and coughing, thereby improving the patient's quality of life.

By analogy, we treat network volatility as a disease—an annoying but unavoidable condition that cannot be eliminated. All we want to do is to treat the symptoms of this disease systematically. By helping the patient (a distributed system) to effectively cope with the symptoms of network volatility (an inability to make remote service calls), we improve the patient's quality of life (QoS).

We next demonstrate our approach by showing how our approach can systematically harden distributed component applications against network volatility. In the following discussion, we first state our design goals, before presenting the architecture of our reference implementation and its individual components.

Design Objectives

Can any distributed component architecture be effectively hardened against network volatility? In other words, are there any special capabilities a distributed component architecture must provide to make itself amenable to hardening? For our approach to work, we assume that a distributed component architecture can detect and convey to the distributed application the following two scenarios:

1. **A remote service becomes unavailable**—this scenario should be effectively detected by the underlying distributed component architecture, so that an appropriate exception could be raised.

2. **A temporarily unavailable remote service becomes available again**—this scenario assumes that the component architecture does not "give up" trying to reach a remote service, periodically attempting to access it.

To the best of our knowledge, most distributed component architectures can effectively handle the first scenario. However, only advanced distributed component architectures can handle the second one. As a concrete example, R-OSGi employs the Service Discovery Protocol, which periodically attempts to reconnect to a remote service, if the service were to become unavailable. If, for example, a remote service becomes unreachable due to a network outage, the R-OSGi Service Discovery Protocol will keep trying to reach the service until the network connection is restored. It is these advanced capabilities of R-OSGi that convinced us to use this distributed component architecture as our experimentation platform.

Our system, called DR-OSGi, can harden existing R-OSGi applications to become resilient against network volatility. In designing DR-OSGi, we pursued the following goals:

1. **Transparency**—any hardening strategy should not affect the core functionality of the underlying R-OSGi application.

2. **Flexibility**—DR-OSGi should be capable of adding or removing the hardening strategies at any time without having to stop the application.

3. **Extensibility**—DR-OSGi should provide flexible abstractions, enabling expert programmers to easily implement and apply custom hardening strategies.

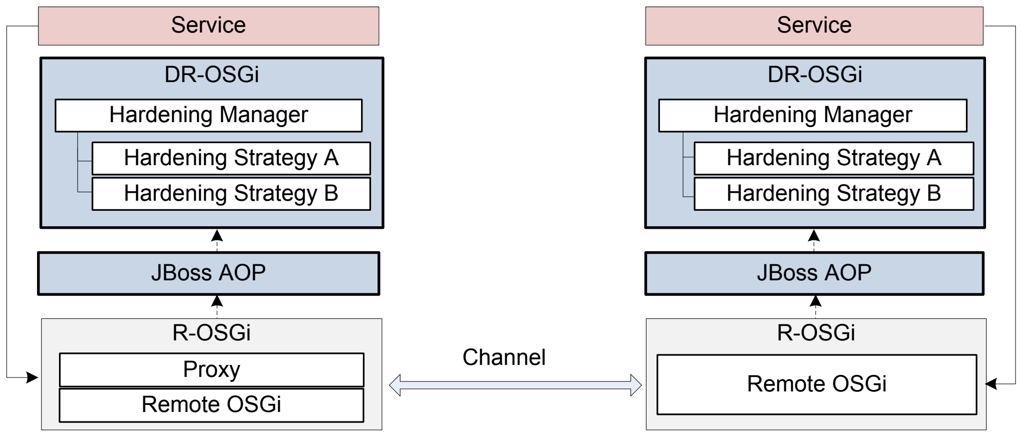


Figure 3.1: Hardened architecture.

Since the channels to a remote OSGi bundle use TCP, which provides reliable data transport, packet loss is handled at the transport layer. TCP, however, provides no assistance to deal with network volatility conditions arising as a result of link failure, node mobility, or high congestion. Therefore, to detect network instability or disconnection, an R-OSGi channel uses a timer to block the caller until the service has returned or the timeout has been exceeded. In the case of exceeding the timeout, an exception is thrown. R-OSGi handles such exceptions by having a remote OSGi bundle dispose of the channel and remove all proxies, preventing remote service calls while the network in unavailable. R-OSGi periodically checks whether the network has become available again and, if so, recreates the remoting proxies and channels.

DR-OSGi intercepts the handling of R-OSGi network-related exceptions and the successful completions of its reconnection attempts. Specifically, DR-OSGi handles R-OSGi network-related exceptions by triggering a hardening strategy. The type of the triggered strategy is determined by a programmer-specified configuration. The hardening strategy stops being applied when DR-OSGi intercepts a successful R-OSGi reconnection attempt.

Figure 3.1 shows how DR-OSGi is integrated into a typical R-OSGi application. DR-OSGi augments an R-OSGi application with a hardening manager and a collection of hardening strategies. The manager and each strategy are encapsulated in separate OSGi bundles. The hardening manager plugs into an R-OSGi application to intercept the handling of network exceptions and of the successful completions of reconnection attempts. In response to these events, the manager starts and stops the hardening strategies as configured by the programmer.

To integrate the hardening manager with an R-OSGi application without changing the application's source code, we employ Dynamic Aspect Oriented Programming (AOP). Because OSGi bundles are deployed at runtime, DR-OSGi has to be able to interpose the hardening logic dynamically. The dynamic AOP technology that fits our design objectives is JBoss AOP.

Programming Model

Next we detail the DR-OSGi programming model and demonstrate how it simplifies the creation and deployment of custom hardening strategies. To harden an R-OSGi application, the programmer has to provide a configuration file that specifies which hardening strategy should be applied to which application bundle. The following configuration file specifies that the application bundle `MyBundle` is to be hardened by the strategy implemented in the DR-OSGi-conformant bundle `CachingHardening`:

```
RemoteServiceName=org.mypackage.MyBundle
HardeningServiceName=org.otherpackage.CachingHardening
```

The simple syntax of the DR-OSGi configuration files is sufficiently expressive and supports wildcards which can be used to specify that a hardening strategy be applied to multiple bundles. Several hardening strategies can be applied to the same application bundle simultaneously. For example, remote invocations can be both cached and queued when the network is available. The programmer can specify in the configuration file which strategy bundle should be primary (i.e., to be applied first). If, when the network becomes unavailable, the first strategy succeeds, DR-OSGi does not apply the second one.

To implement a hardening strategy, the programmer needs only to implement interface `DisconnectionListener`, which is defined as follows:

```
public interface DisconnectionListener {
  public Object disconnectedInvoke(RemoteCallMessage invokeMessage);
  public Object reconnected(String uri);
  public void remoteInvoke(RemoteCallMessage invokeMessage, Object result);
  public void serviceAdded(String uri);
  public void serviceRemoved(String uri);
}
```

Method `disconnectedInvoke` is called by DR-OSGi, when R-OSGi detects that the network connection has been lost. Method `reconnected` is called by DR-OSGi, when R-OSGi manages to successfully reestablish a connection to a remote bundle. Finally, `remoteInvoke` is called when a remote service method has been successfully invoked. The implemented class has to be deployed as a regular OSGi bundle, and an entry describing the implementation must be added to the configuration file.

System Architecture

In the following we discuss the system architecture of DR-OSGi. The key objective of this work is to explore how network volatility hardening strategies can be implemented modularly and applied to an existing distributed component application that may have been written without fault-tolerance capabilities in mind. In other words, we argue that it is possible to treat hardening strategies as reusable software components, which can be developed by third-party programmers and reused across multiple applications.

Figure 3.2 shows how we have designed DR-OSGi, so that it could naturally integrate with the existing OSGi and R-OSGi infrastructures. DR-OSGi makes use of existing OSGi services such as `Service Registration` and `Service Tracker`. Every DR-OSGi component, including the hardening manager and all hardening strategies, register themselves with OSGi, which manages them as standard registered services. This arrangement makes it possible to locate DR-OSGi components using the OSGi `Service Tracker` and load them on demand.

To receive service change events from OSGi, the hardening manager implements the

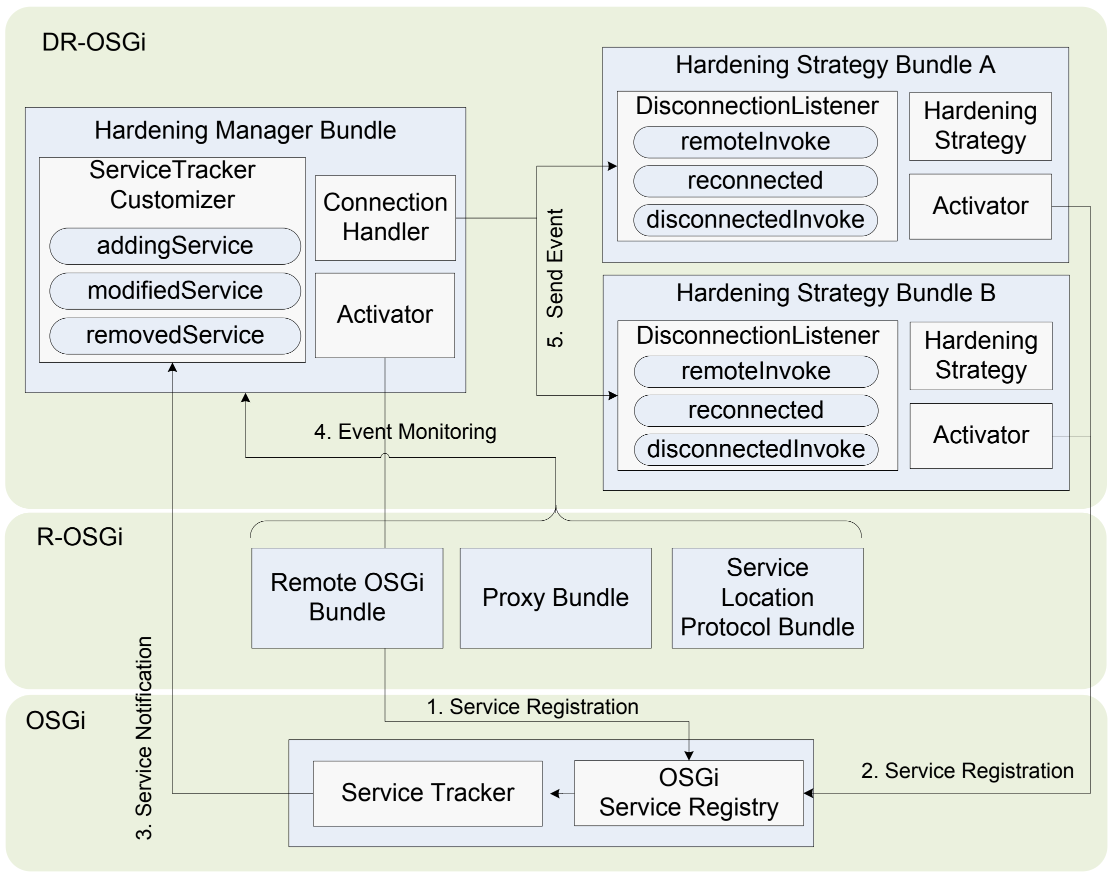


Figure 3.2: DR-OSGi design.

`ServiceTrackerCustomizer` interface, which is discussed below. In turn, to make it possible for the manager to send the relevant events to hardening strategy bundles, each bundle implements the `DisconnectionListener` interface. All the lifecycle events in DR-OSGi are triggered by sending and receiving events, with `Service Tracker` and `Service Registration` enabling the hardening manager and hardening bundles to be loosely coupled.

When a new hardening strategy is deployed, OSGi sends an event—`addingService`—to `Service Tracker`, which then forwards the event to the hardening manager by calling the corresponding `ServiceTrackerCustomizer` interface method.

```
public interface ServiceTrackerCustomizer {
  public Object addingService(ServiceReference reference);
  public void modifiedService(ServiceReference reference, Object service);
  public void removedService(ServiceReference reference, Object service);
}
```

The hardening manager keeps track of which hardening strategies have been registered and maintains a searchable repository of all the registered strategy bundles.

Weaving in Resiliency Strategies with Aspects

To intercept the disconnection/reconnection procedures of R-OSGi, without changing its source code, we use dynamic Aspect Oriented Programming technology, JBoss AOP. The ability to apply aspects dynamically is required due to OSGi loading bundles dynamically at runtime. JBoss AOP makes use of XML configuration files that specify at which points aspects should be weaved. Using AOP enables DR-OSGi to keep its implementation modular and avoid having to modify the source code of R-OSGi.

3.1.3 Evaluation

To evaluate the effectiveness and performance properties of DR-OSGi, we have conducted three benchmark experiments and a larger case study.

Benchmarks

Since R-OSGi can easily distribute any existing OSGi application, our benchmarks use third-party OSGi components accessed remotely across the network. As our benchmark applications, we have used a remote log service, a remote user administration service, and a distributed search engine.

To create a controlled networking environment with predictable network outage rates, we have used a network emulator—`netem` [2]—to introduce network volatility conditions, including transmission delay, packet loss, packet duplication, and packet re-ordering.

In our experimental setup, we have emulated a network with the round trip time (RTT) metrics equal to 14ms, which is typical for a modern wireless network. To emulate network outages, we used `netem` to generate packets losses at the server. Lossy network conditions were emulated by losing a high number of random packets (i.e., over 30% loss); totally disconnected networks were emulated by losing all the transmitted packets.

The experimental environment has comprised a Fujitsu S7111 laptop (1.8 GHz Intel Dual-Core CPU, 2.5 GB RAM) communicating with a Dell XPS M1330 laptop (2.0 GHz Intel Dual-Core CPU,3 GB RAM) via a IEEE 802.11g wireless LAN, with both laptops running the Sun's client JVM, JDK J2SE 1.6.0_13.

Log Service

For this experiment, we used a log service defined by the OSGi specification [107]. The OSGi log service records standard output and error messages printed during a bundle's execution. The service can be configured to log different amounts of messages by calling its `setLevel` methods (the higher the level, the more messages are logged).

Imagine needing to log messages generated by a remote service locally. In this experiment, we have used R-OSGi to access the existing log service of Knopflerfish, a popular, open-source implementation of OSGi. To enable remote access, we have used the surrogation bundle approach to register the existing log service.

Network volatility should not cause a remote log service to stop functioning. Logs are typically examined for a postmortem analysis, for which the actual time when the messages are written to a log file is not important, as long as the messages' timestamps reflect their actual origination time.

In our experiment, we used the log service to record 10 text messages generated consecutively without any delay. The network is available during the remote logging of the first 3 messages. Immediately after logging the third message, the network becomes totally disconnected. Then after the fifth message, the network connection is restored.

Table 3.1: Message delivery delay under a queuing hardening strategy.

Network condition	connection			disconnection		connection				
Message number	1	2	3	4	5	6	7	8	9	10
Sent log time(min:sec)	0:00	1:12	1:21	1:51	2:51	3:19	4:01	4:03	4:42	4:46
Received log time(min:sec)	0:00	1:15	1:21	3:20	3:20	3:20	4:02	4:05	4:42	4:46

We have executed this scenario under two setups: plain R-OSGi and DR-OSGi with a queuing strategy. Recall that queuing works by recording remote service calls when the network is unavailable and replays the recorded calls once the connection is restored. Under the original setup, the remote log service recorded only 8 messages (3 before the disconnection and 5 after). Two messages were lost irretrievably. The hardened version recorded all 10 messages.

Table 3.1 shows the delay for each message delivery. For the queued messages (columns 4 and 5), the delays is significantly higher than for the other messages. Despite the delay of the queued messages, all the messages are delivered in the order in which they are sent. Since real-time guarantees are not required, we can conclude that the hardening strategy has provided the requisite QoS for the remote log service, allowing it to cope with network volatility.

User Admin Service

For this experiment, we used the *User Admin Service*, which comes as a part of the core OSGi system services. The service authenticates and authorizes users by running their credentials against a database. Oftentimes, this service may need to be accessed remotely. To introduce distribution, we have registered the standard User Admin Service bundle using a surrogation bundle, similar to the approach we took in distributing the log service.

A network outage should not prevent a client from using the User Admin Service, if the client has used the service in the past, and the security policy specifies that user credentials change infrequently and can be cached safely. In other words, the caching hardening strategy must be coordinated with the security policy in place, lest the system's security can be compromised. One way to accomplish this is to avoid caching the authentication data that may change while the network is temporarily unavailable.

We have emulated a scenario in which 100 remote authentication attempts have been made across the network, which randomly suffers disconnections with the rate equal to 1 disconnection per 20 authentication attempts. Disconnections always cause the R-OSGi version of the application to fail. The ability of the DR-OSGi hardened version to continue executing depends on the number of clients. In this simulation, we assume that all the clients use the service equally. Thus, if for example, there were n authentication requests made from m users, then the expected number of authentications performed by a single user is n/m. Since the cache size is set to 5, the hit rate is negatively correlated with the number of users, standing at 100% for 2 and 4, and going down to 90%, 85%, and 78% for 6, 8, and 10 users, respectively.

Distributed Lucene

For this experiment, we have used Lucene, a widely-used Java search engine library. Among the capabilities provided by Lucene are indexing files and finding indexes of a given search word. Because searching is computationally intensive, there is great potential benefit in distributing the searching tasks across multiple machines, so that they could be performed in parallel.

Despite several known RMI-based Lucene distributions, for our experiments we have created an R-OSGi distribution, which turned out to be quite straightforward. We have followed a simple Master Worker model, with the Master assigning search tasks to individual Workers as well as collecting and filtering search results. This distribution strategy, depicted in Figure 3.3, requires that only the Master node be hardened against network volatility. This embarrassingly parallel data distribution arrangement imposes a strict one way communication protocol with the Master always calling Workers but never vice versa.

Once again, a caching hardening strategy has turned out to be most appropriate for hardening the distributed Lucene R-OSGi application. Specifically, every work assignment for individual nodes is used as a key mapped to the returned result. The intuition behind this caching scheme is that files are read-only and searching a file for the same string multiple times must return identical results. For writable files, the caching scheme would have to be modified to invalidate all the cached results

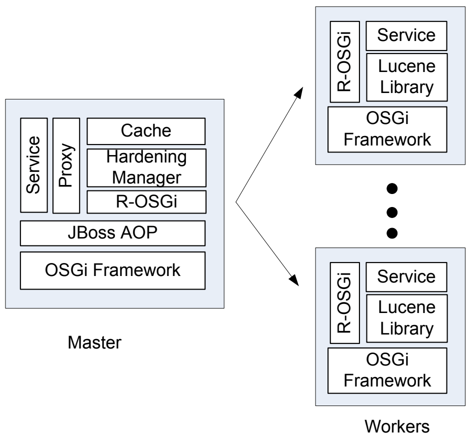


Figure 3.3: Distributed lucene.

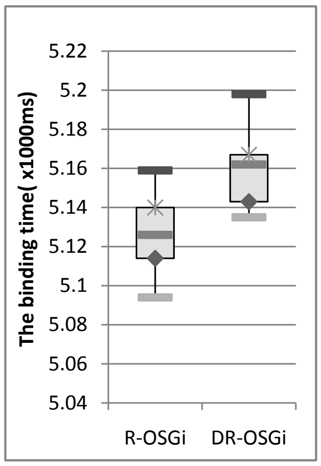


Figure 3.4: Binding time comparison.

for the changed files. As it turns out, the absolute majority of environments that use Lucene feature read-only files only, including digital books, scientific articles, and news archives.

Since distributed Lucene is representative of a large class of realistic applications, we have used it to assess the performance overhead imposed by DR-OSGi. The first benchmark has measured the binding time, which is defined as the total time expended on establishing a remote connection, requesting the service, receiving the interface, and building the remoting proxy. R-OSGi is quite efficient, with R-OSGi application consistently outperforming their RMI versions [120]. The purpose of our benchmark was to ensure that DR-OSGi does not impose an unreasonable performance overhead on top of R-OSGi. As it turns out, there is not a pronounced difference between the binding time of a plain R-OSGi version of Lucene and its hardened with DR-OSGi version, as shown in Figure 3.4. One could argue that binding is a one-time expense incurred at the very start of a service and as such is not critical.

To distill the pure overhead of DR-OSGi, we have measured the total time it took to synchronously invoke a remote service under three scenarios:

1. Running the original R-RSGi version with no network volatility present

2. Running the hardened with DR-OSGi version with no network volatility present

3. Running the hardened with DR-OSGi version with a randomly introduced complete network disconnection

The measurements are the result of averaging the total time taken by $1 * 10^3$ remote service invocations. To emulate a complete network disconnection, we have generated a 100% packet loss. While the original R-OSGi version takes 9043.5 ms to execute, the hardened one takes 9321.9 ms, thus incurring only 3% overhead when no volatility is present. When the network becomes unavailable, the DR-OSGi caching strategy improves the performance quite significantly, as it eliminates the need for any computation to be done by the worker node. While, somewhat unrealistically, we used the 100% hit rate to isolate the overhead of DR-OSGi, the actual performance is likely to vary widely depending on the applicaion-specific caching scheme in place.

These performance results indicate that the insignificant performance overhead that DR-OSGi imposes on a hardened distributed application can certainly be justified by the added resiliency to cope with network volatility.

Case Study: Hardening "DNA Hound"

As a larger case study, we have hardened "DNA Hound," a three-tier R-OSGi application for assisting detectives conducting a criminal investigation. The application works by automating the process of analyzing and warehousing DNA evidence, collected at crime scene investigation sites. Figure 3.5 depicts the architecture of "DNA Hound." The detective collects DNA evidence using a hand-held device, and then sends it to a search facility using a mobile data network (or any other wireless network). The search facility matches the sent DNA evidence against a database of DNA sequences (via parallel processing) and reports if a match is found. The collected DNA evidence is then sent to a crime evidence warehouse for storage.

We have implemented a complete working prototype of "DNA Hound," but in lieu of DNA extracting hardware, we simulated the found DNA evidence by randomly selecting DNA sequences from a GenBank NCBI database [13]. The search is performed using a parallelization of the Smith-

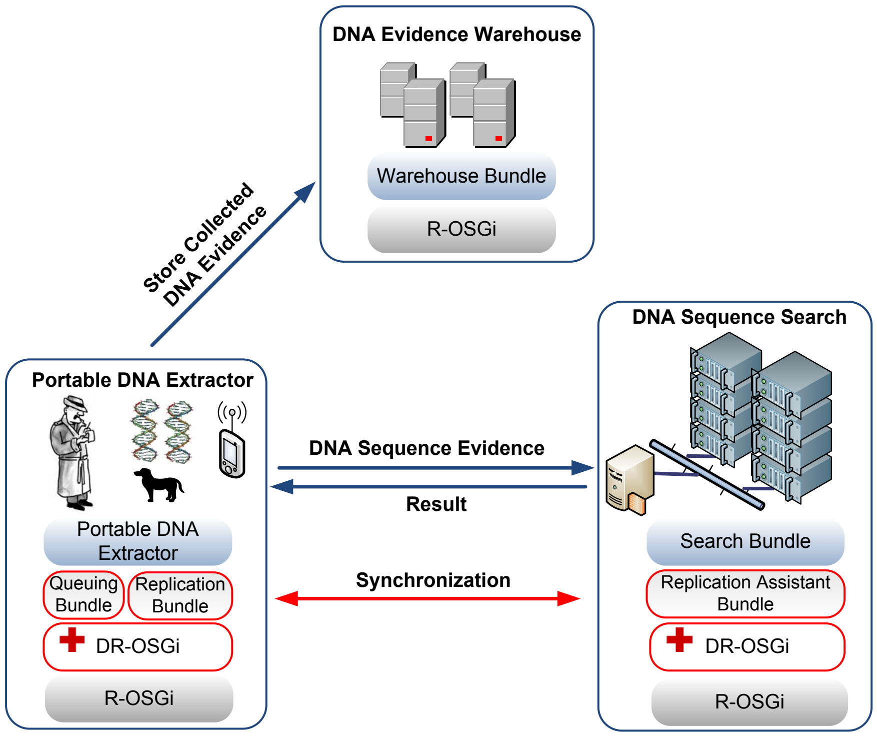


Figure 3.5: *DNA Hound* system architecture.

Waterman algorithm [132] on a compute cluster.

Because "DNA Hound" is used in the field, it relies on a wireless network that can be unreliable. Therefore, to ensure that the application continues to provide service, we have used DR-OSGi to harden it against network volatility. We have used two hardening strategies implemented as regular OSGi bundles.

Replication. To harden the application for the network volatility that can occur between the hand-held DNA extractor and the analyzer, we have used a replication strategy. Although DNA sequence search is very computationally-intensive, usually requiring parallel processing to shorten the search time, it can also be done sequentially, albeit much slower. With the advance in data storage technologies, even a hand-held device can comfortably store a substantial database of DNA sequences. The DNA search bundle is replicated at the hand-held device. We have used the native OSGi replication facilities to install the search bundle at both sites. When the network is up, the search is performed using a compute cluster at the search site, and the index of the most recently searched database sequence is periodically sent to the search bundle at the hand-held site. Once the network becomes unavailable, the search bundle at the hand-held site continues the search locally, using an inefficient sequential algorithm; the search is continued from the index of the last searched sequence at the cluster. If the index is not up-to-date, then some overlap in the search will occur. Once the connection goes back up, the cluster could then report any matches found while the network was not available.

Queuing. To harden the application for the network volatility that can occur between the hand-held DNA extractor and the criminal evidence warehouse, we have used a queuing strategy. The calls to store a new piece of DNA evidence are queued up at the hand-held site once the network becomes unavailable. Then the queued calls are resent to the warehouse once the network connection is restored.

The original R-OSGi version of the application was written without any functionality enabling it to cope with network volatility–it thus fails immediately once either network link is lost. DR-OSGi made it possible to harden this unaware application, so that it can meaningfully continue its operation in the presence of network volatility, thus improving the application's utility and safety. This demonstrates how DR-OSGi makes it possible to treat network volatility resiliency as a separate concern that can be implemented separately and added to an existing application. Furthermore, the queuing bundle came from the library of standard hardening strategy bundles that are part of our DR-OSGi distribution, thus requiring no programmer effort. The replication bundle was custom tailored for this application, but we are currently working on generalizing the implementation, so that only the synchronization functionality would require custom coding.

3.1.4 Discussion

The hardening approach of DR-OSGi is quite general and can be applied to a variety of distributed components. Although our reference implementation is dependent on R-OSGi and JBoss AOP, DR-OSGi relies only on their core features, which are common in other related technologies. Specifically, we leverage the ability of R-OSGi to convey network failure as application-level exceptions and to reestablish connections once the network becomes available. JBoss AOP effectively modularizes hardening strategies. Although our approach delivers tangible benefits to the distributed component programmers, it also has some inherent limitations.

Advantages DR-OSGi makes it possible to handle network volatility consistently throughout a distributed component application. This means that the most appropriate hardening strategy can be applied to any subset of application components, and the strategies can be switched through a simple change in the configuration file. Furthermore, each strategy is modularized inside a separate OSGi bundle, thus streamlining maintenance and evolution. Finally, modularized strategies can be easily reused across different distributed component applications.

Limitations Creating a pragmatic solution that can be implemented straightforwardly required constraining our design in several respects. For example, we chose to maintain a one-to-one correspondence between application bundles and their hardening strategy bundles. That is, a hardening strategy for all the services in a bundle must be implemented in a single DR-OSGi strategy bundle. Strategy bundle implementations, of course, can combine any hardening strategies. We have made this design choice to simplify the deployment and configuration of strategy bundles. Another limitation is inherited from JBoss AOP, which is loaded by the infrastructure irrespective of whether a hardening strategy will be applied, thus possibly consuming system resources needlessly. This may present an issue in a resource-scarce environment such as an embedded system. A possible solution to this inefficiency would be to extend OSGi with a meta-model that would allow the programmer to systematically extend services.

3.1.5 Related Work

DR-OSGi derives its hardening strategies from a recent survey of disconnected operation techniques by Mikic-Rakic and Medvidovic [97]. These techniques are used by several systems, including the Rover toolkit [65], Mobile Extension [27], Odyssey [102], and FarGo-DA [162]. Unlike these systems, DR-OSGi enables the programmer to harden distributed applications without having to modify their source code explicitly. By avoiding ad-hoc modification that can be tedious and error-prone, DR-OSGi not only hardens applications more systematically, but also enables greater reuse of the hardening strategies across different applications.

Aldrich et al.'s ArchJava [3] extends Java to integrate architectural specifications with the implementation by providing language support for user-defined connectors. Their techniques bears similarity to DR-OSGi in separating reusable connection logic from the application logic and integrating them together systematically. ArchJava, however, operates at the source code level, using its language extension to express different connectors. DR-OSGi is a middleware solution that does not need to modify the source code.

Sadjadi and McKinley's adaptive CORBA template (ACT) enables CORBA applications to adapt to unanticipated changes [123]. To do so, ACT employs a generic interceptor, a type of CORBA portable request interceptor [106] that works around the constraints of replying to intercepted requests or modifying the invoked method's parameters. Specifically, a generic interceptor forwards requests to a proxy, a CORBA object that can reply and modify the requests. Similarly to DR-OSGi, ACT introduces additional functionality to a distributed application without modifying its code explicitly. Using ACT to harden against network volatility, however, would require that portable interceptors be available, which may not be the case for many distributed component infrastructures including R-OSGi.

A number of techniques for making existing systems fault tolerant [58, 115, 139] are related to our approach. GRAFT [139] automatically specializes middleware for fault-tolerance. It employs the Component Availability Modeling Language (CAML) to annotate a distributed application's model, and then automatically specializes the application's middleware for domain-specific fault-tolerant requirements. While GRAFT requires that the programmer express the requested fault-tolerance functionality at the model level using a domain-specific language, DR-OSGi provides a simple Java API for implementing hardening strategies as OSGi bundles, which it then manages at runtime.

Our idea of hardening against network volatility was inspired by *security hardening*, a systematic approach to making a pre-existing program artifact more secure such as Wuyts et al's recent work [166]. Our approach hardens distributed components to become more resilient against network volatility.

3.1.6 Conclusions

In this section, we have presented DR-OSGi, a promising approach for systematically hardening distributed components to cope with network volatility. The reference implementation features an extensible framework for deploying hardening strategies, with caching, queuing, and replication used to demonstrate the effectiveness of our approach. As we rely on greater numbers of network-enabled devices with network volatility remaining a permanent presence, the importance of hardening distributed components will only increase, motivating the creation of systematic and flexible hardening approaches as showcased by DR-OSGi.

3.2 A Declarative Approach to Hardening Services Against QoS Vulnerabilities

The mainstream software paradigm has been transitioning from software-as-a-product (SaaP) to software-as-a-service (SaaS). SaaS is a computing modality that comprises a collection of services—encapsulated units of computing functionality accessed by clients through public interfaces. Distributed service-oriented applications—accessed remotely across the network—are rapidly becoming the preferred building blocks for the majority of modern computing domains. The popularity of distributed service-oriented applications stems from the general software engineering benefits of SaaS, including low coupling, strong encapsulation, ease of discovery, and reduced maintenance costs.

Despite their numerous benefits, distributed services may not provide the requisite levels of reliability and security, particularly when operated in volatile network environments, attacked by hackers, or not properly maintained and managed. To that end, this paper describes a new approach that can harden distributed service-oriented applications against three major threats to the QoS: (1) network volatility—services are often accessed through disconnected and limited networks, for which the service must be properly adapted; (2) security exploits—a distributed service-oriented application can be exploited by an adversary for nefarious purposes, and must be protected against all known and future exploits; (3) administrative mismanagement—a distributed service-oriented application and its clients may be upgraded according to conflicting schedules, thus requiring runtime adaptation to avoid service protocol mismatches.

To harden distributed service-oriented applications against the important vulnerability classes described above, we propose *declarative hardening*. Specifically, we design and implement a DSL for expressing *hardening policies*. DSL combine high expressiveness, conciseness, and simplicity by providing constructs that are custom tailored for a given domain. In our case, the target domain is hardening distributed service-oriented applications. A service compiler translates the policies to a hardening components for a target service infrastructure in place. Finally, a hardening framework should be able to integrate the generated hardening components with a distributed service-oriented application, thereby equipping it with the capacity to counteract the specified vulnerabilities. Thus, this approach harmoniously combines several state-of-the-art technologies to address an important set of vulnerabilities that plague distributed service-oriented applications.

The uniqueness of the proposed approach lies in the following advantages over the current state of the art: (1) a declarative approach—we introduce a domain-specific language for describing both service vulnerabilities and the hardening strategies to eliminate them; (2) a compilation of declarative hardening specifications—our compiler is capable of generating working code for Open Service Gateway Initiative (OSGi) service infrastructure; (3) reusable hardening components—strategies are reusable across multiple applications and domains; (4) separation of concerns—reliability/security specialists can focus on their respective areas of expertise.

As our experimental platform, we use a well-known distribution middleware system—CXF-DOSGi—

which enable service-oriented computing in Java. We have created a hardening framework which can harden any remote OSGi application, enabling it to cope with network volatility, security exploits, and mismanaged API. We provide a new programming language for expressing hardening policies and strategies, which can also be reused across applications. The programmer describes hardening policies and strategies. Then, the hardening framework handles all the underlying machinery required to harden the remote OSGi application.

In our experiments, we have executed a realistic OSGi application to measure efficiency and performance. By comparing the execution of the original and hardened version, we have assessed their respective ability to complete the execution, the total time taken to arrive to a result, and the overhead of the hardening functionality. Our results indicate that it is feasible and useful to systematically harden existing service oriented applications with the ability to cope with vulnerabilities.

The rest of this section is structured as follows. Section 3.2.1 introduces the concepts and technologies used in this work. Section 3.2.2 describes our proposed approach, including the proposed hardening language and hardening framework. Section 3.2.3 evaluates the utility and efficiency of the proposed approach through a performance benchmark and a case study. Section 3.2.4 discusses advantages and limitations of this research. Section 3.2.5 compares our approach to the existing state of the art. Finally, Section 3.2.6 presents future research directions and concluding remarks.

3.2.1 Background

In the following discussion, we first describe service oriented architectures and their distributed versions. Then, we introduce the three types of vulnerabilities that can affect the QoS of distributed service-oriented applications and are addressed by our approach.

Service Oriented Architecture

Service-Oriented Architecture (SOA) has been recently employed as a means of providing uniform access to a variety of computing resources across multiple application domains. In SOA, software components are provided as services, self-encapsulated units of functionality accessed through a public interface. The core principles of SOA can be summarized as follows [38]:

- **Loose Coupling:** Services minimize dependencies and are aware only of each other.
- **Abstraction:** Services abstract away their underlying implementation details from their clients.
- **Reusability:** Services provide reusable functionality.
- **Autonomy:** Services control their environment and resources to provide consistency and reliability during the execution.
- **Statelessness:** Services avoid maintaining any state to facilitate failure recovery and minimize resource consumption.
- **Discoverability:** Services can be effectively discovered and interpreted through standard protocols.
- **Composability:** Services compose effectively regardless of their size and complexity.

This work uses the following service technologies.

OSGi The Open Service Gateway Initiative (OSGi) provides a platform for implementing services [107]. It allows any Java class to be used as a service by publishing it as a service bundle. OSGi manages published bundles, allowing them to use each other's services. OSGi manages the lifecycle of a bundle (i.e., moving between install, start, stop, update, and delete stages) and allows it to be added and removed at runtime.

OSGi is a mature software component platform. It has been widely adopted by multiple industry and research stakeholders, organized into the OSGi Alliance. OSGi is used in large commercial projects, including the Spring framework[1] and Eclipse[2], which use this platform to update and

[1] `http://www.springsource.org/`
[2] `http://www.eclipse.org/`

manage plug-ins. The OSGi standard is currently implemented by several open-source projects, including Apache Felix[3], and Knopflerfish[4]. Despite its versatility, OSGi was mainly used for inter-bundle communication within a single host.

OSGi Remote Services Recently, the OSGi alliance released the OSGi R4.2 specification that describes how remote OSGi services can be discovered and used [107]. The OSGi R4.2 specification does not specify how remote OSGi services should be accessed. Instead, the specification codifies only how remote service interfaces should be discovered and retrieved. Once a remote service interface is obtained, it is up to the implementor of this specification how interface methods are to be invoked at a remote OSGi framework and how their results are to be transferred back to the caller. The first reference implementation of R4.2 is Apache CXF-DOSGi[5], which implements the specification as Web services, using SOAP over HTTP for transmission and WSDL contracts for exposing services. In addition, RBI-OSGi [84] is the first non-RPC implementation of the OSGi R4.2 specification. RBI-OSGi does not require any changes to remote service interfaces, which are discovered and bound using a standard OSGi registry. Furthermore, R-OSGi [120] that was introduced prior to the standard OSGi remote services enables proxy-based distribution for services, providing proxies as OSGi bundles.

QoS Vulnerabilities

A distributed service-oriented application becomes vulnerable to several threats. Specifically, the network connecting remote services may be subject to volatility—temporary network outages. Rendering a service remotely accessible can make it vulnerable to security exploits—although security is a vast research area, here we focus only on the security issues pertaining to distribution. A recent study has determined that out of the known 39 OSGi vulnerabilities, as many as 20 vulnerabilities (e.g., exposing internal representation, flaws in parameter validation, and invalid

[3] `http://felix.apache.org/`

[4] `http://www.knopflerfish.org/`

[5] `http://cxf.apache.org/distributed-osgi.html`

work flow) [110] are specific to accessing services remotely. Finally, when services are remote to each other, they can evolve independently, thus causing version inconsistency problems.

Network Volatility A remote service can be accessed through various networks, which are subject to network volatility due to various conditions such as random channel errors, node mobility, and congestion. For example, WiFi networks transmit radio signals, which are volatile, often making it impossible to reach a 100% reliability. Another condition causing network volatility is congestion, which occurs when radio channels interfere with each other or multiple data is transmitted concurrently over the same radio link. When the underlying network fails, a distributed service-oriented application will typically signal an error to the end user, who can then decide on how to proceed. The user, for example, could choose to check the network connection and restart the application. The purpose of hardening strategies is to enable a distributed service-oriented application to continue executing when the underlying network becomes unavailable. A recent survey [97] classifies disconnected operation techniques as well as how they can be applied to improve the overall system dependability. Specifically, the most common disconnected operations are: *caching*—that employs caching techniques to store a subset of remote data locally, so that it could be retrieved and used by remote service requests when the network becomes unavailable; *hoarding*—that prefetches all the remote data needed for successfully completing any remote service invocation; *queuing*—that intercepts and records remote requests made to an unreachable remote service, and the recorded requests are then replayed when the service becomes available; and *replication*—that maintains a local copy of a remote component, so that when the remote component becomes unreachable, the local copy is used.

Security Exploits The openness engendered by SaaS is a double-edged sword. On the one hand, any client can access a distributed service-oriented application through its public interfaces. On the other hand, unless a proper authentication scheme is put in place, the distributed service-oriented application can become exposed to malicious clients. According to the literature [110], distributed service-oriented applications are particularly vulnerable to the following threats:

- Software Defects can cause faults and failures in distributed service-oriented applications. For example, a logic bug may allow clients access certain methods without authentication (i.e., bypassing the invocation of `authenticate()`). Such bugs expose systems to security exploits that can render services unavailable. Eventually, the bugs should be fixed by modifying the source code, but a hardening strategy can also be developed to handle such software defects. For example, recently proposed approaches accomplish that through runtime verification which monitor systems and synthesize corrective functionality [19].

- Improper Parameter Validation exposes service methods to malicious clients that can pass illegal parameters, thus leading to undesirable outcomes. For example, accessing a parameter exceeding the available memory can render the service unavailable.

- Invalid Access has two types of vulnerability patterns—exposing internal representation and accessing from malicious clients. Exposed internal representation enables to execute code that should be hidden, thereby triggering unexpected system behavior or enabling malicious clients to access sensitive data. In addition, remote services should be protected from malicious clients. To deal with such invalid accesses, authorization and access control are commonly used techniques.

Service Mismanagement Service-oriented applications rely on loosely-coupled remote interfaces, each of which could evolve independently. When the vendor releases a new version of the application, the users can update different services at different times. As a result, the device can try to communicate with the old service interface. If the service interface has changed, the requested service methods may no longer be available.

Domain-Specific Language and Security Policies

A domain-specific language (DSL) is a programming language designed to solve problems in a particular domain. Compared to general-purpose languages (e.g., C, C++, Java, etc.) DSLs are custom tailored for the domain at hand, providing expressivenesses and ease of use advantages.

DSL encapsulates its domain expertise, making it easier for non-expert programmers to craft effective solutions for problems in the target domain.

In security research, domain-specific policy languages have been proposed to describe authorization and access control [28, 67]. These languages address the poor fit of general purpose languages to describe all the numerous low-level security issues that occur at the systems level. Equipped with such a DSL, programmers can easily express sophisticated security configurations.

3.2.2 Declarative Hardening

Next, we first outline our proposed solution and then describe our hardening language and hardening framework, respectively.

Solution Overview

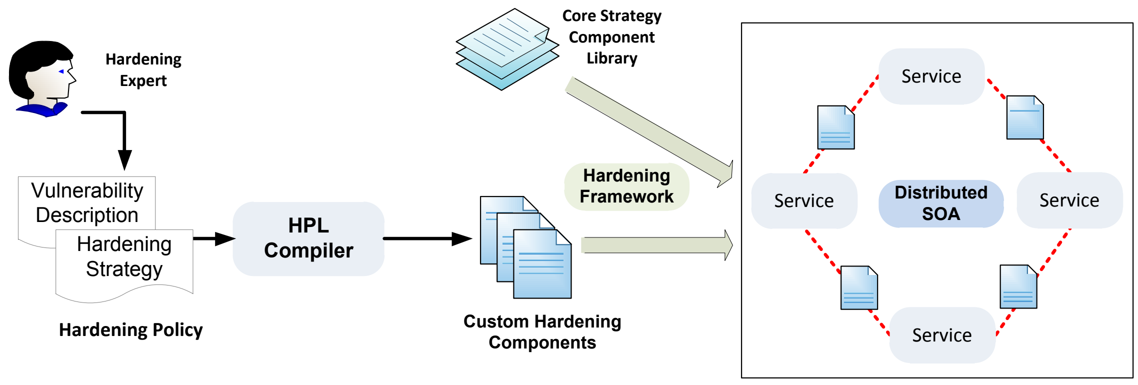


Figure 3.6: Approach overview.

To harden distributed service-oriented applications against the important vulnerability classes described above, we propose *declarative hardening*. Figure 3.6 depicts how our approach leverages the expressive power of DSLs, flexibility of the service compilation, and the adaptivity of the hardening framework. Specifically, we design and implement a DSL for expressing *hardening policies*. DSL combine high expressiveness, conciseness, and simplicity by providing constructs that are custom tailored for a given domain. In our case, the target domain is hardening distributed service-oriented applications. A service compiler translates the policies to a hardening components for a target service infrastructure in place. In our case, the target service infrastructure is the OSGi framework. Finally, our hardening framework seamlessly integrate the generated hardening components with a distributed service-oriented application, thereby equipping it with the capacity to counteract the specified vulnerabilities. Thus, our approach harmoniously combines several state-of-the-art technologies to elegantly address an important set of vulnerabilities that plague distributed service-oriented applications.

Hardening Policy Language—*HPL*

One of the key novelties of our approach is using a DSL for describing vulnerabilities and their hardening strategies. We call our language *Hardening Policy Language (HPL)*. In designing HPL, we aim at combining both expressiveness and ease of use. The specific design goals include:

- *Expressiveness*—a reliability/security expert should be able to express any kind of vulnerability easily, with the resulting code being easy to understand, maintain, and evolve.
- *Extensibility*—it should be possible to integrate existing security and reliability policies with HPL policies.
- *Platform Independence*—HPL policies should be platform independent, with the same policy compilable to any service platform.

Figure 3.7 shows how the HPL is constructed. To provide fault-tolerance and security defense to distributed service-oriented applications, what the programmer should do is only to write a policy script in HPL. First of all, hardening policies consist of five types of policy, including Service Configuration, Network Hardening, Security Hardening, Service API Hardening, and Hardening Strategy. Each policy mainly consists of a set of conditions which describes specific vulnerable situations and applicable hardening strategies. In the following sections, we detail how our HPL can effectively express remote services, vulnerabilities and hardening strategies.

```
Policy (ServiceConfig | NetworkHardening | SecurityHardening
       | ServiceAPIHardening | HardeningStrategy)
Begin
   PolicyName => [name] ;
   ServiceName => [name] ;
   Config => ([config_types] is [type])+ ;

   Condition => //Network Volatility
     (NetworkEvent([event]))
       When (Execution | Call) [From | To] [flow])+ ;
     Config => ([config_types] is [type])+ ;
     Then => ([strategy]) ;

   Condition => //Security Exploits
     ((Execution | NotExecution | Call | NotCall)
       [flow] [From | To] [url])+ ;
     ((ParamChecking [flow] [From | To] [url]
       Using Strategy [strategy])+ ;
     (Access From [url])+ ;
     Then => ([strategy]) ;

   Condition => //Mismanaged Service Interface
     (Exception([exception])
       When (Execution | Call) ([flow]))+ ;
     ((Execution | Call) [flow] [From|To] [url])+ ;
     Then => ([strategy]) ;
End
```

Figure 3.7: Language constructs.

An HPL policy can then be compiled to a specific service platform. For example, if the platform is Java-based, our HPL compiler generates hardening Java components that implement interface `HardeningEventListener`:

```
public interface HardeningEventListener {
  public Object eventNotified(HardeningEvent event);
}
```

The interface is implemented by our core strategy component library (See Figure 3.6), which supplies OSGi-specific hardening components. Reliability/security experts also is able to extend the library with new hardening components that handle newly discovered vulnerabilities. The method `eventNotified` takes `HardeningEvent` which contains invocation information, including a service object, method information, URL, vulnerability type, exception, etc.

Hardening Services with HPL

Service Configurations

Figure 3.8 shows an HPL policy that can configure different operational environments. In particular, the programmer can specify device types (e.g., mobile, server, etc), network types (e.g., WiFi, 3G, LAN, etc), network conditions (e.g., delay, loss, and jitter), service types (e.g., conversational, streaming, interactive, background), and a required QoS-level (e.g., best-effort, guaranteed, etc). Through configuration settings, the programmer can detail characteristics of the remote service, thereby making it possible to provide different hardening scenarios according to dynamically changing environment.

```
Policy ServiceConfig
Begin
  PolicyName => [policy_name] ;
  ServiceName => [service_name] ;
  Config =>
    DeviceType is [mobile | server | ... ]
    NetworkType is [WiFi | 3G | LAN | ... ]
    NetworkCondition.{delay,loss,jitter}
      is {([high|med|low])+}
    ServiceType is
      [conversational|streaming|interactive|background]
    QoS is [best-effort | guaranteed | ... ] ;
End
```

Figure 3.8: A script describing service configurations.

Network Volatility

Figure 3.9 presents an HPL hardening policy that can make a service resilient network volatility. The policy is identified by its name and the `NetworkHardening` type. The same policy can be applied to multiple services by using different service identifiers. The policy contains vulnerability conditions and a hardening strategy description. The `NetworkEvent` keyword describes system network events such as disconnection, reconnection, packets loss, and normal operation. The optional `When` keyword monitors all the exceptions or events related to a specific method. The programmer specifies the conditions using the `Execution` and `Call` keywords. These keywords specify the execution locations to be monitored. The HPL compiler generates aspects to intercept application-level exceptions and system events, raised in response to experiencing volatility.

An HPL policy can be configured for different operational environments by specifying the distributed service-oriented application's device types, network links (i.e., bandwidth/latency), and required QoS. A repository of readily-available hardening components for network volatility will be reusable out-of-the-box and will also serve as building blocks for custom strategies. For network volatility, the hardening components will be based on widely used disconnected operations.

```
Policy NetworkHardening
Begin
  PolicyName => [policy_name];
  ServiceName => [service_name];
  Condition => NetworkEvent([event])
    When (Execution | Call) [From | To] [flow])+ ;
    Config =>
      DeviceType is [mobile | server | ... ]
      NetworkType is [WiFi | 3G | LAN | ... ]
      NetworkCondition.{delay, loss, jitter} is
        {([high | med | low])+}
      ServiceType is
        [conversational | streaming | interactive | background]
      QoS is [best-effort | guaranteed | ... ] ;
    Then => Apply Strategy([strategy_name]);
End
```

Figure 3.9: Hardening a service against network volatility.

Security Exploits

Figure 3.10 depicts an HPL policy for hardening a distributed service-oriented application against security exploits. In particular, we aim at *application level* security exploits of distributed service-oriented applications. Defenses against low-level attacks, such as sniffing, spoofing, etc., have been thoroughly integrated with modern network stacks. To detect software defects, we adopt the notion of a legitimate program control flow—allowable sequences of service method calls—expressed through the `Execution`, `NotExecution`, `Call`, `NotCall` keywords. These keywords parameterize our HPL compiler to generate runtime monitors that can detect and counteract exploits.

To defend a distributed service-oriented application against malicious clients, HPL features the `Access`, `Call`, and `Execution` keywords. By controlling the control flow of a service, our approach prevents malicious clients from exploiting the openness espoused by SaaS architectures. After a service's public interface is published, traditional service platforms exercise little control over how clients use this interface. Our approach adds auditing capabilities to the execution of a service by enforcing its control flow and access control.

To guard the execution of a service against improper service method parameters, HPL features the `ParamChecking` keyword that can be used to generate parameter inspection components. Parameters can be verified to hold certain values or not to surpass certain allocated memory thresholds.

In terms of the specific hardening strategies, suspicious clients can be handled by expressing in HPL a custom written component that will be invoked to counteract the detected exploits. For example, the client's connection can be terminated, a security enhancer strategy can be installed to prevent future exploits, or a service can be registered to be resuscitated if the detected penetration does end up bringing it down.

Security enhancers strategies encapsulate well-known security mechanisms such as security protocols, cryptography, authentication, and authorization schemes. Because these schemes incur a performance cost, one could choose to activate them only if necessary. For example, if unauthorized use of a service is detected, the client's connection will be terminated and an access control strategy can be deployed to control which clients can use the service in the future.

Finally, service resuscitators attempt to return a service to a clean state before or after encountering a fault [138]. Among the strategies that can be useful are *micro-restart* [16] and *checkpoint-restart* [73]. Upon detecting a potentially illegal parameter in the example above, a restart strategy can be installed to restart the service if the illegal parameter does bring down the service.

```
Policy SecurityHardening
Begin
  PolicyName => [policy_name];
  ServiceName => [service_name];
  Condition =>
    ((Execution | NotExecution ) ([flow]) From [url])+;
    Then => Apply Strategy([strategy_name]);
  Condition =>
    ((Call | NotCall) ([flow]) To [url])+;
    Then => Apply Strategy([strategy_name]);
  Condition =>
    ParamChecking([flow]) Using Strategy([strategy_name]);
    Then => Apply Strategy([strategy_name]);
  Condition =>
    Access (From | To) [url]+ ;
    Then => Apply Strategy([strategy_name]);
End
```

Figure 3.10: Hardening a service against security vulnerabilities.

Service Mismanagement

Figure 3.11 shows an HPL policy to harden a distributed service-oriented application against being mismanaged during upgrades. The `Exception` keyword adds monitoring capabilities to invoking a service through an obsolete public interface. In response to detecting such version mismatch, a hardening strategy can automatically generate a service adapter, initiate a dynamic upgrade, or schedule an upgrade at a later point. The `Execution` or `Call` keywords provide fine-grained capabilities in monitoring for service mismanagement (e.g., at the method or client location levels).

The problem of mismanaged service interface in distributed service-oriented applications is well-known [12]. Our approach explores how this vulnerability can be handled systematically. Handling this problem is closely related to managing API evolution, a highly-active area of recent research. Recent approaches include explicit documentation, automatic inference and refactorings, compatibility layers, etc. These approaches provides valuable insights for the design of hardening strategies to handle mismanaged service interfaces.

```
Policy ServiceAPIHardening
Begin
  PolicyName => [policy_name];
  ServiceName => [service_name];
  Condition =>
    Exception([exception]
      When (Execution|Call) ([flow] (From|To) [url]))+;
    Then => Apply Strategy([strategy_name);
  Condition =>
    ((Execution | Call) [flow] (From | To) [url])+ ;
    Then => Apply Strategy([strategy_name);
End
```

Figure 3.11: Hardening a service against mismanaged service interfaces.

Hardening Strategy

Fig 3.12 shows an HPL script that expresses a hardening strategy. To that end, HPL features several keywords that define basic execution directives—`Execute`, `Reject`, `Throw`, `Stop`, `Delegate`, `Replace`, etc. The directives constitute atomic operational units and are expected to be provided as part of the core component library. Using the directives, the programmer can implement service-specific strategies or extend the existing hardening strategies. A strategy script starts with the `HardeningStrategy` keyword, followed by a policy name and service identifier. Then, the `Implements` block describes strategy implementations that consist of the predefined execution directives and custom components that have to be custom implemented for the service platform in place. Our HPL compiler translates HPL scripts to components and distributed aspects that is integrated with distributed service-oriented applications.

Dynamically Composable Hardening Framework

In the following section, we discuss the system architecture of the hardening framework. The key objective of this work is to explore how policies can be interpreted and instantiated in the hardening framework and applied to an existing distributed service-oriented application that may have been written without fault-tolerance capabilities in mind.

```
Policy HardeningStrategy
Begin
 PolicyName => [policy_name];
 ServiceName => [service_name];

 Implements  {
  Method public Object eventNotified(HardeningEvent event)
  {
   @Execute(event);
   @Reject(event);
   @Throw(Exception());
   ... //Implement custom hardening strategies
  } ;
 }
End
```

Figure 3.12: Describing a hardening strategy.

The purpose of the hardening framework is to harden a distributed service-oriented application with resiliency to cope with vulnerabilities.

In designing the hardening framework, we pursue the following goals:

1. *Transparency*—any hardening strategy and the hardening framework should not affect the core functionality of the underlying OSGi framework and applications.

2. *Flexibility*—the hardening framework should be capable of adding or removing policies at any time without having to stop the application.

3. *Efficiency*—the hardening framework should not affect significantly the performance of the OSGi application .

Next, we discuss the system architecture of the hardening framework. Modern state-of-the-art middleware infrastructures reports various low-level symptoms of something going wrong in the execution of remote services (e.g., link failure, node mobility, non-existing service methods, etc.) by means of application-level exceptions. The hardening framework intercepts such application-level exceptions as well as the events signaling some changes in low-level service execution (e.g., a successful network reconnection). Then, the hardening framework handles application-level exceptions by triggering a hardening strategy.

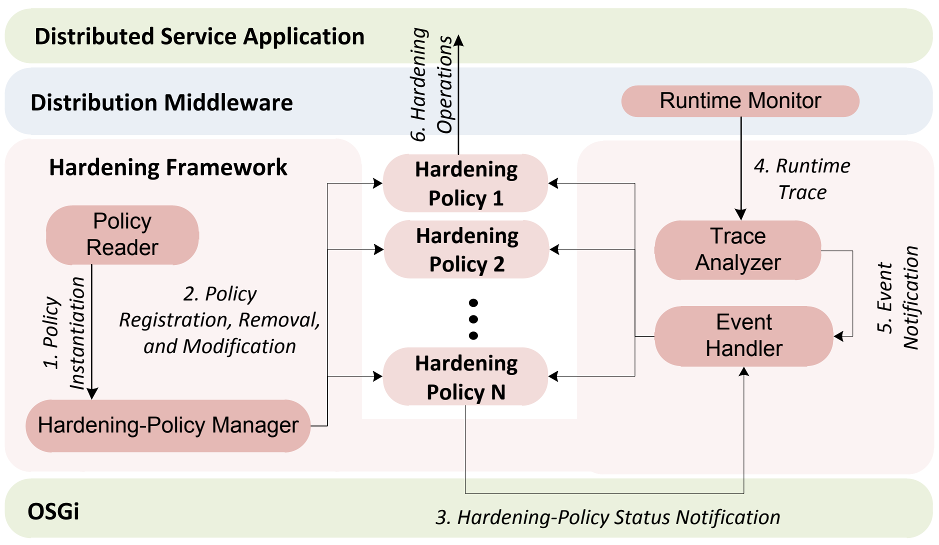


Figure 3.13: The hardening framework.

Figure 3.13 shows how the hardening framework was integrated with the OSGi framework and existing services. The hardening framework periodically reads hardening policies from the specified policy repository. Then, our HPL compiler translates hardening policies and strategy descriptions to XML documents and runtime binaries (e.g., Java bytecode). The hardening policy manager instantiates vulnerability conditions, so that the hardening framework can detect vulnerabilities by comparing vulnerability conditions with runtime traces. A hardening strategy is a standard OSGi service that implements `HardeningEventListener` interface. Then, the dynamically generated or pre-deployed hardening strategies are registered to the OSGi framework, and the hardening policy manager keeps track of their statuses (e.g., registration, unregistration, update, etc) for dynamic loading and unloading.

In addition, according to the standard OSGi specification, `ServiceHook` enables other services to intercept OSGi framework events. Thus, when a distributed service-oriented application starts its remote service, the hardening framework creates a runtime monitor which intercepts remote service invocations and catches exceptions and events. Such traces are analyzed by the trace analyzer and forwarded to the registered hardening strategies to counteract the found vulnerabilities.

3.2.3 Evaluation

We evaluated the effectiveness and performance of our hardening framework through a micro benchmark and a larger case study.

Micro Benchmark

For this experiment, we have used Lucene, a widely-used Java search engine library distributed as an OSGi bundle. Among the capabilities provided by Lucene are indexing files and retrieving indexes of a given search word. We used Lucene to implement a dictionary service that given a word can return its definition, synonyms and neighboring words.

All the experiments were conducted on the client machine running 3.0 GHz Intel Dual-Core CPU, 2 GB RAM, Windows XP, JVM 1.6.0 13 (build 1.6.0 13-b03), and the server machine running 1.8 GHz Intel Dual-Core CPU, 2.5 GB RAM, Windows 7, JVM 1.6.0 16 (build 1.6.0 16-b01, connected via a local area network (LAN) with a 100Mbps bandwidth, and 1ms latency.

In this benchmark, we measured the performance overhead for CXF-DOSGi middleware platform. Specifically, we examined how the service can be effectively executed, in terms of the total execution time when our declarative service hardening module is introduced. Each benchmark method calls three services in sequence, repeating each service call 100 times, and then reporting the total execution time. The results show that as the number of policies grows, 30 hardening policies experience a performance overhead of about 10%, which shows that our approach is practical. If a service-oriented application can afford to run 10% slower, it can benefit from our approach.

Each benchmark method calls three services in sequence, repeating each service call 100 times, and then reporting the total execution time. Figure 3.14 shows the averaged performance for each service according to the number of policies being introduced. As the number of policies grows,

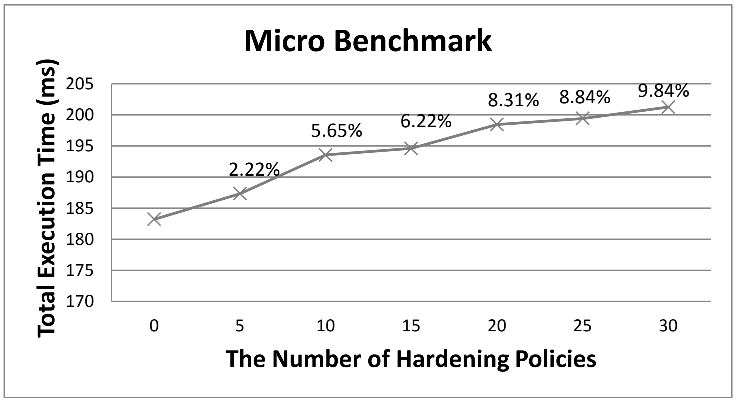


Figure 3.14: Total execution time.

we could observe about 10 percent performance overhead for 30 hardening policies. Such a low overhead shows that our approach is practical and can be applied to the majority of distributed service-oriented applications.

Case Study: OneBusAway

OneBusAway[6] is a bus information system that enables passengers of the local transportation system to track the location and movement of commuter buses over the Internet and using mobile devices [42]. OneBusAway system provides several APIs for different devices, including REST APIs for web applications, iPhone APIs, and SMS APIs.

Describing OneBusAway Configurations

Figure 3.15 shows how a OneBusAway client service can be configured. The service configurations are used for both the server and clients to determine an appropriate hardening strategy. In this case study, since the OneBusAway service aims at providing bus schedule in real time at any location, we assume that a client is a mobile device using a 3G network. Thus, network conditions such as delay, loss, and jitters are relatively high. The service type is `interactive` and the required

[6]`http://www.onebusaway.org/`

```
Policy ServiceConfig
Begin
  PolicyName => onebusaway_configs;
  ServiceName => OneBusAway;
  Config =>
    DeviceType is mobile &&
    NetworkCondition.{dealy, loss, jitter} is {high, high, high} &&
    NetworkType is 3G &&
    ServiceType is interactive &&
    QoS is best-effort ;
End
```

Figure 3.15: An HPL policy describing OneBusAway configurations.

QoS level is `best-effort`. Of course, the service configuration can be differently set according to changes of network conditions or types of a client device.

Hardening OneBusAway Against Network Volatility

```
Policy NetworkHardening
Begin
  PolicyName => onebusaway_net_hardening;
  ServiceName => OneBusAway;
  Condition =>
    NetworkEvent(Disconnection && Normal)
      When Execution(List<StopBean> StopsForLocation(*));
      Config =>
        QoS is best-effort &&
        DeviceType is mobile &&
        NetworkType is 3G &&
        ServiceType is interactive &&
        NetworkCondition.{delay, loss, jitter}
          is {high, high, high} ;
    Then => Apply Strategy(Caching);
End
```

Figure 3.16: An HPL policy against network volatility.

Figure 3.16 depicts an HPL policy to harden the OneBusAway service against network volatility. We harden the method `List<StopBean> StopsForLocation(*)`, which immediately returns bus stops' information for given location. When network events are raised from a distribution middleware system, the `Caching` strategy will be applied. The caching strategy stores all remote method invocation requests and results when the network is operating normally. Then, the strategy retrieves results from the cache. Thus, `NetworkEvent` takes two types of events—`Disconnection` and `Normal`.

Hardening OneBusAway Against Security Vulnerabilities

Figure 3.17 shows an HPL policy to harden the OneBusAway service against security exploits. This policy script describes four types of security vulnerabilities. First, to hide the remote method `CurrentTime()`, the remote service rejects all requests. Typically, removing a method from public service interface requires changing the interface's source code. Our hardening policy, however, makes it possible to hide service methods as needed. This is accomplished by declining all the client calls to the removed methods.

```
Policy SecurityHardening
Begin
  PolicyName => onebusaway_sec_hardening;
  ServiceName => OneBusAway;
  Condition =>
    Execution(TimeBean CurrentTime()) ;
    Then => Apply Strategy(Reject);
  Condition =>
    ParamChecking(List<StopBean> StopsForLocation(*))
      Using Strategy(Checker);
    Then => Apply Strategy(Reject);
  Condition =>
    NotExecution(void authenticate(*)) && Execution(*);
    Then => Apply Strategy(Reject);
  Condition =>
    Access From [malicious_url];
    Then => Apply Strategy(Reject);
End
```

Figure 3.17: An HPL policy against security exploits.

Second vulnerability is passing improper parameters to the method `List<StopBean> StopsForLocation(*)`. Since this method does not validate location data, it throws `NullPointerException` in case of that location data (i.e., longitude and latitude) are out of range. To inspect parameters, we use the `Checker` strategy.

The third vulnerability is a logic flow that can allow malicious clients to bypass authentication. For this experiment, we created a new method `void authenticate(*)` for checking clients' credentials. Thus, before calling any method in OneBusAway, clients should first set their user name that is subsequently used for authenticating all service method invocations from that client. If the method `void authenticate(*)` is not invoked, all requests are denied. The last vulnerability suspicious clients potentially misusing a service. To counter this vulnerability, the `Access` keyword monitors connected clients and can reject all requests from any specified URL.

Hardening OneBusAway Against Mismanaged Service

```
Policy ServiceAPIHardening
Begin
  PolicyName => onebusaway_API_hardening;
  ServiceName => OneBusAway;
  Condition =>
    Exception(NoSuchMethodException)
      When Execution(List<StopBean> StopsForLocation(*)) ;
    Then => Apply Strategy(Adapter);
End
```

Figure 3.18: An HPL policy against the mismanaged service.

Figure 3.18 shows an HPL policy to harden the OneBusAway service against mismanaged service. For this experiment, we added an integer argument for logging client's ID to the method `List<StopBean> StopsForLocation()`. Thus, when clients request `List<StopBean> StopsFor Location()` without specifying their ID, the remote service will throw `NoSuchMethodException`. The `Adapter` strategy supplies the missed parameter and then invokes the updated method.

Describing a Hardening Strategy

Figure 3.19 presents an HPL policy that creates a `Caching` strategy to be used for network volatility hardening. In this example, we simply store execution results in a `HashTable`. This caching strategy stores all results during normal operations and then retrieves their results when the network becomes unavailable.

```
Policy HardeningStrategy
Begin
 PolicyName => onebusaway_caching_strategy;

 Implements {
  Method public Object eventNotified(HardeningEvent event) {
    if(@Caching == null) {
      @CreateStorage
        (@Caching, HashTable<HardeningEvent, Object>);
    }
    if(event.TYPE == NETWORK_NORMAL) {
      Object result = @Execute(event);
      @Store(@Caching, event, result);
    } else if(event.TYPE == NETWORK_DISCONNECTION) {
      Object result = @Retrieve(event);
      if(result != null) { return result; }
      else { @Throw(Exception("Network Disconnection"));}
    }
  } ;
 }
End
```

Figure 3.19: Describing a caching hardening strategy.

3.2.4 Discussion

The approach described has specific engineering objectives, creating pragmatic new technologies that can make distributed service-oriented applications more available, reliable, and secure. One important question concerns whether availability, reliability, and security can be effectively reasoned about and implemented as orthogonal cross-cutting concerns, separate from the core functionality of a given distributed service-oriented application. The scientific consensus has been that it is impossible to achieve this objective in full generality. However, these concerns can be quite effectively separated in certain domains and execution environments.

Being specifically tailored to address the problems of a given domain, DSLs can be powerful and effective tools. However, learning a new DSL takes an additional effort that may negatively affect programmer productivity. Although we designed HPL to be easy to learn and use, programmers tend to differ in their ability to learn new languages. As a result, introducing HPL in the programmer's tool chain may initially inconvenience some programmers.

Finally, to yield its intended benefits, our approach relies on the existence of state-of-the art adaptation facilities of the underlying middelware infrastructure. Although OSGi has all the facilities required to support our approach, other middleware platforms may lack some advanced features such as deploying and undeploying services at runtime. In future work, we plan to explore how generalizable our approach is.

3.2.5 Related Work

Although modern society intrinsically depends on software systems, all computing systems are prone to unreliability. Complex distributed systems often fail to deliver the expected quality of service (QoS), when their constituent components fail. This lack of reliability negatively affect the overall system's trustworthiness. Indeed, defects in deployed software systems cost the US economy billions of dollars annually [128]

Our approach is related to several research domains, which include automated fault tolerance, security hardening, adaptive and fault-tolerant middleware, and aspect oriented software construction. This work synthesizes and enhances some existing common hardening strategies. In the following discussion, we outline the main research domains from which this work draws inspiration and borrows well-established and verified solutions.

DSL for Reliability and Security

Much research explored DSLs to solve reliability problems. In the field of security, policy-based approach has been widely explored in the last decade. Among recently introduced policy languages are including Ponder [28] and Rei [67]. Ponder defines authorization and security management policies. Because policies are separated from a system, it can adapt to changing requirements by disabling or replacing policies without restarting. Rei can be used to define different kinds of policies, including security, privacy, management, and conversation. These policy languages have inspired the design of HPL. However, HPL focuses on application-level security and also aims at availability and reliability.

GRAFT [139] automatically specializes middleware for fault-tolerance. It employs Component Availability Modeling Language (CAML) to annotate a distributed application's model, and then automatically specializes the application's middleware for domain-specific fault-tolerant requirements. GRAFT also uses a DSL to express the requested fault-tolerance functionality. Although similar to our approach in terms of adopting domain-specific approach, GRAFT only copes with reliability problems. On the other hand, our approach counteracts availability and security, as well as reliability.

Business Process Execution Language (BPEL) is a standard language that defines business processes for Web services. A BPEL program can, for example, express that a Web service be composed through a business process involving some existing Web services. To handle failures in BPEL processes, various monitoring techniques have been proposed [53, 10, 101, 11]. Our approach shares the same goal with these techniques, but we strive to achieve greater transparency in detecting anomalies and flexibility in deploying solution components. Unlike the prior state of the art, our approach does not require any modification to the underlying middleware infrastructure (e.g., Web service runtime or the BPEL execution engine). Our approach also deploys special-purpose components to counter the detected vulnerabilities. Furthermore, our approach is flexible and dynamic: special failure-handling components can be deployed at runtime without having to interrupt the execution.

Some of recent research has focused on providing failure handling mechanisms at runtime by using the Aspect-Oriented Programming (AOP) technique, which enables inserting failure handling modules into an unmodified BPEL. However, whenever weaving occurs at deployment time [10], new failure types cannot be handled dynamically. Although reference [101, 11] presents a runtime failure handling mechanism, it can only handle restricted failure types (e.g., service failure) because they were built on top of the existing BPEL specification. As compared to these approaches, our framework includes both a dynamically composable failure handling language and its execution runtime system. Our approach thus equips programmers with the ability to cope with various service QoS vulnerabilities by simply describing a new policy script and dynamically instantiating the required hardening strategies. Finally, most BPEL-based approaches have focused on handling failures at a service provider. However, our approach enables failure handling at both the server and client parts of an service-based application. Thus, whenever a service provider cannot be modified, the service can still be hardened by deploying our framework only at the service consumer side.

Fault-Tolerant Middleware

A number of techniques for making existing systems fault tolerant [58, 115, 82] are related to our approach. JReplica [58] expresses via AOP how adaptable fault tolerance can be added through replication. Reference [115] describes how fault tolerance can be added to CORBA components by automatically instantiating distributed replicated components. DR-OSGi [82] is a component framework to harden distributed service-oriented applications against network volatility. DR-OSGi avoids modifying source code explicitly and enables the reuse of disconnected operations across different applications. Arora and Kulkarni [7] have shown that fault-tolerant systems feature two types of components that they called *detectors* and *correctors*. They have argued that enhancing a fault-tolerant system with a set of fault-tolerant components will lead to a fault-tolerant system. They have also suggested that this division can serve as a basis for designing component-based fault tolerant systems.

Our approach based on above techniques enables the programmer to harden distributed service-oriented applications without having to modify their source code explicitly. By avoiding ad-hoc modification that can be tedious and error-prone, our approach not only hardens distributed service-oriented applications more systematically, but also enables greater reuse of the hardening strategies across different distributed service-oriented applications.

Security as a Separate Concern

Our approach treats security as a separate concern. A popular technology for modularizing cross-cutting concerns is AOP [71], which has been successfully used in prior systems for introducing security-related functionality [159]. In addition, several special security libraries and frameworks are AOP-based, including Java Security Aspect Library (JSAL) [21], Security Annotation Framework [124], and Spring Security [135]. What makes AOP a promising technology for implementing our approach is its ability to weave in concerns at runtime, without restarting the application. This runtime adaption ability aligns well with the dynamic nature of the OSGi infrastructure. AOP is not the only approach for encapsulating security functionality. A middleware-based approach such as CORBA Security Service [105] has been shown successful for modularizing security functionality, including authentication, authorization, confidentiality, integrity, and auditing.

3.2.6 Conclusions

In this section, we have introduced *Declaative Hardening*, a promising approach for systematically hardening service applications to cope with network volatility, security exploits, and service mismanagement. Our HPL language is an expressive a powerful abstraction for the programmer to describe various hardening policies. The HPL compiler translates policy scripts to hardening components, which are applied to distributed service-oriented applications at runtime. The micro benchmark and case study showed effectiveness of our approach. As we rely on greater numbers of network-enabled devices with network volatility, security exploits, and service mismanagement remain a permanent presence. Declarative hardening explores how these vulnerabilities can be handled declaratively, providing a systematic and reusable solution.

Chapter 4

Enabling Cloud-Based Execution via Cloud Refactoring

One of the foundations of cloud computing is Software-as-a-Service (SaaS), a computing paradigm in which clients access a piece of remote, cloud-hosted functionality through a public interface. To take advantage of cloud-based services, centralized software applications must be re-engineered, so that a portion of their functionality is hosted in the cloud. One may want to replace an existing application functionality with an equivalent cloud-based service for a variety of reasons, both business and technical. Replacing a locally implemented functionality with a remotely maintained service reduces the maintenance burden. A service provider can more effectively improve quality and reduce costs by leveraging the economies of scale, when the same service is used by multiple clients. A service may have access to computing resources that are superior to the resources of a local machine. Services often conglomerate other services, thus offering additional benefits. For example, if a service persists user data, it is likely to offer backup and restoration facilities not common outside of large server infrastructures.

Using cloud-based services has become a common avenue for leveraging remote computing resources, with the benefits that include reduced costs, increased automation, greater flexibility, and enhanced mobility [15]. Despite all the benefits of leveraging cloud-based resources, transitioning a centralized application to effectively use remote services requires extensive changes to the application's source code. In addition, the possibility of partial failure requires that proper fault handling functionality be added to any application that invokes services remotely. As a result, programmers manually transition applications to use cloud-based services, changing code in difficult, costly, and error-prone ways. Therefore, there is great potential benefit in automating these changes and making the automation available to the software engineering community.

A popular technique for automating common program transformations is called *a refactoring*. Defined generally, a refactoring is a semantics preserving program transformation performed under programmer control [45]. In other words, refactoring is automated: a programmer determines if a refactoring should be performed and then engages a refactoring engine that transforms the code automatically. In this chapter, we advocate the vision of using refactoring as a means of facilitating the transitioning to cloud-based services. We argue that transitioning an application to take advantage of cloud-based services preserves the application's semantics in the sense that the overall functionality does not change. Executing some functionality in the cloud does not change the semantics from the end user's perspective.

We present a set of refactoring techniques that facilitate the process of transforming centralized applications to use cloud-based services. These techniques automate the program transformations required to (1) render portions of functionality of a centralized applications as cloud-based services and re-target the application to access the services remotely; (2) handle failures that can be raised in the process of invoking a cloud-based service; and (3) switch a service client to use an alternate, equivalent cloud-based service. These refactoring techniques—collectively named *Cloud Refactoring*—have been concretely implemented in the context of the Eclipse IDE and added to its refactoring engine.

To validate *Cloud Refactoring*, we applied its constituent refactoring techniques to transform two centralized, monolithic Java applications to use cloud-based services. We also applied *Cloud Refactoring* to re-engineer a commercial application used by General Electric (GE) to use cloud-based services in an effort to demonstrate how refactoring can help realize the GE strategic vision to take advantage of cloud computing.

Our results indicate that *Cloud Refactoring* can be a powerful tool for the programmer charged with the task of transitioning a centralized application to one that uses cloud-based services. Not only can *Cloud Refactoring* transform code with a high degree of automation, but it can also properly account for the demands of distributed execution. Thus, *Cloud Refactoring* represents a robust and pragmatic approach that can reduce maintenance costs and increase programmer productivity.

The rest of this chapter is structured as follows. Section 4.1 motivates our approach and then introduces the main technologies discussed in this chapter. Section 4.2 describes the new refactoring techniques. Section 4.3 reports our experiences of applying *Cloud Refactoring* to third-party applications. Section 4.5 compares our approach to the existing state of the art. Section 4.4 discusses advantages and limitations of this work. Finally, Section 4.6 presents concluding remarks.

4.1 Motivation and Technical Background

In the following discussion, we introduce an example that motivates this research and then provide a technical background our approach uses.

4.1.1 Motivating Example

Consider JNotes [1], a third-party diary and project management application written to run on a single desktop machine. Our goal is to refactor JNotes into a cloud-based application, with the server part deployed on a remote server and the client part accessed from a mobile device. This transformation offers several advantages. All the JNotes documents and calendars can be saved at the remote server, whose file system can be regularly backedup and replicated, so that data consistency will not suffer from a failure of the client's file system. Furthermore, through a simple change in server deployment, JNotes can be made into a collaborative application, with multiple clients sharing the same server components.

A major technical impediment to realizing the transitioning outlined above is that maintenance programmers have to change the application's source code by hand. JNotes is a typical centralized application that comprises a collection of Java classes. Splitting JNotes into the service and client parts, deploying the services in the cloud, and having the parts communicate with each other

[1] `http://memoranda.sourceforge.net`

reliably can quickly turn into a complex programming undertaking. Furthermore, the resulting cloud-based application is likely to contain software imperfections, commonly introduced when manipulating code by hand.

Although software frameworks have been introduced to ease rendering classes as Web services, the classes must adhere to a rigid set of architectural conventions. Thus, it is unlikely that these frameworks can help transition arbitrary classes to Web services. As an example, consider JAX-WS [63], a framework that represents a significant industry effort to simplify the development and deployment of Web services. With JAX-WS, a programmer can export a standard Java class as a Web service by annotating the class with `@WebService` and the the class's methods with `@WebMethod`. A code generation tool that comes with JAX-WS reads these annotations and creates all the required supporting harness to exposes and deploy the annotated methods as XML-based Web services.

However, this service extraction model simply renders existing methods as Web services without any consideration for the resulting performance and reliability. To ensure good performance and high reliability, the classes that are to become Web services may need to be restructured first. For example, a service may need to use only a subset of the class's fields, thus requiring splitting the class into client and server partitions.

Consider moving class `FileStorage` to the cloud as a means of saving all the JNotes documents in a shared cloud storage. With JAX-WS, the programmer can transform the entire class with all its methods into a Web service. Unfortunately, this "all or nothing" inflexible distribution model may fall short of meeting the needs of realistic applications. For example, some functionality in `FileStorage` pertains to local file paths, and as such should not be moved to the cloud. That is, the functionality tied to the client environment cannot be moved. More specifically, the programmer needs to split methods `storeResourcesList(...)` and `openResourcesList(..)` from the rest of the class before it is transformed into a remote service.

The refactoring approach that we advocate here enables the required level of flexibility when transforming classes into remote services. Our *Extract Service* refactoring takes as input a class name and a set of methods that are to be rendered as a remote service. This refactoring then transforms the given methods into remote service methods, leaves the remaining methods on the client, rewrites all communication between the original and remote methods into remote service calls.

Because centralized and distributed applications have different failure modes, simply rendering a subset of a centralized application remote does not preserve the semantics. Distributed applications are subject to partial failure, in which its different components (client, server, or network) may fail independently from each other. Although one cannot handle all the possible failures in a distributed application, some failures have well-known handling strategies. Thus, to better preserve the original execution semantics, the *Extract Service* refactoring also adds client-side fault tolerance functionality as specified by the programmer. For example, the programmer may specify that an unsuccessful attempt to reach a service be repeated a given number of times. In our approach, the programmer can specify and configure the fault tolerance strategies to apply by means of an XML-based domain-specific language.

Finally, a service application may need to use more than one service implementation for a given functionality. For example, in the case when one service implementation is not available, the client should switch to using a different service, whose method interface is different from that used by the original service implementation. Thus, the client code will need to be adapted to use different service interfaces. The *Adapt Service Interface* refactoring automates the transformations required to be able to switch the client to using an equivalent service exposed through an incompatible service interface.

4.1.2 Technical Background

Next, we provide an overview of the cloud and service computing technologies.

Cloud Computing and Services

Cloud computing provisions resources on-demand through three main virtualization approaches: (1) infrastructure virtualization—provisioning computing power, storage, and machine (e.g., Amazon EC2); (2) platform virtualization—provisioning operating systems, application servers, and databases (e.g., Amazon S3); (3) software virtualization—provisioning complete Web-based applications (e.g., `Salesforce.com`). The main benefits of cloud computing include elasticity, scalability, and availability. From the software development perspective, however, taking advantage of cloud computing requires that the programmer follow a strict set of architectural and design guidelines.

Service Oriented Architecture (SOA) provides uniform access to a variety of computing resources across multiple application domains. Loosely coupled services may be co-located in the same address space or be geographically dispersed across the network. Among the software engineering advantages of services are strong encapsulation, loose coupling, ease of reusability, and standardized discovery. In addition, due to the strong separation between service interfaces and implementations, service developers have the flexibility to mix any middleware platforms and applications as well as to switch service infrastructures without affecting service clients. It is these desirable software engineering properties that made SOA a widely used paradigm for realizing cloud computing solutions.

4.1.3 OSGi Framework as a Cloud Computing Platform

Open Service Gateway Initiative (OSGi) [107]—a service implementation and provisioning infrastructure—has been embraced by numerous industry and research stakeholders, organized into the OSGi Alliance [2]. As a service platform, OSGi can render any Java class as a service bundle. OSGi manages published bundles, allowing them to use each other's services through public interfaces. OSGi manages the lifecycle of a bundle (i.e., moving between install, start, stop, update, and delete

[2]Open-source OSGi implementations include Apache Felix [144] and Knopflerfish [74]. Among large commercial OSGi projects are Spring Framework [134] and Eclipse Equinox [35].

stages) and allows it to be added and removed at runtime.

OSGi has made substantial inroads into the domain of cloud computing. Some enterprises have adopted OSGi as the platform for realizing their private clouds [109]. In light of these developments, a consortium of major industry and academia stakeholders has issued the Request for Proposal (RFP) 133 [108], which codifies how OSGi should be leveraged as a platform for cloud computing.

4.2 Our Approach: *Cloud Refactoring*

The goal of our approach is to alleviate the code transformation hurdles involved in adapting existing applications to take advantage of cloud-based services. To reduce development efforts/costs and increase programmer productivity, we have expressed as refactorings several common program transformations that programmers perform when adapting applications to use cloud-based resources. Although our approach is not fully automatic, programmers only determine if the source code should be transformed. The actual transformations are performed by a refactoring engine. In the following discussion, we first give an overview of our approach and then detail its individual parts.

4.2.1 Approach Overview

Our approach focuses on those common program transformations occurring when using cloud-based services that are well-amenable to be expressed as a refactoring. In particular, we focus on three software re-engineering scenarios. One scenario involves moving some of a centralized application's functionality to the cloud. The second scenario involves adding fault tolerance functionality to the client to handle the faults raised during the invocation of a cloud-based service. The third scenario involves switching an application to use an alternate cloud-based service exposed through a different service interface.

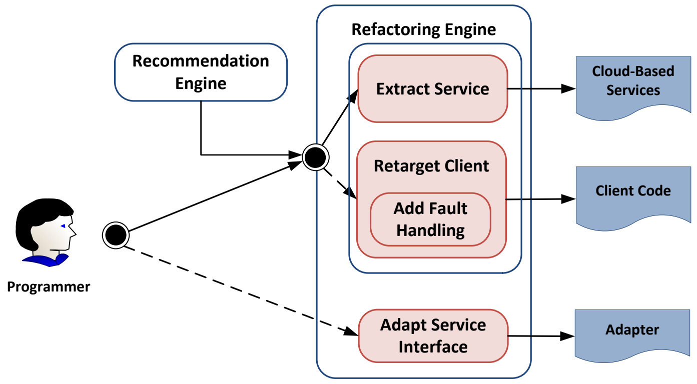


Figure 4.1: Migrating to Cloud-Based Services via *Cloud Refactoring*.

Figure 4.1 shows how the constituent components of our approach fit together. The two main parts of our approach are *The Recommendation Engine* and *The Refactoring Engine*. The recommendation engine uses static and dynamic program analysis techniques to infer class coupling; this optional component can inform the programmer about which classes can be converted into a cloud-based service. The two refactoring techniques of the refactoring engine are intended be used à la carte. By integrating the engine with the Eclipse IDE, our approach makes it possible to use the new refactoring techniques indistinguishably from the existing ones. Furthermore, some of the existing, widely used refactoring techniques can be quite useful for applications that use cloud-based services. For example, *Extract Service* refactoring can be used to move a method to a class prior to converting that class to a cloud-based service.

4.2.2 Service Recommendation

To make sure that moving functionality to the cloud does not render the application unusable due to exploding latency costs, programmers should use service components rather than individual objects as a distribution boundary. Because few existing applications consist of service components, programmers should first ensure that an intended service is not tightly coupled with the rest of the application. For example, they can apply a *Façade* pattern that exposes some tightly coupled functionality through a crude-grained interface.

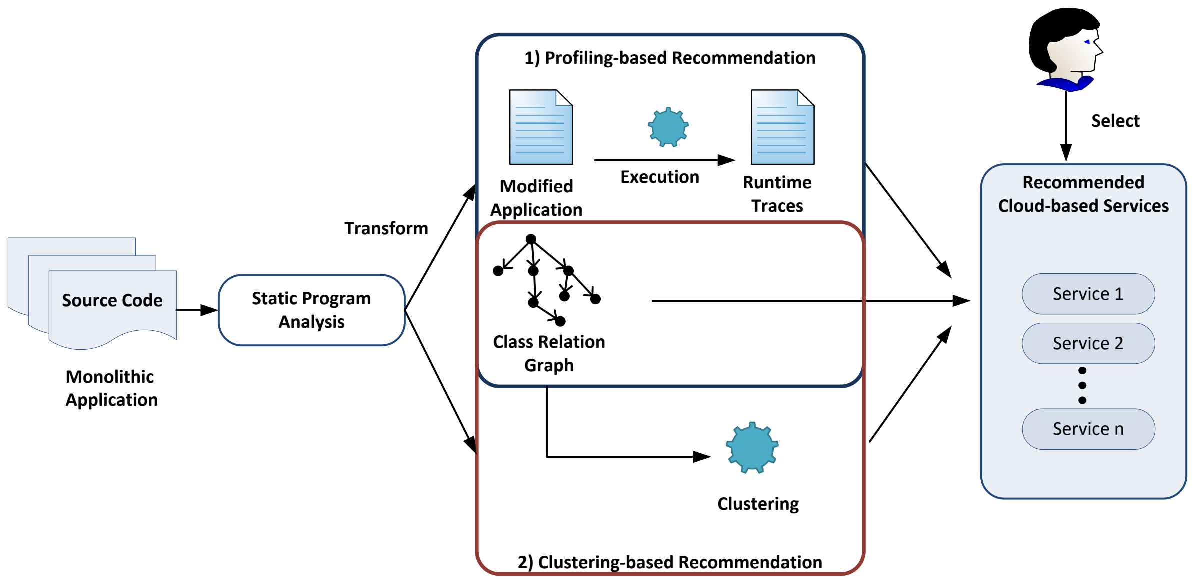


Figure 4.2: Service recommendation process.

Nevertheless, it may be difficult to determine which functionality is a good candidate to be exposed as a cloud-based service. To that end, our approach provides a recommender tool that computes the coupling metrics for all the classes in an application and then displays the classes that are least tightly coupled. Accessing the functionality represented by these classes from a remote cloud-based service should impose only a limited performance penalty on the refactored application.

Figure 4.2 shows the process diagram for identifying classes that can be converted into cloud-based services. Our approach leverages two recommendation mechanisms: profiling- and clustering-based recommenders. The profiling-based recommender engages a static program analysis and runtime monitoring to collect program information. By combining the class coupling metrics collected through both static analysis and runtime monitoring, the recommendation algorithm then suggests a subset of an application that can be transformed to cloud-based services. The profiling-based recommender sorts application classes based on their execution duration and frequencies, so that the programmer can know what classes are computation-intensive and how frequently they are accessed. The clustering-based recommender clusters classes with similar functionality, thus identifying class clusters whose functionality can be naturally exposed as a cloud-based service.

Because the clustering-based recommender groups classes based on their functionality, the programmer can avoid duplicating a functionality in the cloud by selecting candidates for cloud-based service from different clusters.

Note that these recommendation mechanisms are provided as a tool that can inform the programmer about the properties of the applications about to undergo a refactoring. The programmer is ultimately responsible for deciding which classes should be transformed into cloud-based services and even if the transformation should take place to begin with. This design choice is in line with the automated nature of our refactoring techniques. In the following discussion, we describe our two recommendation mechanisms in detail.

Profiling-based Recommendation

In essence, the recommender strives to find a distribution strategy that would not render the application unusable due to the drastically increased latencies of invoking tightly coupled methods across the network. Because the recommender only takes the coupling metrics into consideration, it cannot produce a recommendation that is guaranteed to always exhibit a superior performance. Other factors, such as business logic and system resources used, can impact the performance drastically. As a result, the programmer can only use the recommender as a tool to explore the coupling metrics of the refactored application rather than as an absolute arbiter that determines which functionality is to be extracted into remote services.

Figure 4.3 shows our service recommendation algorithm that operates on a class relation graph. Given a graph, the algorithm calculates a service utility value for each class in the program, a rank that expresses how fit a class is to be rendered as a cloud-based service. Specifically, the algorithm uses the service utility function defined as follows:

$$F(i) = \sum_{i \in edges} \{W_\alpha \times \frac{T_i}{MAX(T_0, .., T_n)} + W_\beta \times \frac{N_i}{MAX(N_0, .., N_n)}\}$$

where N, T, and W denote execution number, execution time, and weight to each measurement

INPUT: A class relation graph, CRG
OUTPUT: A set $CS = \{c_1, c_2, \ldots, c_n\}$ of possible remote classes

```
$classes \longleftarrow$ calculateUtility($CRG$);
$destinations \longleftarrow$ edgesOutOf($class$);

while ($destinations \neq \emptyset$) do
  $class \longleftarrow destinations$.next();
  $utility \longleftarrow class$.getUtility();
  $coupling \longleftarrow class$.getCoupling();
  if ($utility \geq util\_threshold$ && $coupling \leq coup\_threshold$) then
    $CS$.add($class$);
  end if
  $destinations \longleftarrow$ edgesOutOf($class$);
end while
```

Figure 4.3: Profiling-based service recommendation algorithm

metric, respectively. If W_α is larger than W_β, classes related to business logic will be suggested. Conversely, if W_β is larger than W_α, frequently accessed classes will be suggested. Then, we defined our own coupling metrics as follows:

$$CP(i,j) \quad = CC(i,j) + CR(i,j) = \frac{1}{\#ofhops} + \frac{\sum(e_i \cap e_j)}{\sum e_i + \sum e_j}$$

where CP, CC and CR denote coupling, class connectivity, and class relation values. Class connectivity, CC, denotes how the given two classes are closely connected. If class x_i has lower hops to go class x_j, they are strongly connected. If x_i and x_y are directly connected, $CC(i,j)$ is 1. Otherwise, $CC(i,j) = \frac{1}{\#ofhops}$. Class relation, CR, denotes how the given two classes are related. If class x_i creates, reads, writes, and invokes only class x_j, class x_i is tightly related to class x_j. CR is computed using the number of in/out edges from the given class to other classes. Then, the algorithm described traverses the graph from the root to the leaf nodes and then suggests cloud-based service candidates based on the calculated service utility values. Based on this suggestion, programmers can then choose classes that are suitable candidates for cloud migration.

Spectral Clustering-based Recommendation

The second recommender clusters related classes together. In recent years, spectral clustering has become one of the most widely used clustering algorithms. Spectral clustering techniques make use of the spectrum of the similarity matrix of the data to perform dimensionality reduction for clustering in fewer dimensions. Given a set of data points $x_1, \ldots, x_n$ and similarity $S(i, j)$ between all pairs of data points x_i and x_j, the similarity matrix is defined as S. If the similarity $S(i, j)$ between the corresponding data points x_i and x_j is positive, two vertexes are connected. The similarity matrix is computed as follows:

$$S(i,j) \quad = CC(i,j) + CR(i,j) + D(i,j) + L(i,j) + T(i,j)$$

where CC, CR, D, L and T denote class connectivity, class relation, class distance, library usage, and type similarity, respectively. We use the same formula to compute CC and CR, which are defined above. Class distance, $D(i, j)$ is calculated using the Levenshtein distance algorithm [85]. If x_i and x_j have the same package name, these classes are considered more similar to each other that classes in different packages. With respect to library usage, L shows how the relationship between two classes in terms of the similarity of the libraries they use. Using the same library indicates a similarity in functionality. If x_i and x_j use the same libraries, their $L(i, j)$ is 1. Otherwise, their $L(i, j)$ is 0. Finally, type similarity, T, denotes the similarity of classes in terms of their types. If the classes implement the same interfaces or inherit from the same super class, their $T(i, j)$ is 1. Otherwise, their $T(i, j)$ is 0.

Figure 4.4 shows the clustering-based service recommendation algorithm, which is parameterized with a class relation graph. First, the graph's similarity matrix is constructed. Since each node of the graph has method/class information, the similarity between each pair of classes is calculated according to their similarity metrics. Then, a recursive spectral clustering algorithm continuously partitions the similarity matrix until it reaches the base case (the partition size equals 1).

INPUT:	A class relation graph, CRG
OUTPUT:	A set $CS = \{c_1, c_2, \ldots, c_n\}$ of possible remote classes

```
$clusterNum \longleftarrow 2$; //initialize the number of clusters
while (true) do
  $SIM \longleftarrow$ constructSimilarityMatrix($CRG$);
  $cluster \longleftarrow$ buildCluster($SIM, clusterNum$);

  if($cluster = \emptyset$) then exit; end if

  while ($cluster \neq \emptyset$) do
    $class \longleftarrow cluster$.next();
    if ($class$ is accessed from other clusters) then
      $CRG$.remove($class$);
      $CS$.add($class$);
    end if
  end while

  $clusterNum \longleftarrow clusterNum + 1$; //increase the number of clusters
end while
```

Figure 4.4: Clustering-based service recommendation algorithm.

Constraints on Extracting Cloud-Based Services

Not all classes can be easily migrated into remote, cloud-based services. Various constraints make it impossible to refactor some classes for cloud-based execution. These constraints pertain to the use of local resources, parameter passing, and serialization. Classes that make use of local resources, such as databases, disk files, and sensors cannot be moved to be executed by a different host. In our refactoring approach, we assume that the programmer is aware of such local resource usage and would not try to migrate the affected classes to the cloud. Our refactoring techniques can only pass by-copy parameters, which includes primitive and read-only parameters. The classes whose methods contain other types of parameters cannot be transformed into cloud-based services; our recommenders identify and exclude such classes. Finally, OSGi requires that non-primitive remote method parameters be serializable and attempts to automatically serialize them.

Next, we describe two refactoring techniques that form the foundation of *Cloud Refactoring*: 1) Extract Service and 2) Adapt Service Interface.

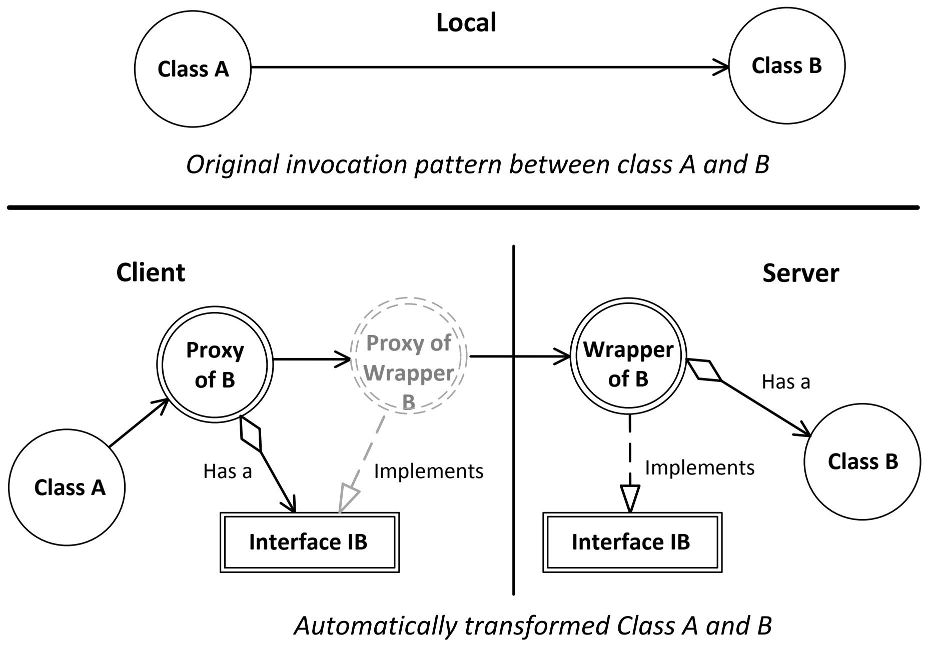


Figure 4.5: *Extract Service* refactoring—service transformation and client redirection.

Cloud Refactoring—1) *Extract Service*

Extract Service refactoring automates the program transformations required to transform regular classes into remote services. A typical *Extract Service* refactoring performs the following four program transformations: 1) rewrite a class making all its methods into remote service methods, 2) partition class methods into service methods and regular methods, rewriting all the communication between the two into remote service calls, 3) re-target all clients of the original class to access its functionality in the cloud by means of remote service calls, and 4) add fault handling functionality to client code.

Transforming Program Code to Extract Services

Figure 4.5 shows local classes `A` and `B` can be transformed by means of the *Extract Service* refactoring, so that `B` becomes a fault tolerant cloud-based service. At the server side, class `B` is exposed via a generated interface IB and class B becomes wrapped into an instance of class WrapperB. The exposed service interface IB can be invoked via standard service protocols (e.g., HTTP-SOAP, REST, etc). At the client side, class A imports the service via interface IB, with the original class B at the client being replaced with a generated proxy class. This example demonstrates the transformation of all the methods in a class into cloud-based services. If only a subset of the methods are to be transformed into services, additional transformations are necessary.

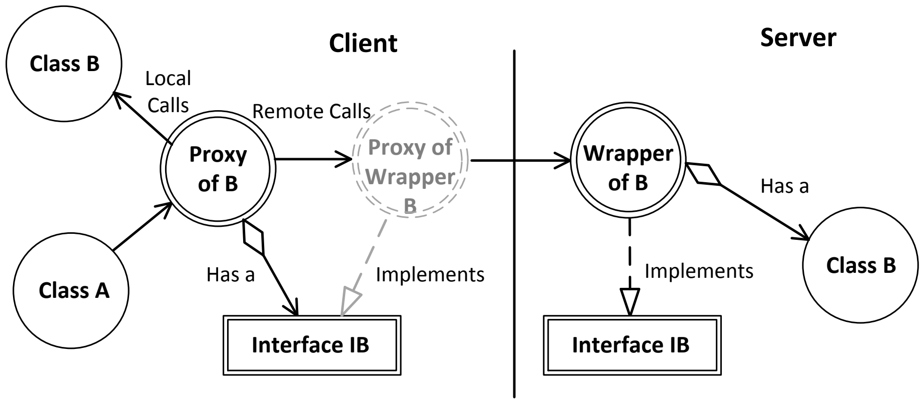


Figure 4.6: Splitting a proxy into two parts.

To split a class, the refactoring engine takes as input its name and then either a set of fields or methods to move to the cloud. If the refactoring input is specified by means of fields, the selected fields and the methods accessing them are moved to the cloud. If the refactoring input is specified in terms of methods, the selected methods and the fields accessed by them are moved to the cloud. Figure 4.6 shows how the refactoring engine splits class B to redirect all the invocations to class B to the cloud-based service interface IB and the local object LB. Figure 4.7 shows an automatically generated proxy class, which redirects all the invocations to class B to the cloud-based service interface IB and the local object LB.

```
public class B {
 private IB proxy;
 private LB local;

 public B() {
  proxy = (IB) getService(IB.class);
  local = new LB();
 }
```

```
 /* Re−targeted methods */
 public String foo(int i1, int i2) {
  return proxy.foo(i1, i2);
 }

 /* Remaining methods */
 public void bar() { local.bar() }
}
```

Figure 4.7: Generating a proxy class.

Handling Service Faults

Whether some functionality is accessed locally or remotely across the network should not change the application's functionality if not for the presence of partial failure. Unlike in a centralized application, components of cloud-based services can fail independently, making such failures difficult to diagnose and handle. Such failures must be handled effectively not only to ensure the overall application utility and safety, but to preserve the semantics of the original centralized applications. Thus, any refactoring technique that separates any functionality to be accessed remotely must take the issue of remote failures into consideration. In our approach, the *Extract Service* refactoring automatically adds well-known fault tolerance strategies configured through a domain-specific language.

Because it would be impossible to handle all possible errors, our refactoring approach focuses on well-known strategies for handling common faults, such as network volatility, service outages, and internal service errors. The generated fault tolerance functionality includes both detection and handling. A fault can be detected via timeout mechanism, exception handling, or runtime execution monitoring. Then, the detected faults should be properly handled to keep continuing the required functionality. Figure 4.8 shows these fault-handling procedures. First, a service administrator needs to provide fault tolerance descriptions written in Fault-Tolerance Description Language (FTDL) [36, 77], a domain-specific language we have developed earlier for expressing fault handling strategies. These descriptions parameterize a fault handling component that detects the specified faults and then counteracts their effect by executing the specified handling strategies.

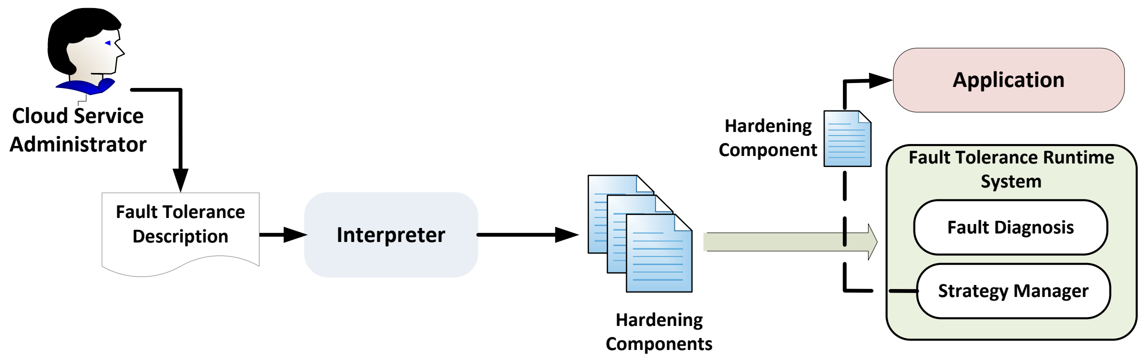


Figure 4.8: Overview of fault handling.

The refactoring engine inserts all the required fault handling code into proxy classes.

Fault Tolerance Description Language

One of the key novelties of our refactoring approach is using a domain-specific language to configure a refactoring engine to synthesize fault tolerance functionality. In our previous work [36, 77], we explored how remote services can be made resilient against failures using domain-specific languages—*Hardening Policy Language (HPL)* [77] and *Fault Tolerance Description Language (FTDL)* [36]. In this work, we combined the features of these two languages—language constructs from FTDL and a runtime system from HPL— to create a refactoring transformation that automatically adds fault tolerance functionality. Figure 4.9 shows how FTDL is used by the refactoring infrastructure. The FTDL design has striven to combine expressiveness and ease of use. The specific design goals of FTDL have included: *Expressiveness*—a programmer should be able to express any kind of fault easily, with the resulting code being easy to understand, maintain, and evolve; *Extensibility*—it should be possible to integrate existing fault tolerance strategies with FTDL strategies; *Platform Independence*—FTDL strategies should be service platform independent, with the same strategy capable to counteract a fault raised by any service implementation.

```
<ftdl>
  <service uri=[remote address] method=[method name]/>
  <condition>
    <timeout>[0-9]*</timeout>
    <exception>[exception types]</exception>
  </condition>
  <strategy>
    <!- attributions for the retry strategy -->
    <retry numRetries=[0-9]* backoffInterval=[0-9]*
       backoffType=["exponential" | "linear"] />

    <!- attributions for the sequential strategy -->
    <sequential numRetries=[0-9]* backoffInterval=[0-9]*
       backoffType=["exponential" | "linear"] >
      <service uri=[http://remote] />
        ... ...
    </sequential>

    <!- attributions for the user defined strategy -->
    <defined>
      <handling name=[fault handling name] />
      ... ...
    </defined>
  </strategy>
</ftdl>
```

Figure 4.9: FTDL constructs.

Fault Tolerance Strategies

Traditionally, fault handling functionality in services computing tends to follow well-defined patterns. To render services reliable, representative approaches replicate SOAP web services [32, 41, 87], introduce transactional processing [88], and add fault handling code to server and client sides of a service-oriented application [126, 171]. Our approach focuses on client-side fault handling for the faults caused by network volatility, service outages, and internal service errors. The first step in handling a fault is detecting it. The fault conditions can be detected via timeout mechanism, exception handling, or runtime execution monitoring. In the context of service-oriented applications, the following fault tolerance strategies are used quite commonly:

- **Retry:** The strategy that is arguably employed most widely is **Retry**, which, for a given number of times, reattempts to invoke a service in response to a failure.

```
public class B {
  ...

  /* Fault handling code */
  public String foo(int i1, int i2) {
    try {
      return proxy.foo(i1, i2);
    } catch(CloudServiceException e) {
      return FaultHandler.notify(new Fault(...));
    }
  }
}
```

```
public class MyFaultHandling implements FaultListener {
  public Object faultNotified(Object service, Method m, Object[] params) {
      // retry the failed service invocation
  }
}
```

Figure 4.10: Automatically generated fault handling code.

- **Sequential:** Another common strategy is **Sequential**, which is also known as *passive* replication. This strategy iterates through different endpoints of a service when encountering a failure. For instance, when experiencing a timeout in response to invoking the service endpoint at `a.com/foo`, `b.com/foo` can be invoked next. This strategy, thus, increases the probability that some invocation will finally succeed. The term *passive* replication refers to the fact that this strategy does not kick in until a failure occurs.

- **Parallel:** An example of a more complex strategy is **Parallel**, which *actively* replicates a service, to invoke endpoints concurrently as a mechanism to counteract potential service unavailability. For example, both endpoints `a.com/foo` and `b.com/foo` would be invoked simultaneously. As a form of speculative parallel execution, this strategy proceeds with the first successfully executed request.

- **Composite:** Because a single strategy may not be sufficient, software designers often combine multiple strategies. For example, all the heretofore described strategies can be combined into composite strategies.

Generating Fault Handling Code

Figure 4.10 shows the fault handling functionality in a generated proxy class. Specifically, this proxy handles all the raised exceptions by passing them to method `notify()` in class `FaultHandler`. The fault handler is our light-weight fault-handling runtime that can execute fault-handling strategies. The runtime can execute both the standard fault tolerance strategies as well as the combinations of thereof. The standard strategies include retry, sequential, and parallel. These strategies can be combined in arbitrary ways into composite strategies by writing a simple FTDL script. To specialize fault handling even further, one can implement any required fault-handling strategy by implementing interface `FaultListener`. The fault tolerance strategies can be reused across applications and can serve as building blocks for custom strategies.

The lightweight runtime system depicted in Figure 4.11 consists of a fault diagnosis module and a strategy manager. The fault diagnosis module catches raised exceptions or failures. The strategy manager associates exceptions with fault tolerance strategies. In response to detecting an exception, the manager initiates the handling strategy as configured by a given FTDL script. A strategy implementation is simply a sequence of corrective actions whose execution counteracts the effect of experiencing the fault. These actions are implemented as part of a library. In our prior work, we have demonstrated the effectiveness of this approach to improve the reliability of OSGi-based systems [77, 82].

4.2.3 Cloud Refactoring—2) *Adapt Service Interface*

Cloud-based services expose their functionality through a set of public interfaces. It is also common that the same business functionality is offered by more than one service provider. For various business and technical reasons, an application may need to choose between multiple service providers for the same functionality. For example, multiple services may need to be consulted to check whether the information they provide is consistent. Multiple service implementation can also be used for fault-tolerance purposes.

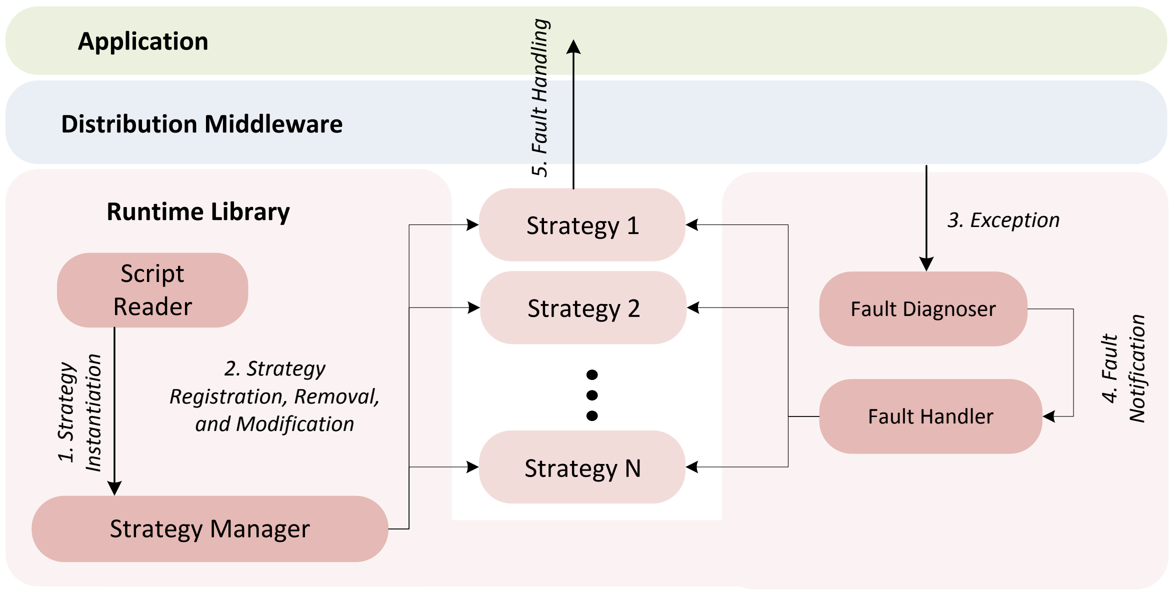


Figure 4.11: Fault tolerant runtime system.

Services providing equivalent functionality are likely to have different service interfaces. One option is to treat the invocation of different equivalent services as unrelated. This way, the client code required to invoke the services is replicated for each service. Another option is to systematically adapt one service's client-side interface bindings for another service interface. This adaptation is automated by means of the *Adapt Service Interface* refactoring.

The *Adapt Service Interface* refactoring automates the transformations required to apply the adapter pattern. Figure 4.12 shows how one service's client bindings can be adapted to use another service. As the first step, a programmer should specify the differences between the original and adapted service interfaces. That is, the programmer uses our refactoring browser to map interface method names to each other. Based on this method name mapping, the refactoring engine generates a skeletal implementation of the required adapter. The programmer can then fill in this skeletal implementation with the adaptation logic. For example, parameters can be simply reordered, missing parameters provided, and extra parameters omitted.

As a specific example, consider switching the remote service invocations of interface `IB` described

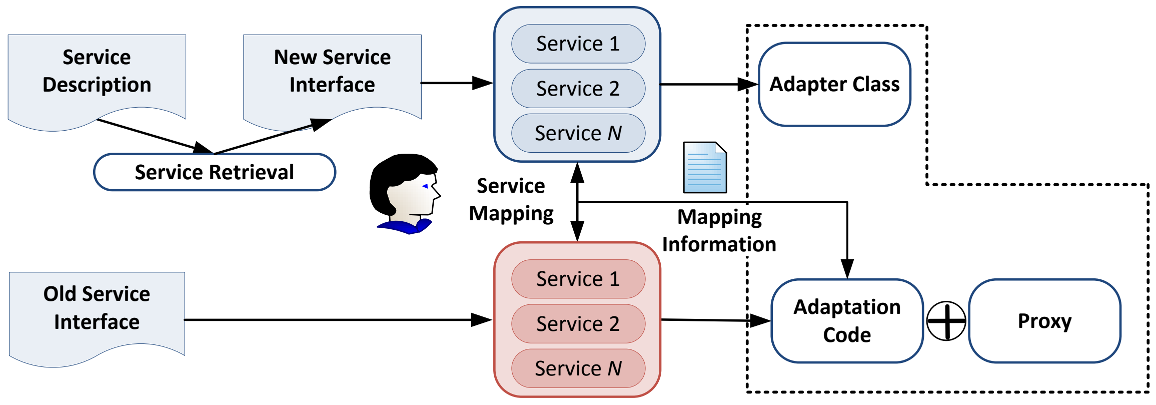


Figure 4.12: Procedure of service adaptation.

in Section 4.2.2 to interface `NewIB`:

```
public interface NewIB {
  String newFoo(int, int, int);
}
```

To switch services, our approach requires that the programmer provides the original and adapted service interfaces. If the adapted service interface is not available locally, the refactoring engine can automatically create one from a WSDL document. Because most web services describe their operations as a WSDL document, a Java interface describing the operations can be retrieved.

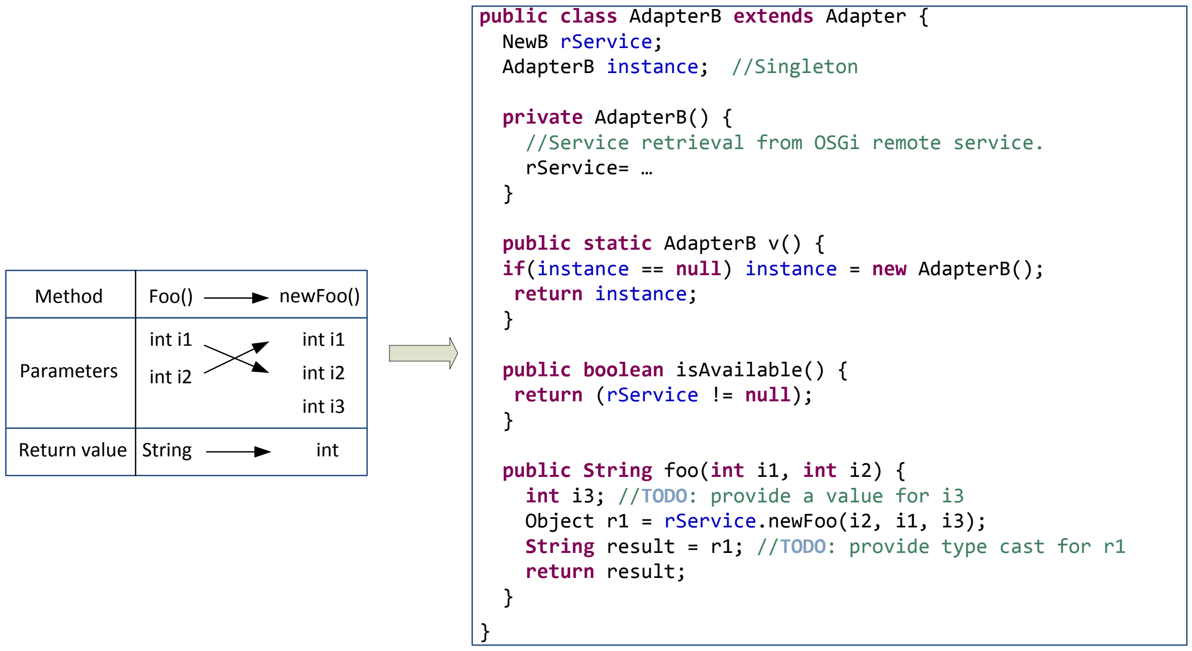
As mentioned above, programmers parameterize the refactoring engine by mapping to each other the original and adapted service interfaces. Figure 4.13 shows how the adaptation code switches the old service invocations to another service's implementation. Figure 4.14 shows the automatically generated adapter class. In this example, method `foo()` is being redirected to method `newFoo()`. The refactoring engine generates an adapter class `AdapterB` which is a singleton. If the adapted service methods differ in terms of their parameter numbers or types, the programmer needs to write code to adapt the parameters and/or return value. This part of the approach is manual, as parameter adaptation is highly application-specific and thus cannot be automated.

```
public class B implements IB { // Proxy class
  public String foo (int i1, int i2) {
    try {
      if(AdapterB.v().isAvailable()) { // Redirected service invocation
        return AdapterB.v().foo(i1, i2);}
      else { // Original service  invocation
        return rService.foo(i1, i2); }
    } catch(CloudServiceException e) {
      return FaultHandler.notify(new Fault(...));
    }
  }
}
```

Figure 4.13: Automatically generated proxy class for service adaptation.

Implementing *Cloud Refactoring*

Figure 4.19 shows the main components of the refactoring tool, which were developed using several state-of-the-art software tools and libraries such as Eclipse plug-ins, OSGi, and Soot Java analysis framework. The refactoring tool consists of three components—1) GUI, 2) recommenda-



```
public class AdapterB extends Adapter {
  NewB rService;
  AdapterB instance;  //Singleton

  private AdapterB() {
    //Service retrieval from OSGi remote service.
    rService= …
  }

  public static AdapterB v() {
  if(instance == null) instance = new AdapterB();
   return instance;
  }

  public boolean isAvailable() {
   return (rService != null);
  }

  public String foo(int i1, int i2) {
    int i3; //TODO: provide a value for i3
    Object r1 = rService.newFoo(i2, i1, i3);
    String result = r1; //TODO: provide type cast for r1
    return result;
  }
}
```

Figure 4.14: Generating an adapter class from interface differences.

tion engine, and 3) refactoring engine. The GUI part was implemented within the Eclipse-IDE's refactoring menus, so that a programmer can easily modify our refactoring and extend the refactored application within the Eclipse-IDE. The recommendation engine was implemented using a static program analysis framework—the Soot Java analysis framework, which manipulates and optimizes Java bytecode. The static analyzer and trace analyzer compute relationships between classes and the service recommender suggests service candidates via the editors and wizards of the eclipse IDE. Lastly, the refactoring engine has a series of code generators including proxy/wrapper generators for remote communications, interface generator for exposing services, adapter generator for switching a service interface.

4.3 Evaluation

To evaluate the applicability of our *Cloud Refactoring* techniques, we applied them to two third-party applications to help transition them to cloud-based execution.

4.3.1 Micro Benchmark: Clustering-Based Recommendation

To evaluate the effectiveness of our recommendation approach, we applied the clustering-based recommendation to seven third-party applications and one our own application.

- Crypto [33]: Java implementation of the Unix crypt utility.
- Compress [33]: Java implementation of the Unix compression utility.
- Dictionary: our own application using the Lucene search engine library [3] to search definitions, find synonyms, and suggest corrects for misspelled words.
- JAligner[4]: an open source Java implementation for biological local pairwise sequence alignment.

[3]`http://lucene.apache.org/java/docs/index.html`
[4]`http://jaligner.sourceforge.net`

Table 4.1: The experimental results.

Name	**# classes**	**# suggested remote services**	**# selected remote services**	**# adaptable remote services**	**ratio**
Crypto	5	4	4	3	80 %
Compress	7	3	3	0	43 %
Dictionary	7	3	3	1	43 %
JAligner	39	8	5	1	12.8 %
Barecue	56	1	1	0	1.7 %
JNotes	164	9	8	0	4.8 %
PMD	597	27	3	0	0.5 %
Weka	1243	14	7	0	0.5 %

- Barecue[5]: an open source Java library to create barcodes.
- JNotes: an open source Java management tool for memos, events and projects.
- PMD[6]: an open source Java program for potential problems like bugs, dead code, suboptimal code, overcomplicated expressions, and duplicate code.
- Weka [55]: an open source Java data mining software implementing a collection of machine learning algorithms.

Table 4.1 shows the benchmark results. Each column represents the number classes, the number of suggested remote services, the number of selected remote services, and the number of adaptable remote services. The first refactoring suggested possible remote services, and we manually selected appropriate remote services. The all suggested classes can be cloud-based services, however, we selected appropriate classes for the reason of the performance, call-by-reference, and meaning of features. Then, the last column shows how many refactored services can be adopted to the third-party services. We found few public Web services through public Web service repositories and manually investigated how the refactored services can be adapted to the new services.

Based on the micro benchmark result, we selected two applications to show refactoring procedures. In the next discussion, we show two case studies—JAligner and JNotes.

[5]`http://barbecue.sourceforge.net`
[6]`http://pmd.sourceforge.net`

4.3.2 Case Study I—DNA Sequence Alignment—JAligner

As the first case study, we applied our refactoring techniques to JAligner—a third-party bioinformatics pairwise sequence alignment tool written to run as a standalone application on a single machine. JAligner takes as input two DNA sequences and computes their similarity metrics. We successfully refactored the application to use a fault tolerant cloud-based service and then switched the alignment functionality to use an equivalent third-party service.

Extracting Remote Service

While the clustering-based recommender suggested 8 classes as potential remote services, the profiling-based recommender suggested 4 classes:

- `Class Commons`: returns basic informations about the application.
- `Class SequenceParser`: parses the given DNA sequence and returns a `Sequence` object.
- `Class SmithWatermanGotoh`: aligns two DNA sequences.
- `Class Example`: returns example DNA sequences.

Although all the recommended classes can be refactored to cloud-based services, in this study we selected only one class—class `SmithWatermanGotoh`, which implements the main functionality of JAligner. For performance reasons, classes `Commons` and `SequenceParser` should not be transformed into cloud-based services. Moreover, because class `Example` forms its own cluster, it should not be moved to the cloud. As the first step, the refactoring engine generated interface `ISmithWatermanGotoh`, which is exposed by underlying middleware, and class `WrapperSmithWatermanGotoh`, a wrapper class of `SmithWatermanGotoh`, as well as some OSGi specific files. The following code snippet shows the automatically generated Java

```
<ftdl>
  <service uri="http://192.168.0.1/SmithWatermanGotoh" method="align"/>
  <condition>
    <timeout>1000</timeout>
  </condition>
  <strategy>
    <sequential numRetries="10" backoffInterval="1000"
    backoffType="linear">
      <service uri="http://192.168.0.2/SmithWatermanGotoh" />
    </sequential>
  </strategy>
</ftdl>
```

Figure 4.15: FTDL description to handle network volatility.

interface, which is exposed through remote OSGi services[7] After the refactoring engine finishes transforming all the code, the newly created service implementation can be deployed in the cloud and accessed remotely.

```
public interface ISmithWatermanGotoh {
  public Alignment align(
    Sequence s1, Sequence s2, Matrix m, float o, float e);
}
```

For the client execution, the refactoring engine re-targets client code to the cloud-based service. To that end, it generates a proxy class— `SmithWatermanGotoh` and OSGi specific files such as remote service configuration files. The generated interface is used for importing the exposed Web service. Through this refactoring, the client is wrapped into a standard OSGi bundle and then uses the Smith-Waterman alignment service over the network. Figure 4.15 shows the FTDL description to handle network volatility.

[7]The services are exposed through Apache CXF-DOSGi.

Adapt Service Interface

We switched the extracted through refactoring remote service—`Smith Waterman` alignment service—to a third-party Web service provided by European Bioinformatics Institute (EBI[8]). EBI provides several bioinformatics Web services, including local and global alignment services. We selected `Waterman Eggert` algorithm and then adapted the client to use this service.

```
<portType name="water"> <operation name="runAndWaitFor">
  <input message="runAndWaitFor"/>
  <output message="runAndWaitForResponse"/>
</operation> </portType>

<complexType name="runAndWaitFor">  <sequence>
  <element name="aSequence" type="SeqInput"/>
  <element name="bSequence" type="SeqInput"/>
  <element name="gapopen" type="float"/>
  <element name="gapextend" type="float"/>
</sequence> </complexType>

<complexType name="SeqInput"> <sequence>
  <element name="direct_data" type="string"/>
  <element name="usa" type="string"/>
  <element name="format" type="string"/>
</sequence> </complexType>
```

Figure 4.16: WSDL contract of the EBI's service.

Figure 4.16 shows the WSDL document that describes the Web service specification. First, based on this WSDL document, we created a Java interface `Water` and its return/argument types such as class `RunAndWaitFor` and `RunAndWaitForResponse`. Figure 4.17 depicts the automatically generated new service interface and other necessary classes. Then, the refactoring generates the skeleton of adapter classes. The only manual part of this refactoring is for the programmer to write code that maps different parameters and return values (e.g., `class Sequence` and `class SeqInput`). As a result, when an unchanged existing client invokes the old service interface, the adapter intercepts the invocation and redirects it to the new service.

[8] `http://www.ebi.ac.uk/soaplab/`

```
public interface SmithWaterMan {
   public RunAndWaitForResponse runAndWaitFor(RunAndWaitFor msg );
}
public class RunAndWaitFor {
  SeqInput aSequence, bSequence;
  float gapopen, gapextend;
}
public class SeqInput{
  String direct_data, usa, format;
}
```

Figure 4.17: Generated interface and classes.

4.3.3 Case Study II—JNotes

As the second case study, we selected JNotes, our motivating example application. We refactored JNotes to used cloud-based services by means of *Extract Service*. As discussed in Section 4.1, we moved class `FileStorage` to the cloud by splitting it into two classes. In this example, we left resource saving functionality at the local machine and moved other functionality to the server. Figure 4.18 shows a proxy class that splits the original class into remote and local parts.

4.3.4 Case Study III: GE Portfolio Analysis Service

We demonstrate how our approach can benefit real companies that want to take advantage of cloud computing. We applied the *Extract Service* refactoring to *Portfolio Analysis Tool*, a real-world application developed by GE Global Research Center and GE Energy to analyze world economy scenarios and predicts how they may affect their customers' billing and costs. The application was developed using standard Web technologies that included the Spring framework and Java servlets.

In particular, Portfolio Analysis Tool 1) calculates billing and costs using hundreds of parameters that are maintained through a DBMS, 2) provides several complex financial components which are computation-intensive functionality, and 3) contains several common functionality that can be reused across multiple applications. Therefore, moving some key components of this application to the cloud would simplify maintenance—the infrastructure (i.e., a Web server, an application

```
public class FileStorage implements Storage {
  IFileStorage proxy;
  LFileStorage local;

  public FileStorage() {
    proxy = (IFileStorage) Activator.v().getService(IFileStorage.class);
    local = new LFileStorage();
  }

  /* Re-targeted methods */
  public void openEventsManager() {
    try {
      proxy.openEventsManager();
    } catch(CloudServiceException e) {
      return FaultHandler.notify(new Fault(...));
    }
  }

  public void openProjectManager() {
    try {
      proxy.openProjectManager();
    } catch(CloudServiceException e) {
      return FaultHandler.notify(new Fault(...));
    }
  }
  //more remote methods

  /* Remaining methods */
  public ResourcesList openResourcesList(Project prj) {
    return local.openResourcesList(prj);
  }

  public void storeResourcesList(ResourcesList rl, Project prj) {
    local.storeResourcesList(rl, prj);
  }
  //more local methods
}
```

Figure 4.18: Generated proxy class.

server, a DBMS, etc.) does not need to be maintained separately for each installation. Moreover, commonly accessed services can be effectively reused.

The recommendation tool of the *Extract Service* refactoring suggested several cloud-based services that can be extracted from the original application. Then, we used our refactoring engine to extract cloud-based services, with the server components deployed in a private cloud environment and the client code transformed to access the cloud-based services remotely.

4.4 Discussion

Next, we discuss some of the advantages and limitations of using refactoring to transition applications to use cloud-computing resources.

4.4.1 Advantages

By automating the required program transformations, a refactoring is more likely to preserve the correctness of a modified program than when a programmer modifies the code by hand. Our cloud refactoring techniques also generate new code used by the modified code. For example, our refactoring engine generates several kinds of proxy classes used at the client. Generating code automatically also helps preserve program correctness. Our recommendation engine also informs the programmer about the parts of the centralized program that can be moved to the cloud while minimizing the incurred performance overhead. Our runtime library features several fault tolerance mechanism implementations that can be used out of the box, thereby increasing the probability that the resulting application will be capable of handling partial failure.

4.4.2 Limitations

A refactoring may not be a proper approach for transforming all kinds of software applications to cloud-based services. First, transforming tightly coupled applications without incurring a significant performance overhead may require deep architectural changes that are not supported by our refactoring techniques. Ensuring good performance requires that remote communication be crude-grained and infrequent. In addition, cloud-based communication is inherently unidirectional: client talks to server but not vice versa. If the original application does not follow this communication pattern, its architecture needs to be changed before our refactoring techniques can be applied.

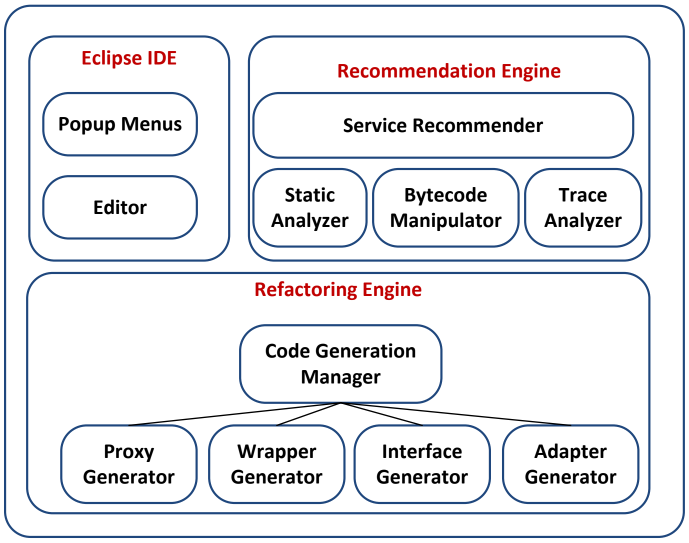


Figure 4.19: Service refactoring tool's components.

Second, to improve accuracy, the recommendation systems require special application-specific parameters. Based on the accuracy of the provided parameters, the recommendation system will show different results. Therefore, the programmer can experiment with different parameters to obtain a recommendation that is most aligned with the business requirements in place.

Third, our refactoring techniques do not make any provision for a situation when a newly extracted cloud service is used by multiple clients. Then the application logic would have to be modified accordingly to ensure a consistent and efficient access by multiple clients.

Lastly, our fault handling strategies cannot cover all the possible failure cases. In some scenarios, the programmer may need to implement some failure handling strategy by hand, outside the framework provided by our refactoring infrastructure.

4.4.3 Motivation for Cloud Refactoring

An important question is what motivates enterprises to move software components to execute remotely in the cloud, thus necessitating the cloud refactoring techniques presented here. One motivation for leveraging cloud resources is to improve performance efficiency by processing large volumes of data in parallel (e.g., using Hadoop). However, this work is motivated by a different set of business cases for using cloud-based resources.

For many business applications, using remote cloud-based service is inevitable, even if the resulting performance efficiency would remain the same or even deteriorate. For example, some shared functionality may need to be shared between multiple clients (e.g., a local accounting component that has to be shared between multiple financial applications). As another example, some functionality may need to be moved into the cloud to take advantage of the cloud provider's data backup and replication services (e.g., a local database-dependent component can be moved to a cloud service along with its database files to guarantee long-term data integrity). Finally, companies may consolidate some replicated functionality and expose it as a cloud service to reduce the software maintenance efforts. Because services are exposed through a public service interface, the service's implementation can change at will without perturbing its clients, as long as the service interface remains fixed.

All these scenario represent a clear need for the cloud refactoring techniques discussed in this section, even though the refactored (i.e., cloud-based) versions of these applications are unlikely to show any increase in performance efficiency. However, unless the invocation of cloud services is in the critical performance paths of these applications, the overall performance impact of cloud refactoring is likely to remain insignificant. From the business perspective, migrating services to the cloud can reduce the overall software development costs and can even enable companies to break into new markets, as software-as-a-product (SaaP) can be easily reused and repurposed.

4.5 Related Work

The presented Cloud Refactoring techniques are related to program partitioning, software clustering, migrating application to services, and fault handling techniques. Next, we compare and contrast our techniques to the most relevant approaches in each of these categories.

One line of research has explored coarse grained program partitioning. The programmer, by means of a GUI, designates different parts of a centralized application, typically at a class or object granularity, to run on different network nodes. The resulting distribution specification then parameterizes a compiler-based tool that automatically rewrites the centralized application for distributed execution. To introduce distribution, a partitioning tool may need to both change the structure of the application (e.g., to introduce a proxy indirection) and add middleware functionality (e.g., to replace local calls with remote ones). In the Java world, recent automatic partitioning tools include Addistant [142], Pangaea [133], and J-Orchestra [153]. Addistant and J-Orchestra partition programs at a class granularity; Pangaea can partition at the individual object level. J-Orchestra addresses the challenges of partitioning programs safely in the presence of unmodifiable code that comes as part of their runtime systems.

Several prior research efforts aim at decomposing software systems into subsystems using clustering techniques [98]. The Bunch tool [98] uses a variety of clustering algorithms (e.g., hill climbing, genetic, etc.) to modularize existing systems; it extracts modules based on their dependence graph and calculates the resulting modularity quality. Clustering can be based on structural data (e.g., dependence graphs) and non-structural data (e.g., names, comments, behavior, etc.) [6]. Combining structural and non-structural clustering can improve the resulting modularity [5].

Much research has gone into decomposing the large, legacy systems into sub systems by assistance of the above clustering techniques. Such decomposition was performed for better understanding to the systems or maintenance for very large systems. However, in recent research, there was an attempt to adopt data mining techniques for partitioning into distributed applications or service oriented applications, including RuggedJ [93]. RuggedJ adopted a classification techniques which determines classes' types and locates them distributed nodes such as server/client.

In addition, our approach is related to migrating legacy systems toward objects [90] and services [17]. The module dependence graph has recently been shown to be effective at guiding the migration toward services [86]; loosely-coupled modules become service components. In addition, a model-based approach has been proposed to extract UML from legacy code and to use proxy wrappers as service interfaces [91].

Commonly used approaches to recover from failure include termination [130], restarting [138], micro-rebooting [16], and checkpoint-restart [73]. Although traditionally such approaches have been integrated with system design, our approach can expose them as a fault strategy configured through FTDL. Therefore our approach can increase the resiliency against faults even further.

4.6 Conclusion

In this chapter, we have presented *Cloud Refactoring*, a set of semantics preserving transformations that can help migrate a centralized application to using cloud-based services. We realized *Cloud Refactoring* in the context of a modern IDE, enhancing its refactoring engine. *Cloud Refactoring* comprises two main refactoring techniques: *Extract Service* and *Adapt Service Interface*. The *Extract Service* refactoring renders a portion of a centralized application's functionality as a remote cloud-based service, rewriting the client code and enhancing it with the required fault-tolerance strategies. The *Adapt Service Interface* refactoring automates the transformations needed to switch a service client to use an alternate, equivalent cloud-based service. We have evaluated *Cloud Refactoring* by transforming third-party applications to cloud-based services, including an application used by General Electric. Our experiences indicate that refactoring can become a valuable tool in the toolset of software developers charged with the challenges of migrating applications to take advantage of cloud-based resources.

Chapter 5

Adaptive Cloud Offloading to Improve Energy-Efficiency

The hardware capacities of modern mobile devices are rapidly approaching those of desktops from the recent past. The majority of today's smartphones, tablets, and e-readers feature multicore CPUs, large RAMs, high resolution displays, and fast mobile networks. These powerful hardware capacities, in turn, lead to mobile applications with increasingly complex computation and communication patterns. Ideally, the growth of application functionality would be matched with corresponding increases in device battery capacity. Unfortunately, physical constraints stunt the improvements in battery capacities, which are known to increase quite moderately [116]. As a result, energy consumption is not only a major resource constraint for modern mobile devices, but it also impedes the mobile programmer's creativity and productivity. Indeed, mobile application developers have no choice but to remain mindful of how computation- and communication-intensive pieces of functionality would affect the overall battery life.

Cloud offloading has been proposed as a mechanism that reduces the energy consumed by mobile applications [26, 20, 127]. Cloud offloading partitions a mobile application into local and remote parts to execute some energy intensive functionality remotely. Thus, cloud offloading leverages network communication and remote execution to reduce the energy consumed by mobile devices.

The high heterogeneity of mobile hardware and the volatility of mobile networks stand in the way of efficiency in implementing cloud offloading optimizations. Mobile devices come in numerous hardware configurations, which vary in terms of their respective CPU, memory, and communication infrastructure. In addition, as we have discovered, the mobile network's conditions can affect the amount of energy spent on transferring the same parameters between the mobile device and the server [79].

Static cloud offloading techniques first determine which application functionality is energy intensive, and then partition mobile applications accordingly. However, to improve its efficiency, cloud offloading must also take into account the specific runtime parameters of a given mobile execution, including the device's hardware capacities and the network's type and condition. In other words, the offloading decisions should be made dynamically at runtime and continuously adjusted as the mobile execution environment changes. Thus, runtime adaptation is the key for improving the efficiency of cloud offloading.

This article presents a novel cloud offloading approach that reduces the energy consumption of mobile applications by leveraging automated program transformation and runtime adaptation. First, a mobile application is transform to make it possible to offload parts of its execution to the server; then, what gets offloaded is determined at runtime as required by the specifics of the execution environment. In other words. a mobile application is transformed into a distributed application, whose local and remote parts are determined at runtime. The flexibility in determining the distribution patterns at runtime is enabled through an elaborate checkpointing mechanism. Depending on the runtime execution environment, different portions of a program's state can be checkpointed and transferred across the network as required by the offloading strategy in place. An adaptive runtime system efficiently switches between local and remote executions, both to reduce client energy consumption and to tolerate network volatility.

Applying our approach to third-party, real-world Android applications has reduced their energy consumption while improving their performance characteristics. Based on these results, the technical contributions of this article are as follows:

1. **Adaptive cloud offloading**—a novel energy optimization mechanism for mobile applications whose operation is driven by the actual runtime parameters.

2. **Cloud offloading analysis**—a new program analysis technique that safely identifies the exact program state that must be transferred across the network during cloud offloading.

3. **Cloud offloading program transformation**—a technique for enhancing a mobile application with optional checkpoints that enable the adaptive offloading.

4. **Adaptive runtime system**—a middleware mechanism that efficiently monitors the environment to determine optimal offloading strategies at runtime.

5. **Empirical evaluation**—a set of benchmarks and case studies that demonstrate the effectiveness of adaptive cloud offloading as a powerful mechanism for reducing the energy consumed by real-world, third-party mobile applications.

The rest of this chapter is structured as follows. Section 5.1 motivates this work, while Section 5.2 introduces the technologies used to implement our approach. Section 5.3 describes our approach and reference implementation. Section 5.4 evaluates our approach with third-party applications. Section 5.5 discusses advantages and limitations of our approach. Section 5.6 compares our approach to the existing state of the art. Finally, Section 5.7 presents concluding remarks.

5.1 Motivation and Research Questions

In this section, we first present a motivating example and then discuss the technical problems addressed in this work.

5.1.1 Motivating Example

Consider Mezzofanti—a third-party, augmented reality application that runs on the Android platform. This application guides travelers visiting foreign countries. When traveling internationally, language differences often become a major source of inconvenience particularly in the locales with writing systems unfamiliar to the traveler. Mezzofanti enables the traveler to use a camera to capture the image of printed text in any language and obtain an automated translation. Mezzofanti uses optical character recognition (OCR) and automatic language translation. Unfortunately, recognizing characters optically is a computationally intensive task, known to heavily consume battery power. Therefore, a foreign visitor using Mezzofanti frequently is likely to quickly run out of battery power, rendering this electronic translation aid unusable.

To reduce the amount of energy consumed by Mezzofanti, one can execute its OCR functionality at a remote server, with the mobile device only showing the results computed and returned by the server as shown in Figure 5.1. To put this energy optimization into effect, one can partition the application into local and remote parts, communicating with each other across the network. However, the heterogeneity of mobile hardware and the dissimilarities of execution environments can quickly render such a static optimization ineffective. For example, the mobile network may be unavailable or limited, making it impossible for the client and server parts to communicate. Thus, maximizing energy savings requires that cloud offloading decisions be made dynamically at runtime. Realizing this dynamic optimization presents a number of technical challenges that we discuss next.

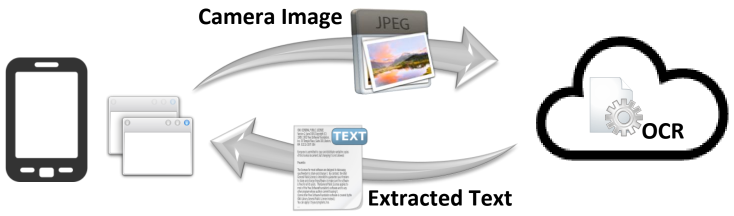


Figure 5.1: Optimizing Mezzofanti with cloud offloading.

5.1.2 Problem Definition and Research Questions

Applying cloud offloading optimizations is hard due to the high heterogeneity of mobile hardware and the diversity of mobile networks. The hardware setups of mobile devices have different CPU, memory, and communication capacities. For example, Facebook reports that the mobile version of their application is accessed from more than 2,500 varieties of mobile devices [40]. At any given time, a mobile device can be connected to the cloud by means of mobile networks whose quality and capacities differ by a wide margin (e.g., WiFi, 3G, 4G, etc.). Indeed, as we have discovered in a recent study, the mobile network characteristics in place determine how much energy will be spent to transfer the same amount of data across the network [79]. The net effect of these dissimilarities is that a mobile application running on different devices connected to the cloud by different networks consume different amounts of energy.

To optimize energy consumption, cloud offloading mechanisms must be implemented efficiently, which requires transferring only the needed program state and aligning the offloading operations with the current runtime conditions. Transferring data over a mobile network also incurs energy costs for mobile devices. Indeed, transferring large data volumes across the network can quickly negate the energy saving benefits afforded by offloading. The research literature shows that the average size of a Java heap commonly exceeds hundreds of MBs [33]. Therefore, one must strive to transfer only the portion of program state that will be used by an offloaded functionality. Because it takes more energy to transfer data across limited networks, an optimal offloading strategy needs to trade the energy potentially saved by moving the execution to the cloud with the energy consumed by transferring the program state to support the offloading. All in all, effective offloading mechanisms should be efficient and adaptive.

The approach described in this article enables adaptive cloud offloading as a means of reducing energy consumption of mobile applications by attacking the following fundamental questions:

1. How can one determine optimal program units that can be offloaded to reduce the energy consumed by mobile devices with dissimilar hardware setups connected by diverse networks?

2. How can one reduce the program state's size to make state transfer a pragmatic energy consumption optimization technique?

3. What kind of adaptation strategies should be put in place to drive offloading operations in heterogeneous hardware and network environments?

5.1.3 Solution Overview

To address the first research question, we introduced the Energy Consumption Call Graph (ECG), a novel program analysis data structure that models how much energy will be consumed under different offloading scenarios. The nodes of ECG represent program components, encapsulated units of functionality that can be offloaded to the cloud. Each node is labeled with an approximate amount of energy consumed by the CPU to execute the functionality of the component and its successor components in the graph. The edges represent the communication between the components, with the labels showing how much energy will be consumed by the mobile device to transmit the data between the connected components.

To address the second research question, we created a program analysis technique that precisely computes the program state that needs to be transferred across the network during offloading operations. The technique leverages forward dataflow and side effect analyses to reduce the checkpointed program state by orders of magnitude, thus rendering state transfer practical for energy optimization. The computed state is then efficiently checkpointed and synchronized upon the completion of offloading operations.

To address the third research question, we introduced an adaptive runtime system that drives offloading operations dynamically by continuously monitoring the runtime environments and estimating the energy saving benefits of offloading operations. As an example of adaptivity, an offloading operation would take place only when a device is connected to the cloud over a WiFi or a 4G, but not a 3G network.

5.2 Technical Background

Our approach combines distributed mobile execution, program analysis, program state synchronization. We describe these technologies in turn next.

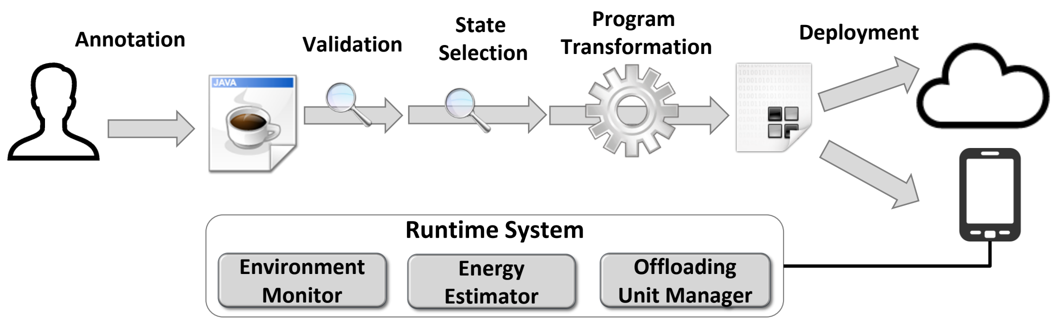


Figure 5.2: Adaptive cloud offloading process.

5.2.1 Distributed Mobile Execution to Save Energy

Distributed mobile execution can save energy by executing a mobile application's energy intensive functionality in the cloud, without draining the mobile device's battery. This optimization is typically implemented as a program partitioning transformation that splits a mobile application into two parts: client running on a mobile device and server running in the cloud; all the communication between the parts is conducted via a middleware mechanism such as XML-RPC. Thus, distributed mobile execution to save energy can be enabled via automated program partitioning—distributing a centralized program to run across the network using a compiler-based tool transform a centralized program [155] or migrating execution between different application images at the OS level [127, 23]. The promise of this technique is demonstrated by the proliferation of competing approaches in the literature. CloneCloud [20] offloads execution at the thread level, while Cloudlet [127] offload at the VM level. MAUI [26] offloads through application partitioning at the method level. In our prior work on cloud offloading [78, 80], we partition applications without destroying their ability to execute locally. All of these prior cloud offloading techniques share the goal of reducing the energy consumed by mobile devices.

5.2.2 Program Analysis

Static program analysis examines a program's code to infer useful facts that inform a variety of tasks, including program testing, optimization, and transformation. Points-to analysis constructs a call graph in object-oriented languages. A control flow graph (CFG) models possible execution paths of a program based on its conditional and looping constructs. Side-effect free analysis [121] determines whether a method changes the program's heap. Dataflow analyses determine which particular program variable is assigned to which variables [69]. Dataflow analyses operate on a method's CFG to calculate reachable variables at each statement. Because dataflow analysis algorithms are inter-procedural, a whole program must be analyzed to calculate a single variable's flow.

We combine dataflow and side-effect analyses to compute the program state that needs to be transferred across the network during offloading operations. The computed state is then checkpointed by directly modifying programs at the bytecode level. For both program analysis and transformation, we used the popular Soot [157] framework.

5.2.3 Program State Synchronization

To allocate objects dynamically, runtime systems of object-oriented languages use the memory region referred to as the heap. In a distributed environment, multiple heaps or their portions can be synchronized across nodes. When synchronizing heaps, aliasing—pointing to the same object by different references—complicates the process, as all the aliases to the synchronized objects must remain in place. An effective approach to synchronizing linked data structures (e.g., linked lists, trees, and maps) is to use the copy-restore semantics for remote parameters [154]. This semantics copies all reachable data to the server and then overwrites the client copy of the parameter with the server modified data in-place (i.e., while preserving the client-side aliases).

5.3 Adaptive Cloud Offloading

In this section, we present our approach by giving an overview of the approach and then describe its major parts in turn.

5.3.1 Approach Overview

Figure 5.2 shows the main workflow of adaptive cloud offloading. The programmer is only responsible for annotating those program methods that are known to consume energy heavily. The question of how such methods are identified is orthogonal to our approach: energy profiling can be used or domain knowledge can be leveraged. The cloud offloading analyzer first checks whether the annotated methods can be offloaded and then determines which portion of the program's state would need to be sent to the server. Based on the computed transferred state information, the code enhancer inserts the checkpoint (at the bytecode level) that captures the state that is updated on both the local and remote heaps. The adaptive runtime system continuously monitors the energy consumed by each offloading candidate component and estimates their energy consumption. Based on the estimated energy consumption, the runtime offloads those components whose cloud-based execution would yield energy savings or performance benefits for the given network conditions rather than executing them locally. The runtime also synchronizes remote and local states in place. Yet another responsibility of the runtime system is fault tolerance—handling temporary network disconnection and volatility. In the following discussion, we describe the aforementioned components in turn next.

5.3.2 Programming Model

Our programming model is straightforward: the programmer marks the components suspected of being energy hotspots. To mark hotspot components, we provide a special Java annotation `@OffloadingCandidate`. Based on this input, an analysis engine first checks whether the spec-

```
public class OCR {
  //member fields
  ...
  OCRConfig ocrConf;
  SpannableString ssResult;

  public void init() { ocrConf = new OCRConfig(); }

  @OffloadCandidate
  public void imgOCRAndFilter(Image img) {
    String ocrResult = process(img);
    ssResult = new SpannableString(ocrResult);
  }

  public SpannableString getParsedResult() {
    return ssResult;
  }

  private String process(Image img) {
    //process img using ocrConf
  }
}
```

Figure 5.3: Motivating example revisited.

ified component can be offloaded as well as any of its subcomponents (i.e., successors in the call graph). The engine also calculates the program state, to be transferred between the remote and local partitions, that would need to be transferred to offload the execution of both the entire component or of any of its subcomponents.

To demonstrate our programming model, we revisit the Mezzofanti application first introduced in Section 5.1.1. Figure 5.3 partially lists class `OCR` that recognizes the textual representation of a given image. Method `imgOCRAndFilter()` extracts text from its `Image` parameter, storing it in member variable `ssResult`. Having identified this methods as energy intensive, the programmer annotates it with `@OffloadCandidate`. To run correctly, `ImgOCRAndFilter()` needs to have member field `ocrConf` properly initialized, an operation performed in method `init()`. Notice that `ImgOCRAndFilter()` accesses `ocrConf` indirectly by calling method `process()`.

Because method `imgOCRAndFilter` does not use any client hardware-specific API, it can be offloaded. To execute this method at the server, we need an instance of class `OCR` whose member variable `ocrConf` is initialized. No other member variables are accessed by `imgOCRAndFilter`, so that transferring them to the sever would be wasteful. When method `imgOCRAndFilter` completes its energy intensive execution on the server, member variable `ssResult` will be modified, so that it contains the method's result. Only this member variable needs to transferred back to the client and integrated into the client heap.

5.3.3 Cloud Offloading Analysis

To compute the program state that needs to be transferred during future offloading operations, we analyze target application in a two-step procedure that includes a pre-analysis and a state-selection analysis, as depicted in Figure 5.4. The pre-analysis step determines whether the marked methods can be offloaded. Then, the state-selection analysis identifies the state that needs to be transferred between the client and the server and vice versa.

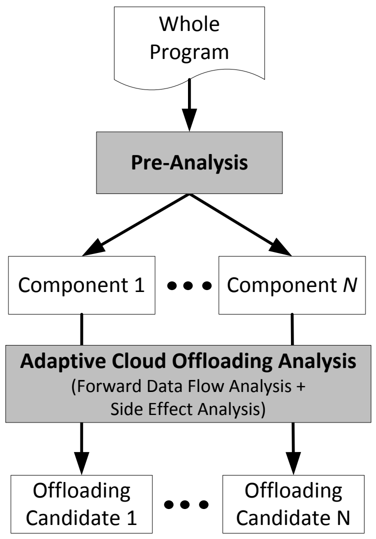


Figure 5.4: Program analysis for adaptive cloud offloading.

Our validation algorithm takes as input a call graph and the client-only API information. Only the methods that do not contain any client-only APIs (e.g., those controlling the GPS, camera, microphone, etc.) can be offloaded. This check simply traverses the program's call graph and checks the reachable statements for the presence of the known libraries that control the mobile device's hardware components (e.g., `android.hardware.Camera.*` controlling the camera on Android-based devices). This simple heuristic turned out to be quite effective in identifying the methods that cannot be offloaded.

Energy Consumption Call Graph

Once offloading candidates are identified, an energy consumption call graph (ECG) is constructed. As a specific example, component `A` of Figure 5.5 consumes approximately 100 joules during a typical execution, thus becoming a viable candidate to be offloaded to the cloud. Because component `A` calls components `C` and `B`, which in turn calls components `D` and `E`, its energy consumption is the sum of the energy consumed by all the successor components in the graph. We assume that the energy spent on executing the offloaded functionality in the cloud is free, as it does not exhaust any battery power of the mobile device. If component `A` is offloaded, then transmitting the necessary data to it across the network enabling it to execute remotely would consume between 30 and 150 Joules depending on the type of the network available to transmit the data. In other words, under some network conditions, offloaded execution will end up using more energy than executing component `A` on the mobile device. As a specific example of using the ECG above, when operating over a 3G network, components `C` and `D` can be offloaded, while when operating over a 4G network, only component `E` can be offloaded. Finally, when operating over WiFi, components `A` or `B` can be offloaded. Because the type of network available is only known at runtime, the offloading decisions must be dynamic to be able to optimize the amount of consumed energy under all runtime conditions.

State-Selection Analysis

Next, the state-selection analysis identifies the state that needs to be transmitted across nodes. The purpose of this analysis is to reduce the size of the transferred program state, lest the transfer costs negate the energy savings afforded by offloading. Figure 5.6 shows our state selection algorithm

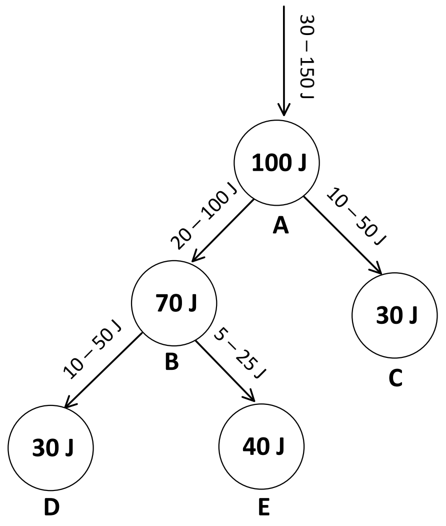


Figure 5.5: Energy consumption call graph.

that identifies those member variables that have to be passed to an offloaded method and back to the mobile device. The algorithm combines forward dataflow and side-effect analyses. The forward data-flow analysis keeps track of local variables of the offloaded methods by calculating the entry and exit of each analyzed statement in the control-flow graph. The analysis examines assignment and invocation statements to determine whether local variables are changed. For each assignment statement, the following cases are considered:

- If the left value is a class member variable, it is marked as required for the client.
- If the right value is a class member variable, it is marked as required for the remote server.

Member variables read directly are marked as required for the remote server. However, member variables aliased via local variables are analyzed transitively. To that end, a special variable relationship graph is consulted to identify the member variables reachable from the local variables. In addition, because our forward dataflow analysis is inter-procedural analysis, it is applied to all the methods in the call graph reachable from the offloaded method.

In addition to the assignment statements, invocation statements are considered to determine the required state. Invocation statements can be categorized as follows:

```
SelectState($method$) {
  $readValues, writtenValues \leftarrow \emptyset$

  $allStmts \leftarrow$ getAllStmts($method$);
  $objGraph \leftarrow$ constructObjectGraph();

  FOREACH $stmt \in allStmts$ DO
    IF $stmt$ is an assignment statement THEN
      $lValue \leftarrow$ getLeftOp($stmt$);

      IF $lValue$ is a member variable THEN
        $writtenValues$.add($lValue$);
      ELSE IF $lValue$ is a transitional variable THEN
        $root \leftarrow variableGraph$.getRoot($lValue$);
        $variableGraph$.add($root$, $lValue$);
      END IF

      IF $rValue$ is a member variable AND
         $variableGraph$ contains $rValue$ THEN
        $readValues$.add($rValue$);
      END IF

    ELSE IF $stmt$ is an invocation statement THEN
      $target \leftarrow stmt$.getInvocationTarget();
      $m \leftarrow target$.getMethod();

      /* recursive call */
      $values \leftarrow$ SelectState($m$);
    END IF
  END FOREACH
```

Figure 5.6: Algorithm for state selection.

- If an invocation is on a member variable, it is marked as needed for the remote server.
- If an invocation on a member variable changes any member variables, the changed variables are marked as need for both the client and the remote server (i.e., need to be transfered in both directions).

If an invocation is on indirectly accessed member variables, the algorithm determines the root member variable by traversing the variable's relationship graph. To determine whether the invocation target changes the heap, we employ a side-effect free analysis [121]. If the invocation changes the heap, we mark the invocation's receiver object as needed for both the client and the remote server. For example, if an invocation target method is `java.utils.HashMap.put()`, we analyze it for the absence of side-effects. Because method `put()` changes the heap, we determine that the entire `HashMap` member field needs to be transfered in both directions.

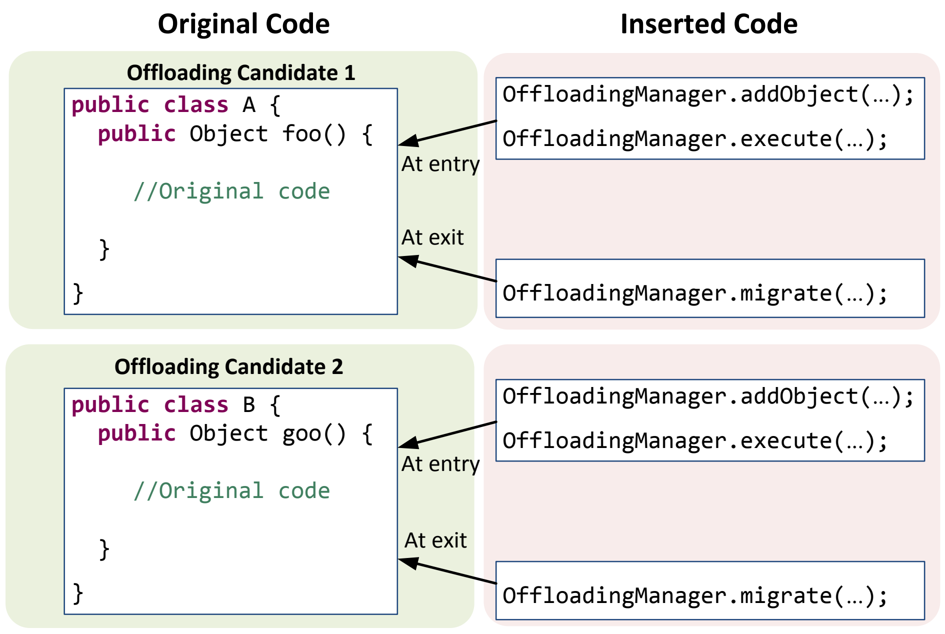
The analysis is conservative—it may mark a variable as needed for the server, even though at runtime the variable would not be used. In other words, the algorithm can produce false positives, but never false negatives. Thus, the algorithm is sound. Nevertheless, a high level of false positives can hinder the adaptive runtime system's effectiveness. When predicting whether an offloading would be beneficial, the runtime system uses the analysis's results.

5.3.4 Enhancing Bytecode to Enable Offloading

Based on the selected program state, the potentially offloaded methods are then transformed to make it possible to run them on the server, with the results transferred back to the mobile device. To that end, the bytecode enhancer transforms the offloaded methods into cloud and server versions. Specifically, the code to checkpoint and restore the necessary program state is inserted at the entry and exit points of the offloaded methods, respectively. Figure 5.7 shows how the original code of a centralized mobile application is transformed. The inserted checkpoints are executed conditionally, as driven by the runtime system, which transfers the checkpointed data between the mobile device and the server and vice versa.

5.3.5 Adaptive Runtime System

Figure 5.8 shows the design of the runtime system that comprises three main components: (1) energy consumption estimation, (2) cloud offloading steering for adaptation, and (3) cloud offloading engine. In the following discussion, we describe each process in detail.



```
public class A {
  public Object foo() {

    //Original code

  }
}
```

```
OffloadingManager.addObject(…);
OffloadingManager.execute(…);
```

```
OffloadingManager.migrate(…);
```

```
public class B {
  public Object goo() {

    //Original code

  }
}
```

```
OffloadingManager.addObject(…);
OffloadingManager.execute(…);
```

```
OffloadingManager.migrate(…);
```

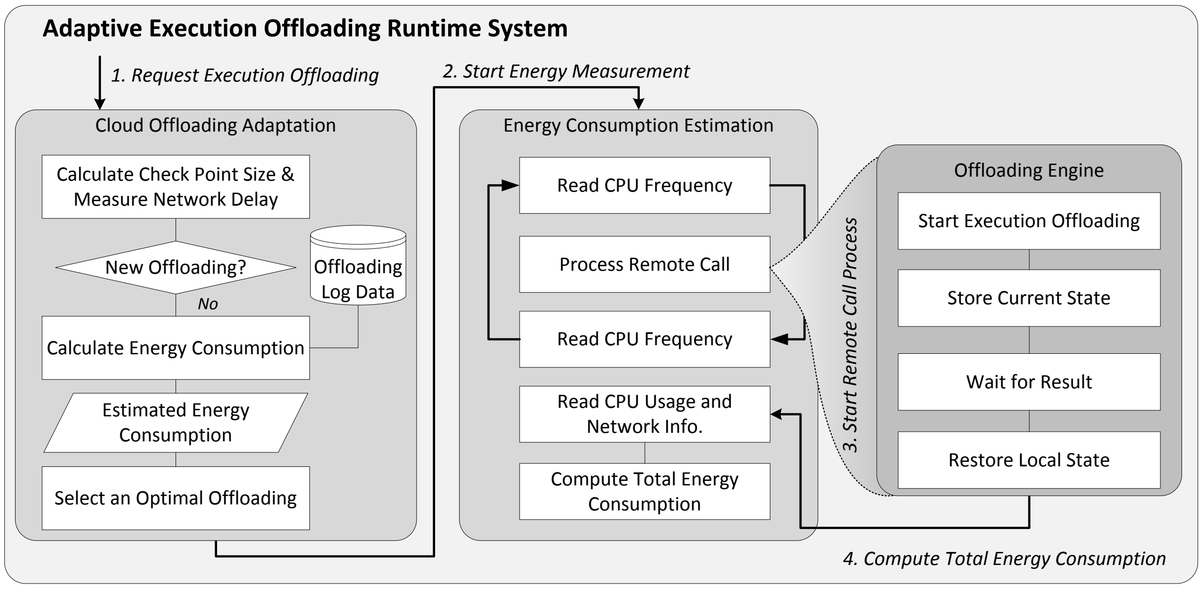
Figure 5.7: An example of program transformation.

Energy Consumption Estimation

The energy consumption estimation module computes the future energy consumption by analyzing the collected runtime execution environment and then makes a cloud offloading decision when the current cloud offloading is beneficial. First, to compute the amount of energy consumed by cloud offloading, we take system parameters and measured runtime parameters as follows:

$$\begin{aligned} \boldsymbol{E} &= E_{cpu} + E_{net} = (P_{cpu} \times T_{cpu} + P_{net} \times T_{net}) \\ &= \{\Sigma(C^{act}_{cpu_f} \times T^{(u+s)}_{cpu}) + (C^{act}_{net} \times T^{act}_{net}) + (C^{idle}_{net} \times T^{idle}_{net})\} \times V \end{aligned}$$

where $C^{act}_{cpu_f}$ is the electric current of the CPU at a particular clock speed. Modern CPUs feature speed-step, a facility that allows the clock speed of a processor to be dynamically changed by the operating system, with different levels of power consumed at each clock speed. For example, Samsung Galaxy S III's AP provides five steps, ranging from 302.4MHz to 1512MHz, and the required electric current at each speed ranges from 55mA to 577mA. T^{u}_{cpu} and T^{s}_{cpu} are user and



Adaptive Execution Offloading Runtime System
1. Request Execution Offloading
2. Start Energy Measurement
Cloud Offloading Adaptation
Calculate Check Point Size & Measure Network Delay
New Offloading?
No
Offloading Log Data
Calculate Energy Consumption
Estimated Energy Consumption
Select an Optimal Offloading
Energy Consumption Estimation
Read CPU Frequency
Process Remote Call
Read CPU Frequency
Read CPU Usage and Network Info.
Compute Total Energy Consumption
3. Start Remote Call Process
Offloading Engine
Start Execution Offloading
Store Current State
Wait for Result
Restore Local State
4. Compute Total Energy Consumption

Figure 5.8: Process details of adaptive cloud offloading.

system times taken by the offloading operation, and they are obtained by consulting the statistics provided by the operating system (e.g., `/proc/[pid]/stat`). V is current voltage, which is also obtained from the operating system (`/sys/class/..../voltage_now`). C_{net}^{act} and C_{net}^{idle} are the electric current required by the network processor during the active and idle phases, respectively. For example, the active/idle electric current for Samsung Galaxy S III is 96mA/0.3mA during WiFi communication, and 250mA/3.4mA during mobile communication (e.g., 4G). Finally, T_{net}^{act} and T_{net}^{idle} are active and idle time periods during the cloud offloading operation, measured at runtime. These device- and execution-specific values are used to measure the amount of energy consumed during each offloading optimization.

Another important responsibility of the energy estimation module is to predict future energy consumption. To predict the energy that is likely to be consumed during an offloading optimization, it correlates the previously measured energy consumption and the current execution environment. Having obtained the current values of network delay, network connection type, CPU frequency, and voltage, the future energy consumption is computed as follows:

$$\boldsymbol{E_{prd}} = \{E_{cpu}^{avg} + (C_{net}^{act} \times T_{net}^{prd\_act}) + (C_{net}^{idle} \times T_{net}^{prd\_idle})\} \times V$$

where E_{prd} is the predicted future energy consumption, E_{cpu}^{avg} is the average energy consumption of the given remote communication, and $T_{net}^{prd\_act}$ is the predicted communication time, which are computed by using the transferred data size and delay, respectively. To avoid a delay spike, the current delay value is then recomputed by weighting the most recently obtained value (i.e., $delay = delay \times \alpha + delay \times (1 - \alpha)$. α was set to 0.3 in our reference implementation). The computed energy consumption value is used as a parameter for selecting the energy optimization strategy for a given execution environment.

Cloud Offloading Steering

The cloud offloading steering module decides whether a given method shoudl be offloaded by comparing offloading candidates' energy consumption and execution time. Figure 5.9 shows the procedure for selecting offloading units at runtime. To select a unit to offload, the module uses the future energy consumption and the future execution time predicted by analyzing the collected runtime execution values and cached prior executions. Then, the module selects an offloading unit that would be expected to consume the lowest amount of energy.

In some cases, offloading energy-intensive functionality can both increase performance and save energy. However, in other cases, cloud offloading can decrease performance, but save energy consumption, and vice versa. The runtime system drives offloading operations by considering both performance improvement and energy savings. To that end, we introduce *cloud offloading index (COI)* as follows:

$$COI = \frac{OET}{ET} \times \alpha + \frac{OEC}{EC} \times (1 - \alpha)$$

where EC and OEC are original and optimized energy consumption, respectively; ET and OET are original and optimized execution times, respectively. If ET/OET is less than 1, the offloading optimization will increase the application's performance. Similarly, if EC/OEC is less than 1, the

```
/** Find a best candidate */
FOREACH $candidate \in \forall Candidates$ DO
  $E_p \leftarrow$ predictEnergy($candidate$);
  $T_p \leftarrow$ predictExecTime($candidate$);
  $coi \leftarrow$ getCOI($E_p, T_p$, $\alpha$)

  IF $coi$ is the smallest among all candidates THEN
    $bestCandidate \leftarrow candidate$ END IF
END FOREACH

/** Offload the selected offloading candidate */
IF current execution path is $bestCandidate$ THEN
  $newState \leftarrow$ offload($bestCandidate$, $state$);
  stateMigration($newState$, $state$);
  $E_m \leftarrow$ getConsumedEnergy($bestCandidate$);
  updateHistory($E_m$, $bestCandidate$);

  CASE Succeed
    $result \leftarrow$ executionCompleted()
  CASE Fail
    $result \leftarrow$ executionFailed()
  return $result$
END IF
```

Figure 5.9: The procedure to select an offloading unit.

offloading optimization will reduce the application's energy consumption. The programmer can use the α parameter (ranging between 0 and 1) to express whether the optimization should favor performance or energy. To consider performance for offloading operations, the α parameter should be greater than 0. When α is exactly 0.5, the focus is on both increasing performance and reducing energy consumption. Finally, if the computed COI value is smaller than 1, which means remote execution is more beneficial than local execution, the runtime system executes the marked energy hotspot remotely.

Additional Optimization via Flexible Configurability

To save even more energy, one can tailor cloud offloading for specific applications and users, so that the runtime system can take these contextual details into account when dynamically adapting its execution patterns. To that end, we enable advanced users to configure the runtime system to specify how state-of-the-art, fine-grained optimization strategies can be applied to offloading network communication.

```
hotspot=[methodName]
host=[url]+
strategy=[name]
weight=[0...1]
```

A configuration file contains a set of key/value pairs, with the keys of `hotspot`, `host`, `strategy`, and `weight`. The `hotspot` key points to the method identified as an energy hotspot. The `host` key points to the locations of the available offloading servers. The `strategy` key points to well-known energy optimization techniques, including data compression, reducing image quality, and redirecting to an easier-to-reach remote server. Finally, the `weight` key points to the W value of COI to express whether the optimization should favor performance or energy.

Next, we give specific examples of optimization strategies—1) data compression and 2) redirecting to an easier-to-reach remote server. Data compression can reduce network transfer, but will be more computationally intensive, thus requiring additional CPU processing. Transmitting raw data increases network transfer, but requires less CPU processing. Which of the strategies will consume less energy depends on the runtime conditions in place. Another optimization technique is redirection. This strategy iterates through different endpoints of a distributed execution in the case of experiencing poor network conditions. For instance, when experiencing a network congestion at `a.com/foo`, `alt.a.com/foo` can be invoked instead. This strategy, thus, will find an optimal execution path, as operating over a congested network is likely to require additional energy resources. In addition to the aforementioned general optimization strategies, one can also apply application-specific optimizations, tailored for particular application scenarios.

Efficient State Synchronization

The cloud offloading engine manages remote connections between the offload server and the mobile client, synchronizes the program state, and provides resilience in the presence of failure due to network volatility. The stored program state is synchronized by means of copy-restore, an advanced semantics introduced into remote method call middleware with the goal of passing as parameters liked data structures (e.g., linked lists, trees, and maps) [154]. Copy-restore efficiently copies all reachable state to the server and then overwrites the client's state with the server modified data in-place. To adapt to operating over cellular networks with limited bandwidth, we modified the original copy-restore implementation to use sparse arrays, which encode `null` values space efficiently. Our implementation uses `null` values to mark the portion of the transferred state that has not been mutated during the offloaded operations.

Figure 5.10 demonstrates how the runtime system synchronizes the checkpointed state. Graph (a) depicts the mobile device's state to be transferred to the server. The runtime system transfers only the nodes that the analysis identified as being used by the server (nodes 2, 3, 4, and 7). Graph (b) depicts the server's state before it is synchronized with the transferred state. Nodes 2, 4, and 5 are updated with new values; node 3 is reassigned to point to node 7. Graph (c) shows the synchronized server state. In this example, the offloaded server execution assigns a new instance, node 8 to node 3, modifies nodes 3 and 5, and assigns the `null` value to node 6, with Graph (d) depicting the results. The mutated state is then transferred to the client and synchronized with its state depicted in Graph (e). Specifically, node 6 is removed, node 3 is reassigned to point to node 8, and nodes 3 and 5 are overwritten with new values. Graph (f) shows the client state after the synchronization.

Handling Failure during Offloading

To handle network outages, the inserted offloading functionality is surrounded by a `try-catch` block that catches and handles network-related exceptions. They are handled by restarting the local computation from the latest checkpoint, thus aborting the current offloading operation without perturbing the application's execution. In other words, network volatility disables the intended optimization of cloud offloading, but it does not render the application unusable, as is the case with the cloud offloading mechanisms that lack such a checkpoint-and-restart functionality.

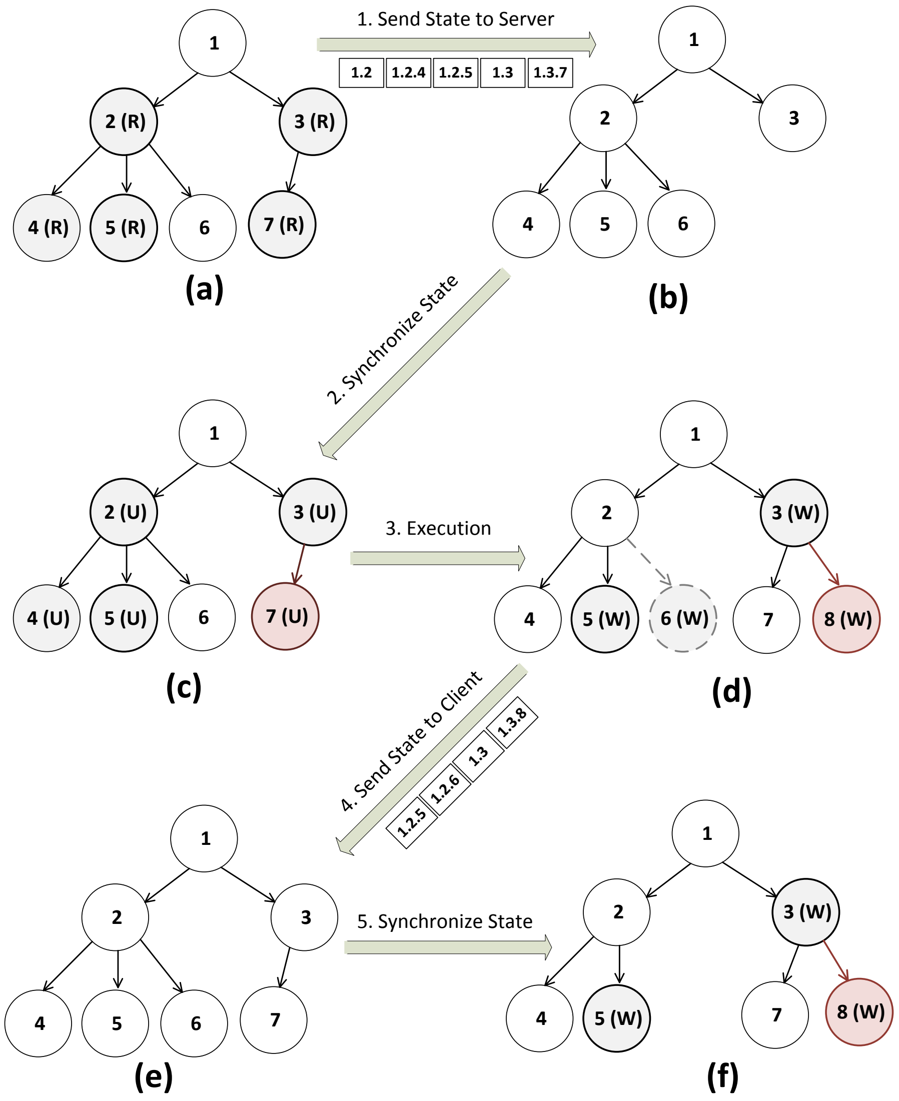


Figure 5.10: Procedure to synchronize program state.

5.4 Evaluation

We have evaluated the effectiveness of our approach in reducing energy consumption and improving performance by applying it to four third-party mobile applications running on the Android platform. We have also evaluated our approach through a series of micro benchmarks and a larger case study. The evaluation has shown that our overall approach can effectively reduce the amount of consumed energy for well-engineered applications, with the introduced program transformations and runtime execution never causing the enhanced applications to exceed their original levels of energy consumption. The results also show that our runtime system can efficiently collect runtime parameters and predict advantageous offloading scenarios without imposing unreasonable performance and energy overheads.

5.4.1 Micro Benchmark

The purpose of this micro benchmark is to understand the overhead imposed by the runtime system, whose responsibilities include monitoring the relevant fluctuations in the environment, estimating potential energy savings due to the possible offloading steps, and synchronizing heaps during the offloading.

Experimental Setup

The experimental setup includes a Motorola Droid (600MHz CPU, 256MB RAM, 802.11g) (a low-end mobile device) and a Samsung Galaxy III (1.5GHz dual-core CPU, 2GB RAM, 802.11n) (a high-end mobile device), and desktop (3.0GHz quad-core CPU, 8GB RAM) as the offloaded server. The mobile device has communicated with the server through a WiFi. For the WiFi connection, we have experimented with two emulated network conditions that have the following respective round trip time (RTT) and bandwidth characteristics: 20ms and 50Mbps, typical for a WiFi network and 50ms and 1Mbps, typical for a mobile network. Table 5.1 shows the energy profiles, which are used to parameterize the runtime system.

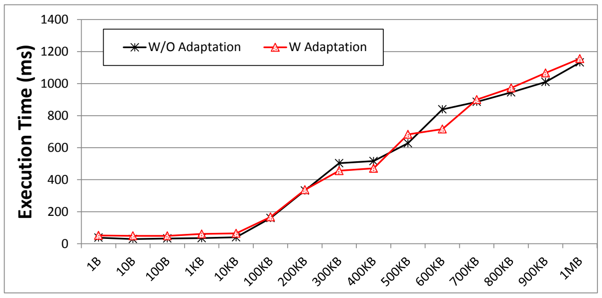
Overhead

In this benchmark, we compared the total execution time and energy consumption of the baseline versions of the benchmarks with that of in the presence of the adaptive runtime system. The

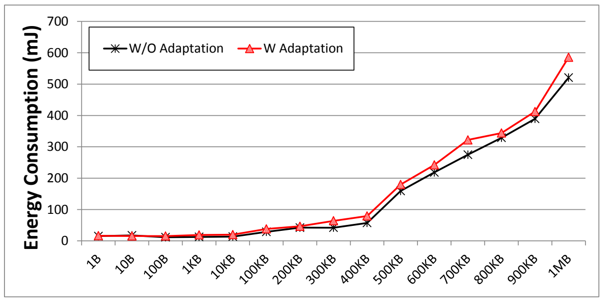
Table 5.1: Manufacturer provided energy profiles

CPU	1512.0MHz: 577mA	**WiFi**	96mA
	1209.6MHz: 408mA		0.3mA
	907.2MHz: 249mA	**Mobile**	250mA
	604.8MHz: 148mA		3.4mA
	302.4MHz: 55mA		

first graph of Figure 5.11 shows execution time and energy consumption. As one can see, both overheads are quite insignificant. In particular, the performance overhead never exceeds 100ms and remains constant for all the measured data transfer sizes, and the energy consumption overhead never exceeds 50mJ, which is insignificant as compared to a typical total energy budget.



(a) Performance overhead.



(b) Energy consumption overhead.

Figure 5.11: Overhead comparison.

Energy Consumption Estimation

Then, we evaluated how accurately the runtime system can predict how much energy will be consumed in a given time interval. Figure 5.12 compares the energy consumption predicted by the runtime system and that estimated by our model based on the actual resource usage. The average error is 10.6% and standard deviation is 21.3%. When considering only 90% data removing outliers, the average error is 8.5% and standard deviation is 6.8%. These results indicate that the runtime system can predict the future energy consumption accurately, with the discrepancies averaging 6-7%.

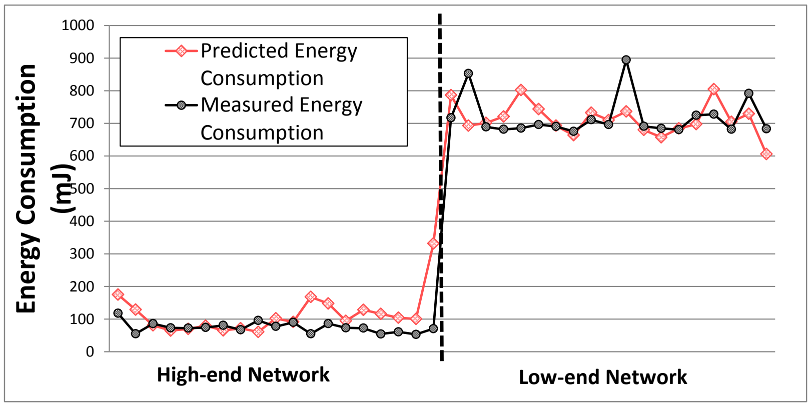


Figure 5.12: Energy consumption estimation.

5.4.2 Case Study

To ensure that our approach is applicable to real-world mobile applications, we chose experimental subjects from the list of open source projects hosted by Google Code and SourceForge. In the following discussion, we discuss how our approach improved the energy efficiency of real-world mobile applications.

Subject Applications

First, we analyzed mobile application domains to identify those ones that can benefit from cloud offloading. Based on this analysis, we found the following domains being most applicable for this optimization:

- Games containing artificial intelligence engines (e.g., Chess, Sudoku, Go, etc.)
- Algorithms involving heavy computation on large data sets (e.g., graph search, shortest path, backtracking, etc.)
- Image processing (e.g., optical character recognition, face detection, object recognition, etc.)
- Security operations (e.g., taint analysis, virus scans, etc.)

To show that our approach is generally applicable, we selected four computation intensive applications from different domains: (1) Pocket chess[1] is a mobile chess game, whose AI engine contained in `SimpleEngine.go()` is offloaded. (2) The N-queens problem solver offloads an algorithm contained in `Solver.enumerate()`. (3) Mezzofanti[2]—used as our motivating example—offloads its OCR functionality, contained in `OCR.ImgOCRAndFilter()`. (4) JJIL [3] detects faces from a picture and offloads its face recognition functionality, contained in `DetectHaarParam.push()`.

Experimental Results

In the following discussion, we show how our approach proved effective in saving the energy consumed by and reducing the execution time of subject applications. First, we applied our approach to the Pocket Chess application. For this benchmark, we created a scenario of randomly selecting and moving a chess piece (i.e., emulating the actions of a human player) and the AI engine computing and making its own move. Figure 5.13 shows the amount of energy consumed by the high-end mobile device connected to two different WiFi networks. The optimized version of the subject application consumed less energy than its original version. It offloads to the cloud the heavy CPU processing required to calculate the next move, and transfer back only the new position for the piece to move. As the game proceeds, the optimized version exhibits a constant rate of energy consumption, while the original version consumes an increasing amount of energy as the required AI processing intensifies. Based on these results, one may benefit from applying our approach to any game application (e.g., Sudoku, Go, etc.) that comes with a computationally intensive AI engine.

Then, we applied our approach to the N-Queens problem solver and measured the amount of energy consumed by two different mobile devices. Figure 5.14 show the amount of energy consumed. In particular, the original application consumes more energy when executing on the low-end mobile device than executing on the high-end mobile device, while the optimized version executing

[1]`http://code.google.com/p/pocket-chess-for-android/`
[2]`http://code.google.com/p/mezzofanti/`
[3]`http://code.google.com/p/jjil/`

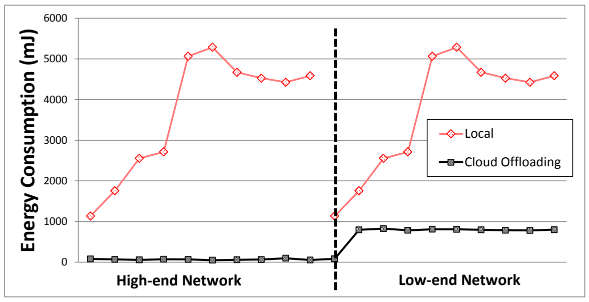


Figure 5.13: Chess application.

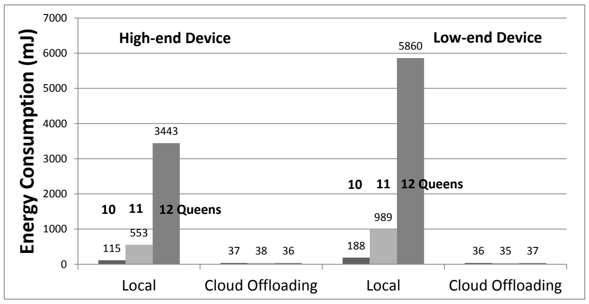
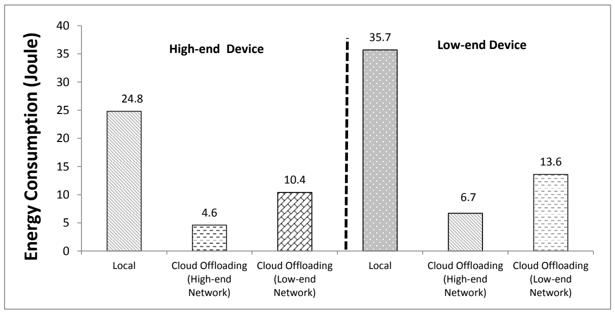


Figure 5.14: N-Queens solver.

on both mobile devices only consumes 0.6% - 50% of the amount of energy consumed by the original application. Furthermore, as the number of queens increases, the original version consumes an increasing amount of energy, while the optimized version consumes energy at a constant rate. These experiments indicate that cloud offloading can be benificial for such computationally intensive applications.

To evaluate the adaptive capabilities of our approach, we evaluated its effectiveness in optimizing the image processing applications (Mezzofanti and JJIL), running on two different mobile devices in two different network conditions. For Mezzofanti, the benchmark measured the energy consumed and time taken by examining one image file containing 200 characters; for JJIL, the benchmark measured the same parameters when examining a single image file for the presence of human faces. The network conditions (delay/bandwidth) were emulated for the high-end network as 20ms/50Mbps (favorable conditions) and for the low-end network as 50ms/1Mbps (poor conditions). For Mezzofanti, Figure 5.15's left graph shows the amount of energy consumed by the original and optimized versions, while Figure 5.15's right graph shows the total execution time. For the majority of measured setups, the optimized version consumed less energy than their original local version, while also outperforming the original local version in terms of total execution time. However, when executed on the high-end device connected to the low-end network, the optimized version took more time than the local version, while still consuming less energy.

For JJIL, we evaluated our approach's adaptive capabilities by studying the impact of changes in network conditions on the energy savings and execution time improvements afforded by offloading.



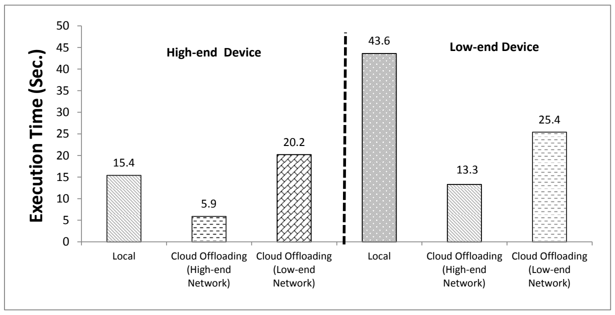


Figure 5.15: Energy consumption and execution time of Mezzofanti (OCR application).

To that end, we compared the respective effectiveness of the plain and adaptive offloading mechanisms. The plain mechanism offloads methods marked with `@OffloadCandidate` on every invocation, without considering runtime conditions; the adaptive mechanism determines whether such methods should be offloaded based on the current runtime conditions.

The network conditions (delay/bandwidth) were emulated for the high-end network as 20ms/50Mbps (favorable conditions) and for the low-end network as 50ms/1Mbps (poor conditions). As shown in Figure 5.16, for the favorable conditions, both plain and adaptive offloading mechanisms performed equally effectively on both high- and low-end devices. However, for the poor conditions, the plain offloading mechanism running on the high-end device consumed more energy and took longer than the original version.

For the poor conditions, the adaptive mechanism on the high-end device chose not to offload, executing locally. Finally, for the poor conditions, the adaptive mechanism on the low-end device consumed less energy, but took longer to run than did the original local version. Our optimization approach is amenable to additional steering from the programmer. For example, the programmer can specify if the runtime should favor execution performance over energy efficiency. To that end, our approach features a simple configuration mechanism described in Section 5.3.

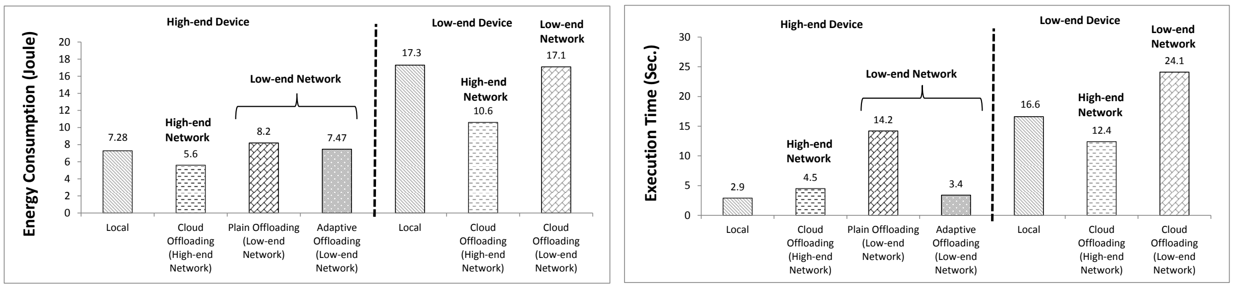


Figure 5.16: Energy consumption and execution time of JJIL (face recognition application).

5.5 Discussion

In this section, we discuss advantages and limitations of our approach.

5.5.1 Advantages

Our approach works with the standard, unmodified hardware/software stack; it employs bytecode engineering to transform programs and a lightweight runtime system to dynamically steer and adapt offloading operations. Our approach makes it possible to keep the maintained version of the mobile application's source code intact, as only the bytecode version is transformed. Cloud offloading requires a minimal programming effort, limited to marking methods as energy hotspots. Our approach makes offloading decisions at runtime by monitoring the execution environment, thus discovering optimal offloading strategies. Finally, the offloading transformations do not preclude the mobile application from executing locally in the case of the network becoming disconnected.

5.5.2 Limitations

Despite its benefits, our approach is not applicable to all applications. Some mobile applications are written in a monolithic style, in which functionality cross-cuts through traditional modularization program constructs such as classes and methods. Without clear offloading program points, our approach, which operates at the method boundary, would be inapplicable. Another limitation of our approach is that we do not take multiple concurrent threads into consideration when determining whether a method can benefit from offloading. If an offloaded method can be simultaneously invoked by multiple concurrent threads, our approach currently does not ensure that the program's state remains consistent. However, extending our program analysis heuristics to work with multiple threads is a matter of engineering. Similarly, our runtime can be easily enhanced to become thread-aware. We plan to investigate how our approach can support concurrency as a future work direction.

5.6 Related Work

In the following discussion, we compare our approach with distributed mobile execution, multiple representative offloading mechanisms, and then discuss approaches to optimizing energy efficiency.

5.6.1 Distributed Mobile Execution

The approach presented in this article adopts many of the techniques above to automatically transform mobile applications without any changes to their source code and to synchronize program states between partitions. The approaches that can distribute a program to run across the network include automated partitioning, replication, and migration. Automated program partitioning uses a compiler-based tool to introduce distribution to a centralized program [155]. To that end, the tool changes the application's structure (e.g., introducing proxies) and inserts middleware calls. As an alternative to partitioning, a centralized program can be replicated on remote nodes, with the replicas' states synchronized according to a given consistency protocol [65, 72]. The advantage of replication is that the application's structure does not need to change, but synchronizing replicas may cause high network traffic. Finally, migration leverages mobile computing to move execution between remote nodes. Unlike replication, migration moves around a single copy of the executed application's image. Because efficient migration requires runtime system support, a customized runtime environment must be provided for each participating node. Applications, however, can migrate without any changes to their source code [127, 23].

5.6.2 Cloud Offloading

Spectra [43] monitors resource usage and availability to determine whether the mobile application's energy consumption can be optimized through cloud offloading. To that end, Spectra requires that the programmer manually partition the application to create a proxy for calling remote functions. By contrast, our approach does not introduce remote proxies and modifies the program automatically through bytecode engineering.

Slingshot [137] leverages state replication, so that the replicas can be deployed on remote servers to increase performance. Although Slingshot shares similarity with our approach by relying on synchronizing distributed state, our approach does not require any changes to the runtime system and is portable across any JVMs. Furthermore, while Slingshot optimizes the offloading efficiency by locating the closest surrogate server, our approach relies on program analysis to reduce the size of the program's state that needs to be transferred across the network.

CloneCloud [20] leverages thread-level offloading to optimize mobile execution. Cloudlet [127] migrates the VM. These approaches require a custom runtime system. By contrast, our approach does not change the runtime system but rewrites the program instead to introduce fault-tolreant offloading. As a result, our approach works with standard systems and runtime environments and is easily portable across platforms.

MAUI [26] offloads resource-intensive functionality to remote servers through program partitioning. While MAUI, similarly to our approach, relies on the programmer annotating the source code to which methods to offload, XRay [112] partitions applications automatically.

ThinkAir [75] offloads energy intensive methods to the cloud, so that the resulting cloud-based execution can be scaled up by running the offloaded methods in parallel on dynamically allocated

VMs. COMET [52] offloads computation at the thread level by means of a distributed transaction memory and VM synchronization techniques.

5.6.3 Optimizing Energy Consumption

In addition to cloud offloading mechanisms, reducing the energy consumed by mobile applications has been the focus of multiple complimentary research efforts at the level of operating systems (e.g., energy-efficient CPU scheduling [169], disk power management [163], network protocols [168]). While much research has focused on system-level solutions, programming-level solutions (e.g., energy saving algorithms [103], design patterns for energy efficient computing [89], software models for energy efficient software [151], and programming languages for energy efficiency [24]) have also been advocated in the literature.

While most of these efforts have focused on one particular system layer (i.e., mainly the network), cross-layer approaches control energy consumption by leveraging the information provided by multiple system layers [44, 99]. We see our approach as lying on the intersection of system- and program-level solutions. Our system architecture and programming model enable the creation of powerful system-level cloud offloading optimization by providing convenient software abstractions exposed as a library.

5.7 Conclusions

In this section, we presented adaptive cloud offloading, a technique for optimizing the energy consumption of mobile applications. To maximize efficiency, adaptive offloaded determines the functionality to offload at runtime, via automated program transformation as well as efficient run-time monitoring and adaptation. Most offloading schemes fail to consider the energy required for network transfer, which can sometimes negate any energy savings gained from offloading. Because mobile network conditions vary, the decision whether or not offloading will save energy must occur dynamically at runtime and requires an immediate cost-benefit analysis to determine whether the energy saved by reducing local processing exceeds the overhead required for distributed processing. This analysis is far from trivial due to mobile hardware heterogeneity and execution environment volatility. The presented approach reduces the energy consumed by mobile applications without changing their source code. We have evaluated our approach by reducing the energy consumed by micro benchmarks and third-party applications in different execution environments. These results indicate that our approach represents a promising direction in optimizing the energy efficiency of mobile applications.

Chapter 6

Configurable and Adaptive Middleware for Energy-Efficient Mobile Computing

Energy efficiency is rapidly becoming a key software design consideration [103], as mobile devices are steadily replacing desktop computers as the dominant computing platform. The increasingly feature-rich nature of mobile applications renders battery capacities a key limiting factor in the design of mobile applications [113]. To reduce the energy consumed by modern mobile applications, system designers must consider all the constituent parts of a distributed mobile execution. Although communication middleware has become an essential component of modern mobile software, existing communication mainstream middleware mechanisms were designed with the primary focus of facilitating distributed communication and improving performance rather than on reducing energy consumption.

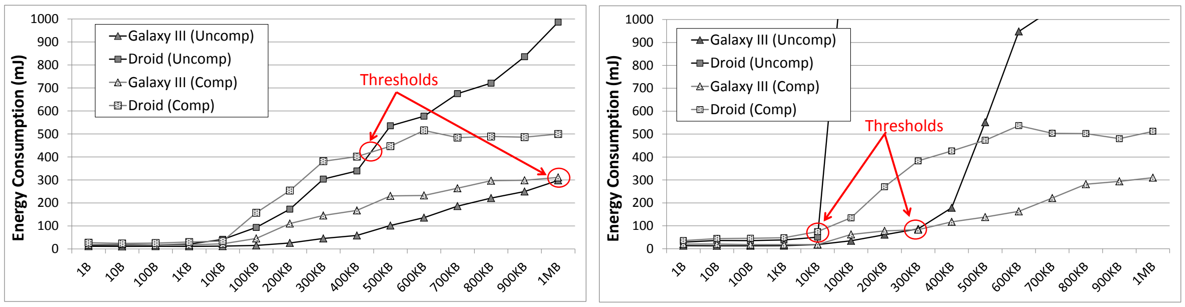
Network communication commonly constitutes one of the largest sources of energy consumption in mobile applications. According to a recent study, network communication consumes between 10% and 50% of the total energy budget of a typical mobile application [111]. Many mobile applications are designed with the assumption that they will be operated over some mobile networks with a certain fixed bandwidth/latency ratio. However, this assumption will not hold if an application is operated across a variety of mobile networks (WiFi, 3G, and 4G), whose conditions (e.g., bandwidth, delay, packet loss, etc) often fluctuate continuously. As an example, during the same execution, the application can be accessing a remote service using either the 3G network (low bandwidth, long delay) or the WiFi (high bandwidth, short delay). Furthermore, the conditions of either network can be fluctuating at runtime. Networks and their conditions can significantly affect how much energy is consumed by a mobile application.

Since communication middleware defines the patterns through which a distributed application transmits data across the network, the choice of middleware can heavily impact the amount of energy consumed by mobile applications. However, the execution patterns in mainstream middleware mechanisms are fixed; they cannot be flexibly adapted to reduce energy consumption when mobile applications switch between mobile networks with different conditions [96]. Furthermore, to maximize energy savings, the middleware execution patterns must be individually tailored for specific applications, so as to take into consideration their application logic. To support that kind of customization, middleware must be equipped with appropriate programming abstractions that can express how energy optimization strategies should be triggered and parameterized.

In this research, we introduce a new communication middleware architecture, which equips mobile application developers with pragmatic tools and methodologies to engineer energy-efficient distributed mobile software. Our middleware architecture employs dynamic, adaptive optimization as a mechanism to minimize the amount of energy consumed by mobile applications to perform distributed interactions. We call our novel middleware mechanism e-ADAM (**e**nergy-**A**ware **D**ynamic **A**daptive **M**iddleware). e-ADAM enables the programmer to express a rich set of middleware energy optimizations and the runtime conditions under which these optimizations should be applied. The e-ADAM runtime system then dynamically applies the expressed optimizations as specified for the network conditions in place.

For the thoughtful system designer, e-ADAM opens up new energy optimization opportunities at the cost of slightly increasing the programming effort: specialized optimization strategies are crafted for individual runtime conditions. However, the e-ADAM continuous dynamic adaptation makes it possible to reach the middleware energy efficiency levels that cannot be achieved via automatic optimizations performed outside of the programmer's purview.

Our experiments have demonstrated the effectiveness of the e-ADAM approach to reduce the amount of energy consumed by a set of benchmarks and third-party Android applications executed across volatile mobile networks. By presenting e-ADAM, this research makes the following technical contributions:



(a) Mobile network. (b) WiFi network.

Figure 6.1: Energy consumption comparison showing different thresholds.

- **A communication middleware architecture that enables dynamic, application-specific energy consumption optimization:** e-ADAM features configurable energy optimization that effectively addresses the execution volatility common in modern mobile networks.

- **Application-specific energy consumption estimation:** e-ADAM features an application-level energy model that enables the e-ADAM runtime system to accurately measure and predict the energy consumption levels experienced by mobile applications under fluctuating runtime conditions.

- **Systematic evaluation:** e-ADAM optimized the amount of energy consumed by a set of benchmarks and third-party mobile applications, with the resulting energy savings as high as 30% in some cases.

The rest of this chapter is structured as follows. Section 6.1 defines the problem that our approach aims at solving and introduces the concepts and technologies used in this work. Section 6.2 details our technical approach. Section 6.3 discusses how we evaluated our approach. Section 6.5 compares our approach to the related state of the art. Section 6.6 concludes and presents future work directions.

6.1 Problem and Technical Background

In this section, we outline the problem that our approach is intended to solve and the major technologies it uses.

6.1.1 Problems and Technical Challenges

Dissimilar networks are known to consume different amounts of energy to transmit the same data [79]. Consequently, the amount of energy consumed on a network transmission can be minimized by employing the communication patterns tailored for a given network. In other words, communication adaptation in response to changes of network conditions can reduce the overall energy consumption.

To elaborate on our prior results, we measured the amount of energy that can be saved by applying the common energy optimization technique of data compression. In this experiment, we used TCP sockets to transfer simple data buffers between a mobile device and a remote server. Figure 6.1 shows the impact of compressing the transferred data on the mobile device's energy consumption for the WiFi and two typical cellular networks. We experimented with two mobile devices that differed vastly in their respective hardware setups (i.e., Motorola Droid (low-end) and Samsung Galaxy S3 (high-end)) to ensure that the observed energy consumption differences are due to the network transmission rather than any other execution parameters.

When executing over the WiFi network using a high-end device, data compression does not seem to affect the amount of consumed energy. When executing over the WiFi network using a low-end device, data compression does not affect the amount of consumed energy until the 400kB data transfer threshold has been reached. Starting from that threshold, data compression ends up reducing the overall energy consumption. When executing over the 3G network and 4G network using either low-end or high-end device, data compression does not affect the amount of consumed energy until the 10kB and 300kB data transfer thresholds have been reached, respectively. Starting from that thresholds up, data compression ends up reducing the overall energy consumption.

The specific thresholds, devices, and network types used in this experiment are immaterial and only prove the point that compression must be applied in a device- and network-specific fashion, so as to maximize the potential energy savings. Because the network environment and device in place determine the thresholds at which compression should be engaged to reduce energy consumption, middleware should be able to turn this and other optimizations on and off at runtime as needed. This experiment demonstrates the need for adaptivity in middleware to be able to transfer data using the communication and execution patterns that match the execution environment in place.

At the same time, the middleware adaptations should be sufficiently general to benefit users using a variety of mobile devices. For example, Facebook reports that the mobile version of their application is accessed by 2,500 varieties of mobile devices [40]. Each of these devices is likely to exhibit different energy consumption patterns due to the hardware differences of the devices. Since it would be unrealistic to statically specify adaptations for each mobile device and application, we designed our approach to rely on runtime monitoring that can trigger the available adaptations as required by a given execution environment.

In this section, we present a middleware architecture that realizes the vision outlined above as energy-aware dynamic, adaptive middleware (e-ADAM). Enabling effective runtime adaptations with the goal of saving energy requires innovation in programming abstraction expressiveness and sophisticated runtime support. Specifically, our approach enables the programmer to implement multiple strategies for the same middleware functionality, each of which is deployed as dictated by the runtime changes in the execution environment. At execution time, e-ADAM monitors runtime network conditions and then automatically selects an appropriate energy optimization strategy provided by the programmer. Furthermore, in response to the changes of runtime network conditions, e-ADAM dynamically switches between the provided strategies.

6.1.2 Technical Background

Our middleware architecture combines dynamic adaptation and runtime energy consumption monitoring.

Middleware for Distributed Execution

Our middleware architecture uses features from mainstream middleware mechanisms for distributed execution as building blocks. To facilitate effective reuse, we classify existing middleware architectures on two main axes: level of abstraction and network communication footprint. In terms of the level of abstraction, there are socket-, remote procedure call-, message-, and service-based platforms. In terms of the network communication footprints, they transfer data across the network in binary and text (primarily XML)-based formats. Major, widely used middleware architectures include sockets, Message Oriented Middleware (MOM), remote method invocation (RMI), and Web services. Our middleware architecture uses a proxy-based distributed execution mechanism and encodes the transferred data in binary.

Transport Layer Energy Saving Provisions

The IEEE 802.11 standard codifies a power saving mode (PSM), under which the wireless network interface enters the sleep mode when idle. Other approaches have leveraged this mode to save energy. For example, reference [140] describes one such energy saving strategy that takes advantage of the prior knowledge of the application's communication patterns. This strategy employs a bandwidth throttling mechanism, implemented via a custom network protocol stack, to control the network transmission rate. Thus, adjusting application communication patterns can lengthen the wireless network interface's sleep time, thereby saving energy. This strategy has been shown effective in media streaming or large data transfer applications. The goal of our approach is to achieve similar energy saving benefits, but without modifying the standard protocol stack. By operating at the application level, our approach adapts crude-grained communication patterns, providing comparable energy saving benefits. For example, application communication patterns can be adapted to be periodic and predictable by breaking down large transmitted data into blocks or by reshaping the TCP traffic into bursts.

Dynamic Middleware Adaptation

Dynamic middleware adaptations change the execution strategies at runtime to optimize the overall execution by leveraging the information provided by various system components. Dynamic adaptation has been also used to optimize energy consumption [44, 99]. The Odyssey platform [44] adapts data or computational quality to save energy consumption, so as not to exceed the available system resources. DYNAMO [99] is an another middleware platform that adapts energy optimizations across various system layers, including applications, middleware, OS, network, and hardware, to optimize both performance and energy. These energy-aware adaptations identify possible trade-offs between energy consumption and quality of service and then choose optimal energy optimizations based on runtime conditions.

e-ADAM shares the same vision with these approaches. However, as compared to these approaches, our approach aims at providing a high degree of customizability. It provides a programming model that enables programmers to implement application-specific energy optimization strategies as well as to express how these strategies should be applied at runtime.

Energy Consumption Measurement

To optimize energy consumption, first one must be able to accurately measure how much and how an application consumes energy. That is why in recent years several research efforts have focused on creating effective approaches to measuring energy consumption in mobile applications. Three primary approaches have been described in the literature: at the architecture, network, and application levels.

An example of an architecture level energy measuring approach is PowerPack [48], which physically connects to hardware resources (e.g., CPU, disk, memory, and mother board component) and then maps the measured energy consumption patterns to the application's source code, making it possible to analyze energy consumption both at the hardware and source code levels. An example of a network level energy measuring approach is described in reference [8], which measures energy consumption of the general network activity for 3G, GSM, and WiFi networks. Examples of an application level measurement approach are described in reference [54]. Another application level approach [129] puts forward a model that divides the total energy consumed by an application into the *functions* of computation, communication, and infrastructure (e.g, JVM garbage collection, implicit OS routines, etc.). The focus of this work is on middleware, whose energy consumption is measured by an application level measurement methodology.

6.2 Energy Aware Adaptive Middleware (e-ADAM)

Next, we present e-ADAM by giving an overview of the approach and describing its major parts.

6.2.1 Approach Overview

The e-ADAM approach hinges on the concept of configurable energy optimization strategies. e-ADAM provides a Java API for implementing the strategies, whose triggering and operation is specified using simple key-value configuration files (for an example, see Figure 6.4). By continuously monitoring the execution environment, the e-ADAM runtime system dynamically loads and applies the strategies as specified in the provided configurations. By selecting the strategies to apply at runtime in accordance with the environment in place, e-ADAM can optimize energy efficiency more effectively than static approaches.

Figure 6.2 presents the architectural design of e-ADAM that comprises three main components: a strategy manager, a runtime monitor, and an adaptation manager. First, the *policy handler* parses configuration files and maps the parsed parameters to the available strategy implementations. Sec-

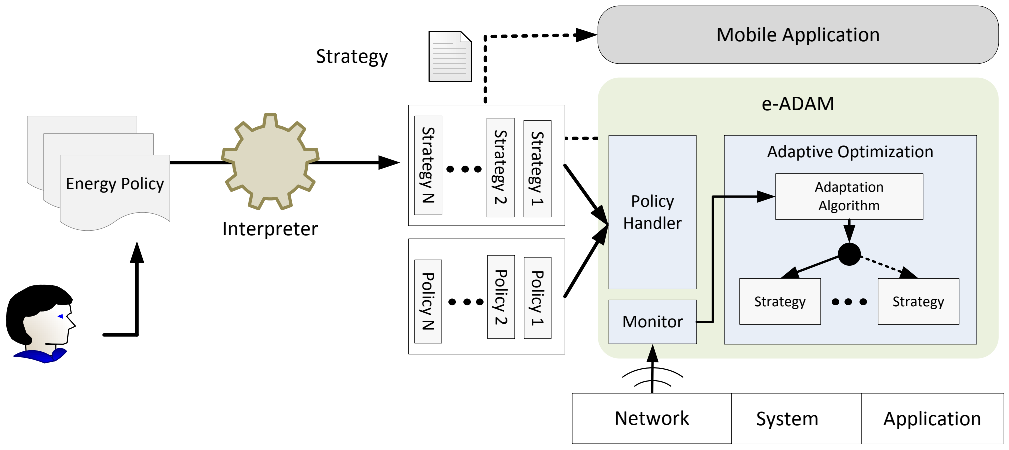


Figure 6.2: e-ADAM component diagram.

ond, the *monitor* module continuously collects runtime information that includes network and hardware characteristics (e.g., delay, network connection type, CPU frequency, etc.) by leveraging the Android system monitoring API. Third, the *adaptive optimization* module correlates the collected execution data with the configuration parameters. If the resulting correlation indicates that a different energy optimization strategy should be triggered, the *adaptive optimization* module dynamically locates, loads, and executes the triggered strategy.

6.2.2 *e-ADAM* Process Flow

Having described the individual components of e-ADAM, we now explain how they interact with each other. The e-ADAM process flow in Figure 6.3 comprises three main processes: (1) energy consumption prediction, (2) communication monitoring, and (3) distributed communication.

The *energy consumption prediction* process estimates future energy consumption levels and communication latencies to select the specified energy optimization strategy. To that end, the adaptation manager requests snapshots of the current and prior execution environment (e.g., CPU, delay, transferred data size, execution time, etc.) from the runtime monitor and the execution history

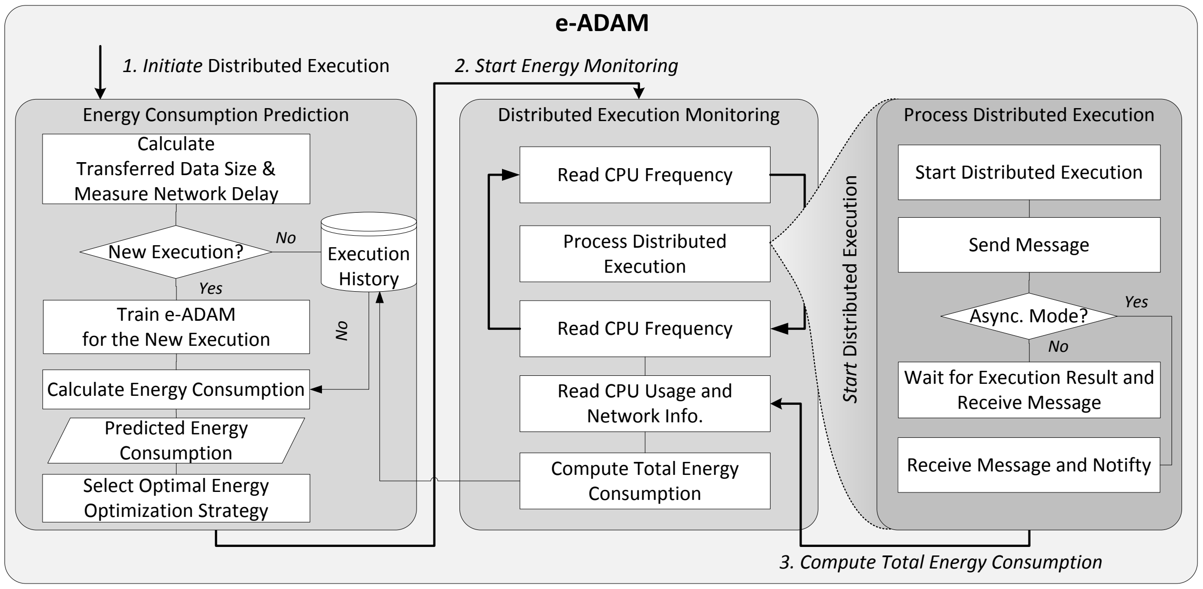

e-ADAM
1. Initiate Distributed Execution
2. Start Energy Monitoring
Energy Consumption Prediction
Calculate Transferred Data Size & Measure Network Delay
New Execution?
No
Yes
Execution History
No
Train e-ADAM for the New Execution
Calculate Energy Consumption
Predicted Energy Consumption
Select Optimal Energy Optimization Strategy
Distributed Execution Monitoring
Read CPU Frequency
Process Distributed Execution
Read CPU Frequency
Read CPU Usage and Network Info.
Compute Total Energy Consumption
Start Distributed Execution
Process Distributed Execution
Start Distributed Execution
Send Message
Async. Mode?
Yes
No
Wait for Execution Result and Receive Message
Receive Message and Notifty
3. Compute Total Energy Consumption

Figure 6.3: e-ADAM process diagram.

(cache), respectively. Based on these parameters, the adaptation manager estimates the energy and latency to be incurred by a given communication operation and applies the energy optimization strategy as guided by the configuration in place.

The *distributed communication monitoring* process continuously collects runtime data and creates an execution history cache to consult when estimating the energy consumption and latency of future communication operations. This process dispatches remote operations in accordance with the applied energy optimization strategy. Next, we describe each process of e-ADAM in detail.

6.2.3 *e-ADAM* Configuration

As shown in Figure 6.4-(a), the e-ADAM energy optimization configuration files follow a simple key-value format, thus making it straightforward for programmers to compose and understand. Each set of configuration settings is identified by a unique name followed by a collection of key-value pairs. Configurations are demarcated by an empty line.

The three configuration keys are **execution**, **strategy**, and **criteria**. The **execution** pair identifies a remote server by means of a URL or an IP address. The **strategy** pair identifies the adaptation strategies to be applied for this communication. If a configuration file has multiple strategies, the runtime adaptation module then makes use of the selection **criteria** values. Recall that the runtime continuously applies policies speculatively, so as to evaluate their effectiveness.

The **criteria** pair defines which notion of *effectiveness* should be used with a given configuration. The **criteria** value of **energy** indicates the effectiveness to reduce energy consumption, while that of **performance** to speed up performance. The value of **epr** indicates the effectiveness to increase the energy/performance ratio. The value of **never** disables the configuration from being applied, while the value of **always** applies the configuration irrespective of its effectiveness.

```
configuration = [name]
  execution = [remote API]@[address]
  strategy = [name] ((and|or) [name])*
  criteria = (energy|performance|epr|never|always)
```

(b) Configuration file format.

```
public enum Pointcut {Before, After, Around;}

public class Invocation {
  public Method getMethod() {...}
  public Object[] getParams() {...}
  public Object getResult() {...}

  public void setParams(Object[] params) {...}
  public void setResult(Object result) {...}

  public Object proceed(int blocking) {...}
}

public class [name] extends Strategy {
  public Pointcut getPointcut() {...}
  public Object invoke(Invocation invocation) {...}
}
```

(b) Strategy API class.

Figure 6.4: Energy optimization configurations.

Adaptation strategies are implemented by extending class `Strategy`, which provides a single method `invoke`. To enable the programmer to control at which execution point a strategy should be applied, e-ADAM features an Aspect Oriented Programming [71] abstraction to specify whether the implemented strategy is to be invoked `before`, `after` or `around` (instead of) a given remote communication.

The components implementing the strategies referenced in configuration files follow the Java naming convention, in which class names are prefixed with their full package names (e.g., `edu.vt.eadam.Compression`). The e-ADAM runtime calls method `invoke(...)` at the specified pointcut when both the specified remote API is invoked and the energy optimization strategy is activated. A typical adaptation strategy makes use of common energy optimizations, including data compression, reducing image quality, and redirecting to an easier-to-reach remote server as discussed in Section 6.2.3.

Energy Consumption Estimation

The energy consumption estimation module predicts how much energy will be consumed by a given remote communication by computing the workload expected to be carried out by the communication. Specifically, e-ADAM only computes the energy consumed by the CPU and network communication as follows:

$$\begin{aligned}\boldsymbol{E} &= E_{cpu} + E_{net} = (C_{cpu} \times T_{cpu} + C_{net} \times T_{net}) \times V \\ &= \{\Sigma(C^{act}_{cpu_f} \times T^{(u+s)}_{cpu}) + (C^{act}_{net} \times T^{act}_{net}) + (C^{idle}_{net} \times T^{idle}_{net})\} \times V\end{aligned}$$

where $C^{act}_{cpu_f}$ is the electric current of the CPU at a particular clock speed. Modern CPUs feature speed-step, a facility that allows the clock speed of a processor to be dynamically changed by the operating system, with different levels of power consumed at each clock speed. T^{u}_{cpu} and T^{s}_{cpu} are user and system times taken by the distributed execution, and they are obtained by consulting the statistics provided by the operating system (e.g., `/proc/[pid]/stat`). V is current voltage, which is also obtained from the operating system (`/sys/class/..../voltage_now`). C^{act}_{net} and C^{idle}_{net} are the electric current of the network processor required during the active and idle phases, respectively. T^{act}_{net} and T^{idle}_{net} are the active and idle runtime periods during the remote communication, respectively. These device- and execution-specific values are cached to compute the predicted amount of energy to be consumed during future remote communications.

Training-Based Energy Consumption Prediction

To predict the amount of energy that is likely to be consumed during remote communications, e-ADAM correlates the device- and execution-specific values that were previously measured and cached. These measured values are cached and used for predicting the future energy consumption and execution time. Then, using the cached prior execution parameters (e.g., delay, communication time, transferred data size, total execution time, etc.) and the current measured execution parameters, e-ADAM predicts the communication latency. During the initialization phase, e-ADAM bootstraps the training process by executing all the strategies specified in the input configuration file and persists the obtained results to permanent storage. Based on the estimated communication latencies, as observed from prior executions, the e-ADAM runtime system predicts the expected energy consumption for a given remote communication as follows:

$$\boldsymbol{E_{prd}} = \{E_{cpu}^{avg} + (C_{net}^{act} \times T_{net}^{prd\_act}) + (C_{net}^{idle} \times T_{net}^{prd\_idle})\} \times V$$

where E_{prd} is the predicted future energy consumption, E_{cpu}^{avg} is the average energy consumption of the given remote communication, and $T_{net}^{prd\_act}$ is the predicted communication time, which are computed by using the transferred data size and delay, respectively. To avoid a delay spike, the current delay value is then recomputed by weighting the most recently obtained value (i.e., $delay = delay \times \alpha + delay \times (1 - \alpha)$). α was set to 0.3 in our reference implementation). The computed energy consumption value is used as a parameter for selecting the optimal energy optimization strategy for a given scenario.

Figure 6.5 shows how we measure the delay and each time taken during the idling, sending and receiving phases. These times are averaged in a given measurement window to estimate the expected communication time as depicted in Fig 6.6. The estimates are cached to be used to predict the communication time and execution time.

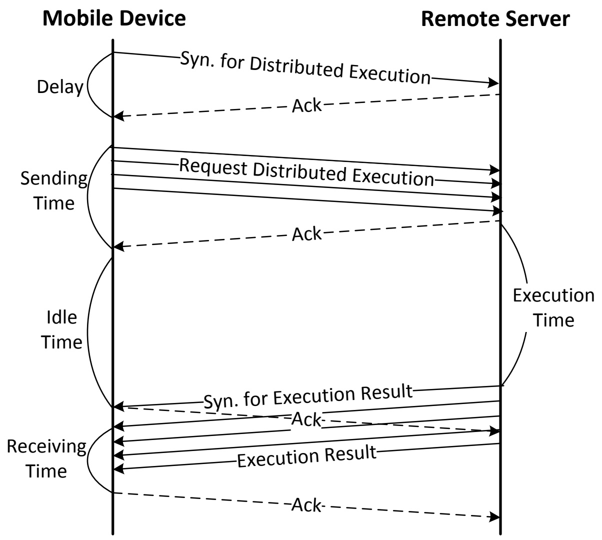


Figure 6.5: Measuring communication time.

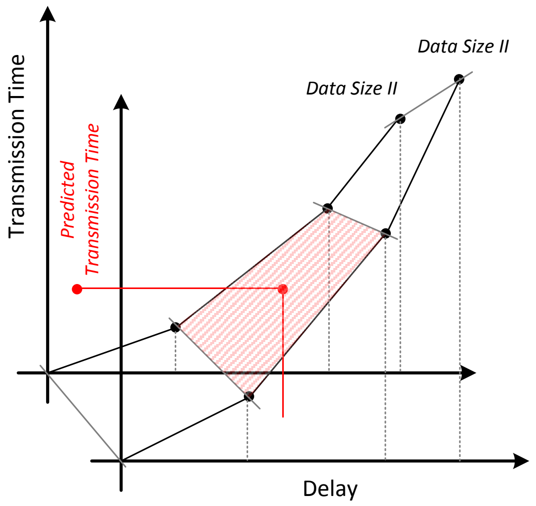


Figure 6.6: Estimating communication time.

Energy Optimization Strategy Selection

Figure 6.7 shows the procedure for selecting the provided energy optimization strategies at run-time. To select an appropriate energy optimization strategy, the adaptation module predicts the future energy consumption and the future execution time by analyzing the collected runtime execution values and cached prior executions. Then, the module selects the optimization strategy that would yield either the lowest expected energy consumption, or the shortest expected execution time, or the highest expected energy/performance, as specified by given selection criteria—`energy`, `performance` `epr`, etc. While the first two parameters are self-explanatory, `epr` (the energy/performance ratio) is a parameter that we have formulated in our prior research [79]. This ratio correlates performance and energy consumption values so as to maximize the resulting correlation. The energy/performance ratio is computed as follows:

$$\boldsymbol{EPR(x)} = \frac{MIN(T_{prd}(1),...,T_{prd}(n))/T_{prd}(x)}{E_{prd}(x)/MIN(E_{prd}(1),...,E_{prd}(n))} \times 100$$

```
/** Find an optimal strategy */
FOREACH $strategy \in \forall Strategies$ DO
  CASE Energy
    $E_{exptd} \leftarrow$ estimateEnergy(..., $strategy$)
    IF $E_{exptd}$ is the smallest THEN
      $bestStrategy \leftarrow strategy$ END IF
  CASE Performance
    $T_{exptd} \leftarrow$ estimateExecTime(..., $strategy$)
    IF $T_{exptd}$ is the smallest THEN
      $bestStrategy \leftarrow strategy$ END IF
  CASE EPR
    $E_{exptd} \leftarrow$ estimateEnergy(..., $strategy$)
    $T_{exptd} \leftarrow$ estimateExecTime(..., $strategy$)
    $epr \leftarrow$ getEPR($E_{exptd}, T_{expted}$)
    IF $epr$ is the smallest THEN
      $bestStrategy \leftarrow strategy$ END IF
END FOREACH

/** Redirect the curent execution path to the strategy */
$bestStrategy$.invoke(...);

/** Receive result or exception and update execution history */
CASE Succeed
  $result \leftarrow$ executionCompleted()
CASE Fail
  $result \leftarrow$ executionFailed()
update($result$)
```

Figure 6.7: The procedure to select an optimal strategy.

where, T_{prd} and E_{prd} are the expected execution time and energy consumption values, respectively. A higher EPR represents a condition under which the energy optimization strategy in place consumes less energy while retaining high performance as compared to other strategies.

Energy Optimization Strategies

Recall that the observation that underlies the design of e-ADAM is that certain pieces of middleware functionality can be implemented differently. In the following discussion, we give specific examples of middleware functionality and the alternatives for their implementations.

Data compression: Data compression will reduce network transfer, but will be more computationally intensive, thus requiring additional CPU processing. Transmitting raw data will increase network transfer, but will require less CPU processing. Which of the strategies will consume less energy depends on the runtime conditions in place.

Redirection: Another optimization is redirection. This strategy iterates through different end-points of a distributed execution in the case of experiencing poor network conditions. For instance, when experiencing a network congestion at `a.com/foo`, `alt.a.com/foo` can be invoked instead.

Batching: A common middleware optimization is batching multiple distributed communications into a single bulk communication. For modern networks, whose bandwidth improvements surpass that of their latencies, transmitting data in bulk can reduce the aggregate latency. However, the degree of batching should be determined by the network conditions in place.

In addition to the aforementioned general optimization strategies, one can also apply application-specific optimizations, tailored for particular application scenarios. For example, in a video conferencing application, the QoS can be traded for energy efficiency when the battery level gets below a certain threshold.

6.3 Evaluation

We have evaluated the effectiveness of e-ADAM by applying it to benchmarks and third-party applications.

6.3.1 Micro-Benchmarks

In this micro-benchmark, we compared the performance and energy consumption characteristics of XML-RPC and e-ADAM in executing a collection of remote invocations with different parameter sizes. In this benchmark, the client executes empty server methods with different parameters. This strategy isolates the energy consumed by the underlying middleware mechanism.

Experimental Setup

The experimental setup includes a Motorola Droid (600 MHz CPU, 256 MB RAM, 802.11g, 3G) (a low-end mobile device), a Samsung Galaxy III (1.5 GHz 2-core CPU, 2 GB RAM, 802.11n, 4G) (a high-end mobile device), and a Dell PC (3.0 GHz 4-core CPU, 8 GB RAM) (the remote server). The network types are WiFi, 3G network, and 4G. For the WiFi, the following two network conditions were emulated: high-end (2ms round trip time and 50Mbps bandwidth) and medium-end (50ms and 1Mbps). The Droid used a 3G network (70ms and 500Kbps), while the Galaxy III used a 4G network (70ms and 1Mbps). We used the same energy profiles described in Table 5.1 of Chapter 5.4.

Benchmark I: Performance and Energy Consumption Comparison

In this benchmark, we compared the total execution time and energy consumption of two middleware mechanisms (XML-RPC and sockets) with e-ADAM. Figure 6.8 shows the total execution time and the energy consumed by each middleware mechanism on the high-end device, respectively. As expected, XML-RPC has the poorest performance and consumes the largest amount of energy to perform the same functionality. The socket implementation shows the highest performance and the smallest energy consumption. The e-ADAM energy consumption and performance are close to those of the socket mechanism. XML-RPC performed closely to the other mechanisms until the transferred data's volume reaches 1kB. Thus, the benefits of e-ADAM are particularly pronounced when in the presence of volatile networks and large fluctuating data volumes.

Benchmark II: Performance and Energy Consumption Overhead

In this benchmark, we compared the total execution time and energy consumption of the baseline versions of the benchmarks with that using an adaptation strategy. Figure 6.9 (top) shows the total execution time measured on each device. As one can see, the performance overhead is quite insignificant. In particular, the overhead for both devices never exceeds 100ms and remains constant

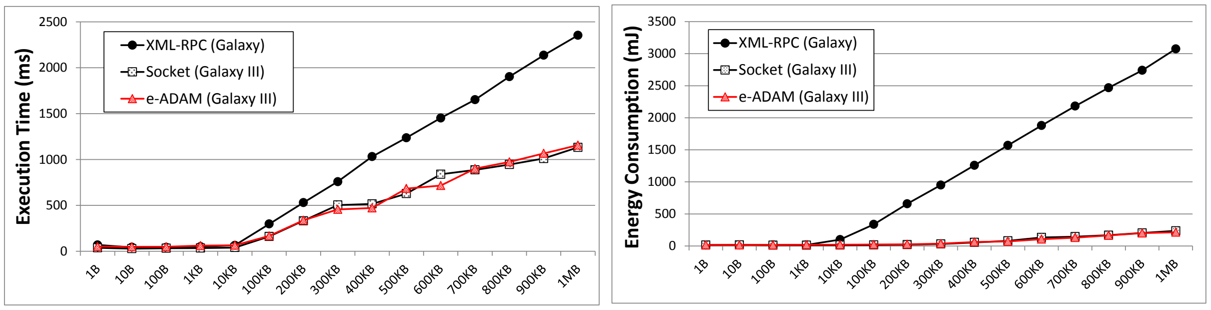


Figure 6.8: Performance and energy consumption.

for all the measured data transfer sizes. Figure 6.9 (bottom) shows the amount of energy consumed by each device. As expected, the high-end device (Samsung Galaxy) consumes less energy than the low-end device (Motorola Droid). In particular, the overhead for both devices never exceeds 50mJ, which is insignificant as compared to a typical total energy budget.

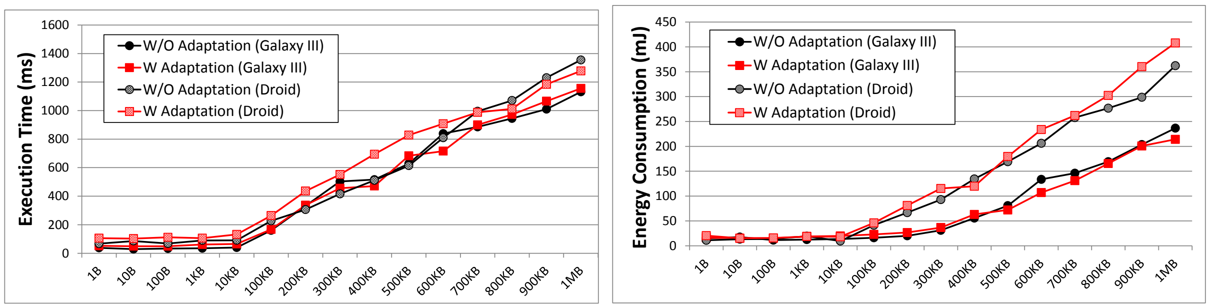


Figure 6.9: Performance and energy consumption overhead.

Benchmark III: Adapting Energy Optimizations

In this benchmark, we evaluated how the runtime system can adapt its middleware functionality between no optimization and a compression optimization in response to changes in network conditions on the high-end device. First, we evaluated how accurately the runtime system can predict how much energy will be consumed when using two different optimization strategy on the high-end device. Figure 6.10 shows both the predicted and the consumed energy by e-ADAM with no optimization vs. a compression optimization. In this benchmark, the average error was 23.09

% and standard deviation was 10.74 %, which is higher than in other benchmarks, whose error rates are 6-7 %). This is because when an application consumes a small amount of energy, small changes in the execution environment, such as delay or CPU frequency, can significantly affect the predicted energy consumption. (e.g., when the transferred data size increases, the average error decreases.).

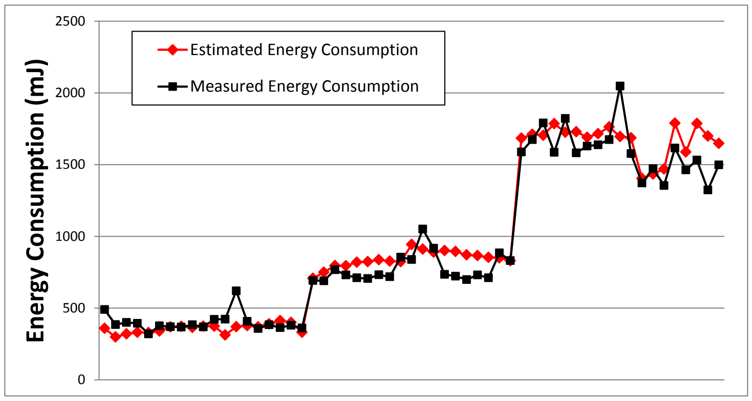


Figure 6.10: Energy consumption prediction.

Then, we evaluated the effectiveness of the e-ADAM runtime system in selecting the energy optimization strategies that would be optimal for different execution environments. As an optimization strategy we chose data compression, which trades CPU processing for network transfer. Compressing the data reduces its size, thus reducing the workload of the remote operation transferring the data. However, running the compression algorithm uses up additional CPU cycles. First, we measured the actual amount of energy used by the same remote operation, with and without the compression strategy applied. To obtain statistically relevant measurements, each pair of remote operations (compressed and uncompressed) was repeated a 100 times under the 3 simulated networks whose parameters are explained above. After we measured the concrete amount of energy consumed under compression and without compression, we queried the e-ADAM runtime system whether it would trigger the compression optimization strategy. Furthermore, to evaluate the impact of the training process, we compared the effectiveness of the untrained and trained states of the runtime system (for 10 consecutive execution cycles). Table 6.1 shows the evaluation criteria for this experiment.

Table 6.1: The evaluation criteria.

Compression causes →	**Less Energy**	**More Energy**
Trigger compression	Success	Failure
Not Trigger compression	Failure	Success

Table 6.2 shows the failure rates for each network type and data size. As expected, when transferring small data volumes, compression creates some noise. Because the runtime system uses a moving average to estimate future energy consumption, it continuously reacts to the relevant changes in the execution environment. Because the runtime does not respond instantaneously, e-ADAM does not suffer from the noise that can arise due to sudden fluctuations, such as a delay spike. However, in the cases of low energy consumption (e.g. the test case with network type I and 10kB consumes only 15-100mJ.), frequent fluctuations make noise unavoidable, thus increasing the failure rate of the runtime system. However, the programmer can configure e-ADAM not to engage any optimizations when the average energy consumption level is already low. In all other test cases, nevertheless, the e-ADAM runtime system showed itself quite effective, with the training decreasing the failure rate across the board.

Table 6.2: Failure rate when triggering the opt. strategy.

Data size	**Network I** No Training/Training	**Network II** No Training/Training	**Network III** No Training/Training
10 kB	18 % / 12 %	9 % / 7 %	3 % / 2 %
100 kB	7 % / 2 %	3 % / 0 %	1 % / 0 %
1000 kB	3 % / 0 %	1 % / 0 %	0 % / 0 %

6.3.2 Case Study

To determine how well our approach works with real-world mobile applications, we experimented with open source projects, used as experimental subjects in our prior research on cloud offloading [78]. JJIL[1] is a face recognition application; its recognition functionality executes remotely in class `DetectHaarParam`. OSMAndroid[2] is a navigation application; its shortest path calcula-

[1]`http://code.google.com/p/jjil/`

[2]`https://code.google.com/p/osm-android`

```
configuration = JJIL
  execution = DetectHaarParam.push(*)@[*:*]
  strategy = edu.vt.eadam.Compression
  criteria = energy

configuration = OSMAndroid
  execution = ShortestPath.execute(*)@[cs.vt.edu:*]
  strategy = edu.vt.eadam.Redirection;
  criteria = epr

configuration = Mezzofanti
  execution = OCR.ImgOCRAndFilter(*)@[cs.vt.edu:9999]
  strategy = edu.vt.eadam.Batching and edu.vt.eadam.Compression
  criteria = energy
```

Figure 6.11: Configuration file for the case study.

tion functionality executes remotely in class `ShortestPathAlgorithm`. Mezzofanti[3] is a text recognition application; its OCR functionality executes remotely in class `OCR`.

For each subject, we measured the amount of the energy consumed and the execution time by typical, simple use cases. Specifically, for OSMAndroid, we selected two locations and the requested route between them. For the face recognition application, we examined one image file for the presence of human faces. Then, we selected the compression strategy for the face recognition application because it transfers a large amount of data; we selected the redirection strategy for OSMAndroid. The use cases were executed under two optimization modes: (1) original distributed execution without an energy optimization and (2) the e-ADAM approach with either the `epr` or `energy` criteria. Figure 6.11 shows the configurations used in this case study, and Figure 6.12 shows code snippet of the compression strategy.

Figure 6.13's upper graph shows how the e-ADAM approach has reduced the amount of energy consumed by the face recognition application. Because in a high-end mobile network (i.e., Network I) the compression strategy incurs additional processing overhead, e-ADAM does not apply this strategy. However, in other networks (i.e., Network II and III), the compression strategy reduced the amount of energy consumed by 30%. Figure 6.13's lower graph shows the total exe-

[3]`https://code.google.com/p/mezzofanti`

```
public class Compression extends Strategy {
  public Pointcut getPointcut() { return Pointcut.Around; }

  public Object invoke(Invocation invocation) {
    Object[] params = invocation.getParams();
    /** compress parameters */
    loop(parameters) {
      ByteArrayOutputStream baos = ... ;
      GZIPOutputStream gzipOut = ... ;
      ObjectOutputStream objectOut = ... ;

      objectOut.writeObject(params[i]);
      params[i] = baos.toByteArray();
    }
    point.setParams(params)

    /** Carry out the remote invocation , blocking for result */
    Object result = invocation.proceed(Proxy.MODE_BLOCK);

    return result ;
  }
}
```

Figure 6.12: Compression strategy implementation.

cution time taken by the face recognition application. Similarly, e-ADAM improved the overall performance.

For the OSMAndroid application, we used a different scenario. Because the application transfers less data than the first subject application, we selected a redirection strategy. Figure 6.14 shows the total execution time and total execution time for the subject application. At the first phase, two remote servers (i.e., Server I and II) have the same execution environments (e.g., network condition), but at the second phase, we injected network delay to both remote servers and injected 500 ms processing delay at the Server I. With the `epr` criteria, while e-ADAM selects Server I during the first phase, it selects Server II during the second phase, as it considers both energy consumption and performance metrics when selecting an optimal strategy.

For the OCR application, we used two strategies—`Batching` and `Compression`—to optimize the transfer of the fragments of a large (∼6MB) image file. The strategies are applied sequentially in the order of appearance in the configurations, `Batching` followed by `Compression`. Figure

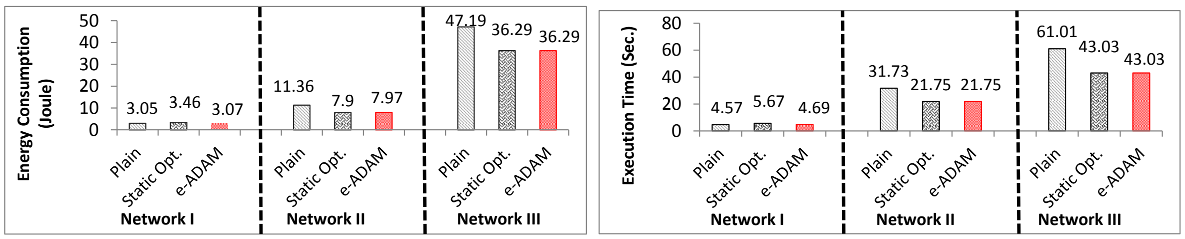


Figure 6.13: Experimental results of the JJIL app.

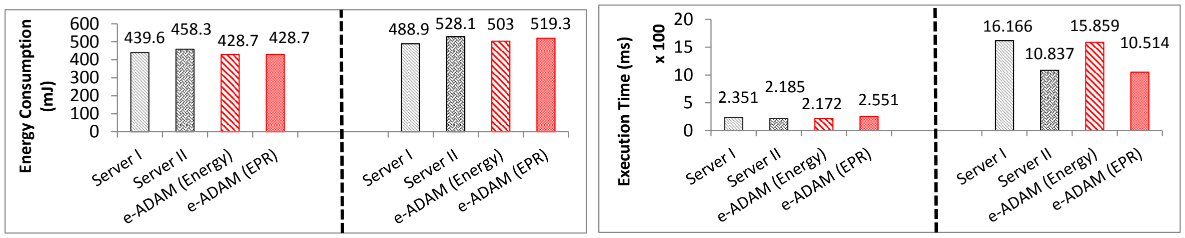


Figure 6.14: Experimental results of the OSMAndroid app.

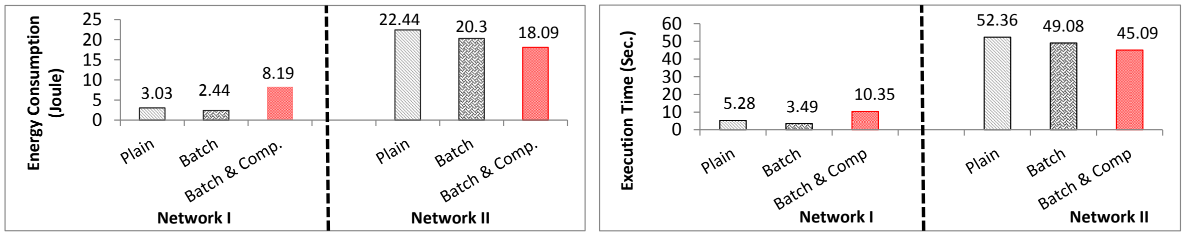


Figure 6.15: Experimental results of the Mezzofanti app.

6.15 shows the results of e-ADAM applying these strategies: first, `Batching` alone and then combined with `Compression`. In a high-end mobile network (i.e., Network I) compressing data incurs additional processing overhead, whose energy costs are not offset by the resulting reductions in bandwidth utilization. Thus, for these networks, the `Batching` strategy should be the only one applied. However, in limited networks, adding the `Compression` strategy causes the overall energy consumption to be reduced by 20%. Figure 6.15's lower graph shows the total execution time and total execution time for the same OCR application. Energy consumption and total execution time are positively correlated. Indeed, e-ADAM reduced the total runtime by 6% and 14%, when the `Batching` and the (`Batching` + `Compression`) strategies were applied, respectively. Furthermore, reusing the `Compression` strategy has reduced the implementaiton burden of this case study.

6.3.3 Threats to Validity

The results presented above are subject to internal and external validity threats. The internal validity is threatened by how the interactive subject applications were exercised. The performance and energy consumption of interaction applications depend on how the user chooses to use them. To minimize this threat, the benchmarked use cases were fixed to using the same media (i.e., picture file) and location (i.e., GPS coordinates). Another internal validity threat is the fashion in which we implemented the benchmarked optimization strategies (e.g., compression and redirection).

The external validity is threatened by the accuracy of our energy and performance models. Since relying on models always provides approximated values, the question is how accurate our energy and performance profiles are. To minimize this threat, we adopted commonly used models parameterized with the energy profiles provided by the manufactures of the devices used in the evaluation.

6.4 Discussion

The e-ADAM approach provides the generality, separation of concerns, and reusability advantages. e-ADAM is general in that it can be applied to a variety of distributed mobile applications. By leveraging a proxy-based implementation, e-ADAM can serve as a drop-in replacement for mainstream mechanisms structured around the RPC paradigm, both synchronous and asynchronous. Mimicking asynchronous communication may require writing additional glue code to emulate the original order of response messages. e-ADAM enables a greater separation of concerns in that it can change a mobile applications's energy/performance characteristics without affecting its core business logic. The energy optimization strategies and the configurations to apply them are expressed separately from the main source code. This degree of separation also makes it possible to effectively reuse energy optimization strategies and configurations across components and applications.

Although e-ADAM can deliver tangible benefits to the mobile application programmer, it also has some inherent limitations. In particular, the limitations concern its ranges of applicability and usability. The overhead imposed by the e-ADAM runtime makes the approach inapplicable to those distributed mobile applications that use simple, infrequent remote interactions. The runtime overhead is offset if the optimized application spends a substantial amount of energy on remote interactions. Thus, application designers have to decide whether using e-ADAM would be beneficial for each application. Another limitation of e-ADAM is that the approach is automated rather than automatic. The programmer is responsible for implementing energy optimization strategies using the provided API and for expressing how the strategies should be applied. Although implementing common optimization strategies is facilitated by the presence of multiple third-party libraries and frameworks, the programmer must be aware of which predefined building blocks they have at their disposal.

6.5 Related Work

Reducing the energy consumption of mobile applications to extend the battery life of mobile devices has been the focus of multiple complimentary research efforts, including system- and application-level optimizations. The system-level optimizations include CPU scheduling algorithms [169], disk power managements [163], network interfaces [4], specialized-network protocols [8], and process migration [20]. Although these system-level optimizations have proven quite effective in extending the battery lives of mobile devices, the system changes these optimization require complicate their deployment to heterogeneous mobile devices.

In contrast to system-level optimizations, application-level optimizations provide pragmatic, automatic tools or guidelines to the programmer [78, 103, 50]. The effectiveness of application-level optimizations hinges on the accuracy of the information provided by the underlying system and execution environments.

Cross layer optimizations leverage the information provided by multiple system layers. The Odyssey platform orchestrates the interactions between the OS and applications [44]. Similarly to our approach, Spectra [43] provides a specialized APIs for the mobile programmer. By monitoring multiple execution environments, Spectra selects an optimal communication path to a remote server. While Spectra only provides a single fixed optimization, e-ADAM enables the programmer to implement multiple application-specific optimizations. The e-ADAM approach makes it possible to reuse known energy optimization techniques to design application-specific energy optimizations.

6.6 Conclusions

In this research, we introduced e-ADAM, a novel communication middleware architecture that employs dynamic adaptation to reduce the energy consumption of mobile applications. e-ADAM features a sophisticated runtime system that predicts and regulates the energy consumed during remote interactions. By means of configuration files, the runtime can deploy programmer-provided optimization strategies. Our evaluation comprised applying e-ADAM to reduce the energy consumed by benchmarks and third-party applications under different execution environments. These results indicate that the e-ADAM approach represents a promising direction in developing energy efficient mobile applications.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This dissertation have explored novel solutions to the problem of improving the quality of service of mobile applications by leveraging advanced software engineering techniques (e.g., program analysis, automated program transformation, distributed programming abstractions, etc.). Also, we discussed how a combination of novel software system architectures, automated software tools, and empirically derived programming guidelines can assist the programmer in developing and optimizing modern software, especially in the area of distributed and mobile computing. The focus on distributed/mobile applications and state-of-the-art software technologies increases the potential impact on real-world software systems and development practices. Next, we present major contributions of this dissertation research and future work plans.

7.1.1 Summary of Contributions

The research presented in this dissertation was published in the proceedings of Middleware'09 [82], SCC'10 [84], MESOCA'11 [77], ICDCS'12 [78], and ICSM'13 [80] and in the journals of Service Oriented Computing and Applications [83], Information and Software Technology [79], IEEE Computer [152], and Automated Software Engineering [81]. Major contributions of this research include:

1. **Automated program transformations for the transitioning to distributed applications** [81]: A set of refactoring techniques facilitate the process of transforming centralized ap-

plications to use remote services. These techniques automate the program transformations required to render portions of functionality of a centralized applications as remote services and re-target the application to access the remote functionality.

2. **Hardening distributed applications with resiliency against partial failures** [82, 77]: A declarative approach for hardening distributed applications with resiliency against partial failures—we introduce a domain-specific language for describing both failures and the hardening strategies to eliminate them.

3. **Energy-efficient distributed execution through offloading** [78, 80, 152]: An effective energy consumption optimization approach efficiently and fault-tolerantly synchronizes execution state between the mobile device and remote server to provide energy-efficient and reliable mobile execution.

4. **Middleware with dynamic adaptation capabilities:** Energy-aware adaptive middleware enables the programmer to express how to dynamically adapt middleware execution patterns in the presence of volatile mobile networks, so as to reduce the mobile application's energy consumption.

5. **Systematic assessment of distributed applications across middleware** [84, 83, 79]: A novel mechanism can accurately assess the performance, conciseness, complexity, reliability, and energy consumption of distributed applications across middleware for accessing remote functionality.

7.2 Future Work

In the near future, we will continue the work on improving the energy efficiency of mobile applications by leveraging advanced software technologies. In the longer term, we will look at the issue of energy consumption holistically, considering all the constituent components of a distributed system. Finally, our long term vision is to apply the lessons learned improving the QoS of mobile applications to orchestrating and adapting big data workflows.

7.2.1 Adapting Cloud Offloading via Constraint Solving

Offloading a mobile application's functionality via a remote server has become an important energy and performance optimization technique. Adaptive offloading determines the functionality to offload and the offloading server at runtime. Mobile applications, executed over networks with dissimilar latency/bandwidth characteristics, access cloud-based servers that offer different levels of performance, availability, and trust. An effective adaptive offloading mechanism must consider all these factors when determining which functionality should be offloaded to which server. Implementing an adaptive offloading mechanism driven by both runtime conditions and user preferences is non-trivial.

By directly following upon this dissertation research, we will explore how mobile applications can be optimized for energy and performance efficiency by using constraint solving to dynamically adapt cloud offloading. The main idea is to express the optimization priorities of cloud offloading as *a constraint satisfaction problem (CSP)*. A CSP computes the values that a set of variables must take in order to satisfy a set of conditions imposed on the variables. One can map variables to offloading optimization criteria (e.g., energy savings, performance efficiency, server availability, server trustworthiness, etc.); values to the actual runtime parameters of the criteria (e.g., the amount energy consumed by a method, the time taken to execute a method, the average failure rate for an offload server, and the user-specified degree of trust for a server); conditions to the end user's specified optimization priorities (e.g., minimize energy consumption, minimize execution time, maximize a given energy/performance ratio, prefer offload servers with higher trust levels). By expressing cloud offloading in terms of constraint solving that drives an adaptive runtime system, this research direction can provide an expressive and efficient solution to the problem of adaptively leveraging cloud computing resources to optimize mobile applications.

7.2.2 Holistic Energy Optimization for Distributed Mobile Applications

Mobile computing is characterized by a high heterogeneity of the hardware/software stack and network environments. A mobile application is executed on devices with dissimilar CPU, memory, and network card capacities [79]. Server execution environments also exhibit high heterogeneity, with job scheduling policies impacting server energy consumption. Mobile networks differ widely in their respective latencies, bandwidths, congestion, packet loss, and interference. Looking at energy optimization holistically requires considering the mobile device, the offloading server, and the network connecting the two.

Because using remote, cloud-based resources can impose additional costs on the user, new energy optimizations are needed to be able to offload mobile functionality while being mindful of the overall server workload. A holistic energy optimization will coordinate the executions at both the mobile device and the server. To realize this level of coordination would require innovation in the middleware space. Equipped with the new middleware mechanism, system designers will be able to engineer highly energy efficient mobile applications that not only save the mobile device's battery power, but also use cloud-based offloading servers energy efficiently.

7.2.3 Workflow Based Automated Big Data Analytics

As huge amounts of data are being continuously produced, data scientists without CS expertise need intuitive but powerful tools to effectively analyze the available data. Although there are numerous data analytics tools, a typical data analytics procedure requires that multiple tools be combined into a workflow. For example, a big data workload may need to be preprocessed, passed to a data mining routine, visualized, etc, while running these tools on different systems such as a data center, analytics servers, and large displays. Moreover, each data analytics procedure should be tailored for specific data or user scenarios. As a result, orchestrating all these tools into coherent workflows is non-trivial.

Our vision is to explore how automated code generation and middleware can help data scientists in integrating analytics tools into data analytics workflows. To that end, data scientists must be equipped with powerful expression media, with domain specific languages offering an attractive possibility; generating connectors automatically will seamlessly combine tools with dissimilar input/output formats into a workflow; advanced middleware mechanisms will drive the workflows' execution. Easily constructing data analytics workflows will not only improve the productivity of data scientists, but will democratize the access to big data analytics for the average user.

Bibliography

[1] CCA-Forum. `http://www.cca-forum.org/`.

[2] Net:Netem. `http://www.linuxfoundation.org/en/Net:Netem/`.

[3] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language support for connector abstractions. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'03)*, July 2003.

[4] M. Anand, E. Nightingale, and J. Flinn. Self-tuning wireless network power management. *Wireless Networks*, 11(4):451–469, 2005.

[5] B. Andreopoulos, A. An, V. Tzerpos, and X. Wang. Clustering large software systems at multiple layers. *Inf. Softw. Technol.*, 49(3):244–254, 2007.

[6] P. Andritsos and V. Tzerpos. Information-theoretic software clustering. *IEEE Trans. Softw. Eng.*, 31(2):150–165, 2005.

[7] A. Arora and S. Kulkarni. Detectors and correctors: a theory of fault-tolerance components. In *The 1998 18 th International Conference on Distributed Computing Systems*, pages 436–443, 1998.

[8] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proceedings of the* 9^{th} *ACM SIGCOMM Conference on Internet Measurement Conference*, 2009.

[9] G. Banavar, T. Chandra, R. Strom, and D. Sturman. A case for message oriented middleware. In *Proceedings of the* 13^{th} *International Symposium on Distributed Computing*, pages 1–18. Springer-Verlag London, UK, 1999.

[10] L. Baresi and S. Guinea. Towards dynamic monitoring of WS-BPEL processes. In B. Benatallah, F. Casati, and P. Traverso, editors, *Service-Oriented Computing - ICSOC 2005*, volume 3826 of *Lecture Notes in Computer Science*, pages 269–282, 2005.

[11] L. Baresi, S. Guinea, M. Pistore, and M. Trainotti. Dynamo + Astro: An integrated approach for BPEL monitoring. volume 0, pages 230–237, Los Alamitos, CA, 2009.

[12] K. Becker, A. Lopes, D. S. Milojicic, J. Pruyne, and S. Singhal. Automatically determining compatibility of evolving services. In *ICWS '08: Proceedings of the 2008 IEEE International Conference on Web Services*, pages 161–168, Washington, DC, USA, 2008. IEEE Computer Society.

[13] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, B. A. Rapp, and D. L. Wheeler. Genbank. *Nucleic Acids Res.*, 30:17–20, 2002.

[14] N. Brown and C. Kindel. Distributed Component Object Model Protocol–DCOM/1.0, 1998. Redmond, WA, 1996.

[15] R. Buyya, C. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5^{th} utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.

[16] G. Candea and A. Fox. Recursive restartability: turning the reboot sledgehammer into a scalpel. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 125–130, May 2001.

[17] G. Canfora, A. Fasolino, G. Frattolillo, and P. Tramontana. A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. *Journal of Systems and Software*, 81(4):463–480, 2008.

[18] N. Carr. *The big switch: Rewiring the world, from Edison to Google*. WW Norton & Company, 2008.

[19] F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 569–588, New York, NY, USA, 2007. ACM.

[20] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: elastic execution between mobile device and cloud. In *Proceedings of the 6th ACM European Conference on Computer Systems (EuroSys)*, 2011.

[21] M. H. Chunlei, C. Wang, and L. Zhang. Toward a reusable and generic security aspect library. In *AOSD Technology for Application-level Security*, 2004.

[22] Cisco Market Trends. Cisco service provider: Wi-Fi: Offload mobile data and create new services, 2012.

[23] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, 2005.

[24] M. Cohen, H. S. Zhu, S. E. Emgin, and Y. D. Liu. Energy types. In *Proceedings of the ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Oct 2012.

[25] S. Colbert. Speech at the White House Correspondents' Association dinner. Transcript, April 26 2006.

[26] E. Cuervo, A. Balasubramanian, D.-K. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2010.

[27] M. Dahlin, B. Chandra, L. Gao, A. Khoja, A. Nayate, A. Razzaq, and A. Sewani. Using mobile extensions to support disconnected services. Technical Report CS-TR-00-20, University of Texas at Austin, 2000.

[28] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. A language for specifying security and management policies for distributed systems. *Imperial College Research Report DoC*, 1, 2000.

[29] C. Damm, P. Eugster, and R. Guerraoui. Linguistic support for distributed programming abstractions. In *Proceedings of the* 24^{th} *International Conference on Distributed Computing Systems*, 2004.

[30] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The cost of doing science on the cloud: the montage example. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[31] C. Demarey, G. Harbonnier, R. Rouvoy, and P. Merle. Benchmarking the round-trip latency of various Java-based middleware platforms. *Studia Informatica Universalis Regular Issue*, 4(1):7–24, 2005.

[32] V. Dialani, S. Miles, L. Moreau, D. De Roure, and M. Luck. Transparent fault tolerance for web services based architectures. *Euro-Par 2002 Parallel Processing*, pages 107–201, 2002.

[33] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 java benchmark. In *Proceedings of the 13*th *European Conference on Object-Oriented Programming*, 1999.

[34] T. Do, S. Rawshdeh, and W. Shi. pTop: A process-level power profiling tool. In *Proceedings of the* 2^{nd} *Workshop on Power Aware Computing and Systems (HotPower'09)*, 2009.

[35] Eclipse Equinox. `http://www.eclipse.org/equinox/`.

[36] J. Edstrom and E. Tilevich. Reusable and extensible fault tolerance for restful applications. In *Proceedings of the 11*th *IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2012.

[37] R. Elfwing, U. Paulsson, and L. Lundberg. Performance of soap in web service environment compared to corba. *Asia-Pacific Software Engineering Conference*, 0:84, 2002.

[38] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[39] J. Erlichman. Special report: Cloud computing. *Government Computer News*, 2009.

[40] Facebook Mobile. Facebook for every phone, July 2011.

[41] C. Fang, D. Liang, F. Lin, and C. Lin. Fault tolerant web services. *Journal of Systems Architecture*, 53(1):21–38, 2007.

[42] B. Ferris, K. Watkins, and A. Borning. OneBusAway: results from providing real-time arrival information for public transit. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems (CHI '10)*, 2010.

[43] J. Flinn, S. Park, and M. Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, 2002.

[44] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. *ACM SIGOPS Operating Systems Review*, 33(5):48–63, 1999.

[45] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[46] C. L. Fullmer and J. Garcia-Luna-Aceves. Solutions to hidden terminal problems in wireless networks. In *Proceedings ACM SIGCOMM*, pages 39–49, 1997.

[47] Gartner, Inc. Gartner highlights key predictions for IT organizations and users in 2010 and beyond, Jan. 2010.

[48] R. Ge, X. Feng, S. Song, H. Chang, D. Li, and K. Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):658–671, 2010.

[49] J. Gibbons and S. Ruth. Municipal Wi-Fi: big wave or wipeout. *Internet Computing, IEEE*, 10(61):107–125, 2006.

[50] I. Giurgiu, O. Riva, and G. Alonso. Dynamic software deployment from clouds to mobile devices. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, 2012.

[51] A. Gokhale and D. C. Schmidt. Measuring the performance of communication middleware on high-speed networks. In *Proceedings on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '96)*, 1996.

[52] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. COMET: code offload by migrating execution transparently. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, volume 12, pages 93–106, 2012.

[53] S. Guinea, L. Baresi, G. Spanoudakis, and O. Nano. Comprehensive monitoring of bpel processes. *IEEE Internet Computing*, 99, 2009.

[54] A. Gupta and P. Mohapatra. Energy consumption and conservation in wifi based phones: A measurement-based study. In *Proceedings of the* 4^{th} *Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, 2007.

[55] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, 2009.

[56] B. Haumacher, T. Moschny, and M. Philippsen. The JavaParty project. `www.ipd.uka.de/JavaParty`, 2007.

[57] B. Hayes. Cloud computing. *Commun. ACM*, 51(7):9–11, 2008.

[58] J. L. Herrero, F. Sanchez, O. Sanchez, and M. Toro. Fault tolerance AOP approach. In *Workshop on AOP and Separation of Concerns*, pages 44–52, 2001.

[59] Y. Huang, S. Mohapatra, and N. Venkatasubramanian. An energy-efficient middleware for supporting multimedia services in mobile grid environments. In *Proceedings of International Conference on Information Technology: Coding and Computing, 2005.*, volume 2, April 2005.

[60] B. Hull, K. Jamieson, and H. Balakrishnan. Mitigating congestion in wireless sensor networks. In *ACM SenSys 2004*, Baltimore, MD, November 2004.

[61] A. Ibrahim, Y. Jiao, E. Tilevich, and W. R. Cook. Remote batch invocation for compositional object services. In *Proceedings of the* 23^{rd} *European Conference on Object-Oriented Programming (ECOOP)*, 2009.

[62] K. Jain, J. Padhye, V. N. Padmanabhan, and L. Qiu. Impact of interference on multi-hop wireless network performance. In *Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 66–80, 2003.

[63] JAX-WS Expert Group. JSR-224 Java API for XML-based Web services 2.0. Technical report, Java Community Process, 2006.

[64] JBoss AOP. `http://www.jboss.org/jbossaop`.

[65] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. . Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. In *Proceedings of the 15*th *Symposium on Operating Systems Principles*, December 1995.

[66] M. B. Juric, B. Kezmah, M. Hericko, I. Rozman, and I. Vezocnik. Java rmi, rmi tunneling and web services comparison and performance analysis. *SIGPLAN Not.*, 39(5):58–65, 2004.

[67] L. Kagal, T. Finin, and A. Joshi. A policy language for a pervasive computing environment. In *Proceedings of IEEE International Workshop on Policies for Distributed Systems and Networks*.

[68] A. Kansal and F. Zhao. Fine-grained energy profiling for power-aware application design. *SIGMETRICS Perform. Eval. Rev.*, 36(2):26–31, Aug. 2008.

[69] U. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

[70] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353, 2001.

[71] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming(ECOOP)*, pages 220–242, 1997.

[72] M. Kim, L. P. Cox, and B. D. Noble. Safety, visibility, and performance in a wide-area file system. In *Proceedings of the 1st USENIX conference on File and storage technologies (FAST '02)*, 2002.

[73] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATC)*, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.

[74] Knopflerfish - open source OSGi. `http://www.knopflerfish.org`.

[75] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proceedings of the IEEE Annual Joint Conference of the IEEE Computer and Communications Societies (Infocom)*, 2012.

[76] Y.-W. Kwon. Orchestrating mobile application execution for performance and energy efficiency. In *Proceedings of the ACM SIGPLAN Conference on Systems, Programming, Languages and Applications (SPLASH), Student Research Competition*, 2013.

[77] Y.-W. Kwon and E. Tilevich. A declarative approach to hardening services against QoS vulnerabilities. In *Proceedings of the 2011 IEEE International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA)*, 2011.

[78] Y.-W. Kwon and E. Tilevich. Energy-efficient and fault-tolerant distributed mobile execution. In *Proceedings of the 32nd International Conference on Distributed Computing Systems (ICDCS)*, 2012.

[79] Y.-W. Kwon and E. Tilevich. The impact of distributed programming abstractions on application energy consumption. *Information and Software Technology*, 55(9), 2013.

[80] Y.-W. Kwon and E. Tilevich. Reducing the energy consumption of mobile applications behind the scenes. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM)*, 2013.

[81] Y.-W. Kwon and E. Tilevich. Cloud refactoring: Automated transitioning to cloud-based services. *Automated Software Engineering Journal*, 2014.

[82] Y.-W. Kwon, E. Tilevich, and T. Apiwattanapong. DR-OSGi: Hardening distributed components with network volatility resiliency. In *Proceedings of the ACM/IFIP/USENIX* 10^{th} *International Middleware Conference*, 2009.

[83] Y.-W. Kwon, E. Tilevich, and W. Cook. Which middleware platform should you choose for your next remote service? *Service Oriented Computing and Applications*, 5:61–70, 2011.

[84] Y.-W. Kwon, E. Tilevich, and W. R. Cook. An assessment of middleware platforms for accessing remote services. In *Proceedings of the 2010 IEEE International Conference on Services Computing (SCC)*, 2010.

[85] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, volume 10, pages 707–710, 1966.

[86] S. Li and L. Tahvildari. A service-oriented componentization framework for java software systems. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, 2006.

[87] D. Liang, C. Fang, and C. Chen. FT-SOAP: A fault-tolerant web service. In *Tenth Asia-Pacific Software Engineering Conference, Chiang Mai, Thailand*, 2003.

[88] A. Liu, Q. Li, L. Huang, and M. Xiao. FACTS: A framework for fault-tolerant composition of transactional web services. *IEEE Transactions on Services Computing*, 3(1):46–59, 2010.

[89] Y. Liu. Energy-efficient synchronization through program patterns. In *Proceedings of the 1st International Workshop on Green and Sustainable Software (GREENS)*, 2012.

[90] G. A. D. Lucca, A. R. Fasolino, P. Guerra, and S. Petruzzelli. Migrating legacy systems towards object-oriented platforms. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 122–129, Washington, DC, USA, 1997. IEEE Computer Society.

[91] A. Marchetto and F. Ricca. From objects to services: toward a stepwise migration approach for Java applications. *Int. J. Softw. Tools Technol. Transf.*, 11(6):427–440, 2009.

[92] T. J. McCabe. A complexity measure. In *Proceedings of the 2nd International Conference on Software Engineering*, Los Alamitos, CA, USA, 1976.

[93] P. McGachey, A. L. Hosking, and J. E. B. Moss. Pervasive load-time transformation for transparently distributed Java. *Electron. Notes Theor. Comput. Sci.*, 253:47–64, December 2009.

[94] Microsoft. Component Object Model (COM).

[95] Microsoft Research. Network Emulator for Windows Toolkit (NEWT) version 2.1, 2010.

[96] A. Miettinen and J. Nurminen. Energy efficiency of mobile clients in cloud computing. In *Proceedings of the 2nd USENIX conference on Hot Topics in Cloud Computing*, 2010.

[97] M. Mikic-Rakic and N. Medvidovic. A classification of disconnected operation techniques. In *Proceedings of the 32nd EUROMICRO Conference on Software engineering and Advanced Applications (EUROMICRO-SEAA'06)*, 2006.

[98] B. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[99] S. Mohapatra, N. Dutt, A. Nicolau, and N. Venkatasubramanian. DYNAMO: A cross-layer framework for end-to-end QoS and energy optimization in mobile handheld devices. *IEEE Journal on Selected Areas in Communications*, 25(4):722 –737, 2007.

[100] R. Monson-Haefel and D. Chappell. *Java Message Service*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[101] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In *Proceeding of the 17th international conference on World Wide Web (WWW '08)*, 2008.

[102] B. D. Noble, M. Sayanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997.

[103] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier. A preliminary study of the impact of software engineering on greenIT. In *Proceedings of the 1st International Workshop on Green and Sustainable Software*, 2012.

[104] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier. Runtime monitoring of software energy hotspots. In *Proceedings of the* 27^{th} *IEEE/ACM International Conference on Automated Software Engineering (ASE '12)*, 2012.

[105] Object Management Group. The CORBA security service specification. Specification, Object Management Group, 2002.

[106] Object Management Group. The CORBA component model specification. Specification, Object Management Group, 2006.

[107] OSGi Alliance. OSGi release 4.1 specification. Specification, 2007.

[108] OSGi Alliance. RFP 133 cloud computing. 2010.

[109] Paremus Ldt. The Paremus service fabric - a technical overview, 2008.

[110] P. Parrend and S. Frénot. Classification of component vulnerabilities in java service oriented programming (sop) platforms. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering (CBSE '08)*, pages 80–96, Berlin, Heidelberg, 2008. Springer-Verlag.

[111] A. Pathak, Y. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7^{th} ACM European Conference on Computer Systems (EuroSys)*, 2012.

[112] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Enabling automatic offloading of resource intensive smartphone applications. Technical Report ECE-TR-11-3, Purdue University, 2011.

[113] K. Pentikousis. In search of energy-efficient mobile networking. *Communications Magazine, IEEE*, 48(1):95–103, January 2010.

[114] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, 2000.

[115] A. Polze, J. Schwarz, and M. Malek. Automatic generation of fault-tolerant CORBA-services. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 2000)*, 2000.

[116] R. Powers. Batteries for low power electronics. *Proceedings of the IEEE*, 83(4):687 –693, apr 1995.

[117] R. Pressman. *Software engineering: a practitioner's approach*. McGraw-Hill Higher Education, 2005.

[118] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling resource usage for mobile applications: a cross-layer approach. In *Proceedings of the 9^{th} International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 2011.

[119] J. S. Rellermeyer and G. Alonso. Concierge: a service platform for resource-constrained devices. In *the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 245 – 258, 2007.

[120] J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-OSGi: Distributed applications through software modularization. In *Proceedings of the ACM/IFIP/USENIX 8th International Middleware Conference (Middleware)*, November 2007.

[121] A. Rountev. Precise identification of side-effect-free methods in Java. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, 2004.

[122] K. Sachs, S. Kounev, J. Bacon, and A. Buchmann. Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark. *Performance Evaluation*, 66(8):410 – 434, 2009.

[123] S. M. Sadjadi and P. K. McKinley. ACT: An adaptive CORBA template to support unanticipated adaptation. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS '04)*, pages 74 – 83, 2004.

[124] Security Annotation Framework. `http://safr.sourceforge.net/`.

[125] U. Saif and D. Greaves. Communication primitives for ubiquitous systems or RPC considered harmful. In *Proceedings of the* 21st *International Conference Distributed Computing Systems Workshop*, 2001.

[126] G. T. Santos, L. C. Lung, and C. Montez. FTWeb: A fault tolerant infrastructure for web services. *Enterprise Distributed Object Computing Conference, IEEE International*, 0:95–105, 2005.

[127] M. Satyanarayanan, V. Bahl, R. Caceres, and N. Davies. The case for VM-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[128] D. Scott. Assessing the costs of application downtime. Technical report, Gartner Group, 1998. `www.gartner.com`.

[129] C. Seo, S. Malek, and N. Medvidovic. An energy consumption framework for distributed Java-based systems. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2007.

[130] S. Sidiroglou, G. Giovanidis, and A. D. Keromytis. A dynamic mechanism for recovering from buffer overflow attacks. In *Proceedings of the 8th Information Security Conference*, 2005.

[131] L. Siegele. Let it rise: A special report on corporate IT. *The Economist (October 2008)*.

[132] T. Smith and M. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.

[133] A. Spiegel. *Automatic Distribution of Object Oriented Programs*. PhD thesis, FU Berlin, FB Mathematik und Informatik, 2002.

[134] Spring Framework. `http://www.springsource.org/`.

[135] Spring Security. `http://static.springsource.org/spring-security/site/`.

[136] J. W. Stamos and D. K. Gifford. Remote evaluation. *ACM Trans. Program. Lang. Syst.*, 12(4):537–564, 1990.

[137] Y.-Y. Su and J. Flinn. Slingshot: Deploying stateful services in wireless hotspots. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, 2005.

[138] M. Sullivan and R. Chillarege. Software defects and their impact on system availability-a study of field failures in operating systems. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing,, year=1991,*.

[139] S. Tambe, A. Dabholkar, J. Balasubramanian, and A. Gokhale. Automating middleware specializations for fault tolerance. In *Proceedings of the International Symposium on*

Object/component/service-oriented Real-time distributed Computing (ISORC 2009), March 2009.

[140] E. Tan, L. Guo, S. Chen, and X. Zhang. PSM-throttling: Minimizing energy consumption for bulk data communications in WLANs. In *Proceedings of the IEEE International Conference on Network Protocols*, 2007.

[141] A. S. Tanenbaum and R. v. Renesse. A critique of the remote procedure call paradigm. In *EUTECO 88*, 1988.

[142] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Distributed Execution of ”Legacy” Java Software. In *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[143] B. Tay and A. Ananda. A survey of remote procedure calls. *Operating Systems Review*, 24(3):68–79, 1990.

[144] The Apache Software Foundation. Felix. `http://felix.apache.org/site/index.html`.

[145] The Apache Software Foundation. Lucene. `http://lucene.apache.org/`.

[146] The Apache Software Foundation. Nutch. `http://wiki.apache.org/nutch/NutchOSGi/`.

[147] The Apache Software Foundation. Solr. `http://lucene.apache.org/solr`.

[148] The Apache Software Foundation. ActiveMQ. `http://activemq.apache.org/`, 2010.

[149] The Apache Software Foundation. Apache CXF Distributed OSGi. `http://cxf.apache.org/distributed-osgi.html`, 2010.

[150] The Economist Editorial Staff. Creating the cumulus: Software will be transformed into a combination of services. *The Economist*, 2008.

[151] C. Thompson, H. Turner, J. White, and D. Schmidt. Analyzing mobile application software power consumption via model-driven engineering. In *Proceedings of the* 1^{st} *International Conference on Pervasive and Embedded Computing and Communication Systems*, 2011.

[152] E. Tilevich and Y.-W. Kwon. Cloud-based execution to improve mobile application energy efficiency. *IEEE Computer*, 47(1), 2014.

[153] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP '02)*, 2002.

[154] E. Tilevich and Y. Smaragdakis. NRMI: Natural and efficient middleware. *IEEE Transactions on Parallel and Distributed Systems*, 19(2):174–187, 2008.

[155] E. Tilevich and Y. Smaragdakis. J-orchestra: Enhancing java programs with distribution capabilities. *ACM Trans. Softw. Eng. Methodol.*, 19(1):1–40, 2009.

[156] P. Tran, P. Greenfield, and I. Gorton. Behavior and performance of message-oriented middleware systems. In *Proceedings of the* 22^{nd} *International Conference on Distributed Computing Systems (ICDCS '02)*, 2002.

[157] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the conference of the Centre for Advanced Studies on Collaborative research (CASCON '99)*, 1999.

[158] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009.

[159] J. Viega, J. T. Bloch, and P. Ch. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14:31–39, 2001.

[160] S. Vinoski. RPC under fire. *IEEE Internet Computing*, pages 93–95, 2005.

[161] J. Waldo, A. Wollrath, G. Wyant, and S. Kendall. A note on distributed computing. Technical report, Sun Microsystems, Inc. Mountain View, CA, USA, 1994.

[162] Y. Weinsberg and I. Ben-Shaul. A programming model and system support for disconnected-aware applications on resource-constrained devices. In *Proceedings of the 24th International Conference on Software Engineering*, pages 374 – 384, Orlando, Florida, May 2002.

[163] A. Weissel, B. Beutel, and F. Bellosa. Cooperative I/O: A novel I/O semantics for energy-aware applications. *ACM SIGOPS Operating Systems Review*, 36(SI):117–129, 2002.

[164] D. West. Saving money through cloud computing. Technical report, Governance Studies at the Brookings Institution, 2010.

[165] A. Wollrath, R. Riggs, J. Waldo, et al. A distributed object model for the Java system. *Computing Systems*, 9(4):265–290, 1996.

[166] K. Wuyts, R. Scandariato, G. Claeys, and W. Joosen. Hardening XDS-based architectures. In *Proceedings of the 3rd International Conference on Availability, Reliability and Security (ARES '08)*, pages 18–25, Washington, DC, USA, 2008. IEEE Computer Society.

[167] Y. Xiao, R. Kalyanaraman, and A. Yla-Jaaski. Energy consumption of mobile YouTube: Quantitative measurement and analysis. In *Proceedings of the 2ndInternational Conference on Next Generation Mobile Applications, Services and Technologies (NGMAST '08)*, 2008.

[168] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of the 21st IEEE Computer and Communications (INFOCOM '02)*, volume 3. IEEE, 2002.

[169] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. *ACM SIGOPS Operating Systems Review*, 37(5):149–163, 2003.

[170] M. Zhang and R. Wolff. Crossing the digital divide: cost-effective broadband wireless access for rural and remote areas. *Communications Magazine, IEEE*, 42(2):99–105, Feb 2004.

[171] Z. Zheng and M. Lyu. Optimal fault tolerance strategy selection for web services. *International Journal of Web Services Research*, 7(4):21–40, 2010.