Martin Weißbach, Nguonly Taing, Markus Wutzler, Thomas Springer, Alexander Schill, Siobhan Clarke

Decentralized coordination of dynamic software updates in the Internet of Things

Erstveröffentlichung in / First published in:

IEEE World Forum on Internet of Things (WF-IoT). Reston, 12. – 14.12.2016. IEEE Xplore, S. 171 – 176. ISBN 978-1-5090-4130-5.

DOI: https://doi.org/10.1109/WF-IoT.2016.7845450

Final edited form was published in "IEEE World Forum on Internet of Things (WF-IoT). Reston 2016", S. 171 - 176. ISBN: 978-1-5090-4130-5
https://doi.org/10.1109/WF-IoT.2016.7845450

Decentralized Coordination of Dynamic Software Updates in the Internet of Things

Martin Weißbach*, Nguonly Taing*, Markus Wutzler*, Thomas Springer*, Alexander Schill* and Siobhán Clarke+
*Computer Networks Group, Technische Universität Dresden, Dresden, Germany
+Future Cities Centre, Trinity College Dublin, Dublin, Ireland
{martin.weissbach1,nguonly.taing,markus.wutzler,thomas.springer,alexander.schill}@tu-dresden.de, siobhan.clarke@scss.tcd.ie

***Abstract*—Large scale IoT service deployments run on a high number of distributed, interconnected computing nodes comprising sensors, actuators, gateways and cloud infrastructure. Since IoT is a fast growing, dynamic domain, the implementation of software components are subject to frequent changes addressing bug fixes, quality insurance or changed requirements. To ensure the continuous monitoring and control of processes, software updates have to be conducted while the nodes are operating without losing any sensed data or actuator instructions. Current IoT solutions usually support the centralized management and automated deployment of updates but are restricted to broadcasting the updates and local update processes at all nodes. In this paper we propose an update mechanism for IoT deployments that considers dependencies between services across multiple nodes involved in a common service and supports a coordinated update of component instances on distributed nodes. We rely on LyRT on all IoT nodes as the runtime supporting local disruption-minimal software updates. Our proposed middleware layer coordinates updates on a set of distributed nodes. We evaluated our approach using a demand response scenario from the smart grid domain.**

***Keywords*-Coordinated dynamic update; decentralized middleware; unanticipated adaptation; dynamic instance binding;**

I. Introduction

The interconnection of things via the Internet as envisioned by the Internet of Things (IoT) is a key technology for a multitude of new application areas as autonomous vehicles, factory automation, smart grid or smart home. Since IoT is a fast growing and dynamically developing domain, software changes to address bugs, quality issues or changed requirements occur frequently. An efficient mechanism for the distribution and installation of updates is therefore required.

Due to the vast number of devices involved in large setups and their widespread spacial distribution, IoT systems enter a new order of magnitude in terms of scale and heterogeneity. This requires an automated distribution and update process. In addition, all nodes should be continuously in operation to ensure that no sensed data or actuator instruction gets lost. Therefore, the update mechanism should be disruption-minimal ensuring minimized stop time and zero data loss. Finally, software running on a set of distributed IoT nodes might be interdependent. A coordination of updates on all of these nodes is thus required to ensure that the IoT system is always in a consistent configuration.

Consider, for instance, a service for demand response that is running in the cloud. It is connected to IoT Gateways of houses to monitor energy production and consumption in these houses. We foresee a communication component on each node that is responsible for sending and receiving sensed data and device commands (see Figure 1). Assuming that the data is transferred unencrypted, the communication component on each node has to be updated to support confidentiality of data.

Updating the communication components on the distributed nodes completely independent from each other could lead to a temporarily inconsistent system configuration. In the case the encryption could be activated at the sender side first (e.g., an IoT Gateway), all messages sent by that communication component are encrypted from that point in time. Depending on the time difference between the update on the IoT Gateway and the cloud server, a number of encrypted messages arrives at the cloud server but cannot be processed since the server can only handle plain messages, which results in the loss of the transmitted data. If the demand response service updates first, information will also be lost since the data is still transmitted unencrypted while the service expects data to be encrypted.

Existing dynamic software update approaches still require human interaction and cannot be realized completely automated [1], especially not in distributed infrastructures in which applications deployed to several devices have to be updated. Performing updates on a distributed application resembles the (self-)adaptation of distributed software systems in which components or services might be added, removed, exchanged or migrated dynamically at run time. However, these approaches require application-specific knowledge on the communication patterns of the adaptable software system [2], [3], [4], which we consider impracticable for a general approach for dynamic software updates in distributed environments.

In this paper we propose an update mechanism for IoT systems that coordinates the update of multiple distributed nodes involved in a running service. Our solution consists of two parts. First, a mechanism for consistent unanticipated adaptation ensures disruption-minimal and safe updates on local nodes. It is based on the principle of dynamic instance binding as introduced in our previous work [5], [6]. The mechanism is implemented by *LyRT*, the runtime we assume to be installed on all IoT nodes. Second, a decentralized

Final edited form was published in "IEEE World Forum on Internet of Things (WF-IoT). Reston 2016", S. 171 - 176. ISBN: 978-1-5090-4130-5
https://doi.org/10.1109/WF-IoT.2016.7845450

middleware layer coordinates software updates on distributed nodes transactionally to ensure a consistent transition of the application throughout the update process. We evaluated our approach by implementing the demand response scenario introduced above. We demonstrate the support of zero data loss and measure the interrupt time in a setup with four nodes.

II. Coordinated Dynamic Software Updates

Our introductory example illustrated the need for a simultaneous update of software running on a set of distributed IoT nodes to prevent mismatching communication due to the added encryption functionality. Such an update requires a reliable mechanism for local updates on the one hand and a means for coordinating a number of local updates on distributed nodes on the other hand. In this section we present our solution for coordinated dynamic software updates that consists of three parts. The first part is an architecture for handling software updates in distributed IoT infrastructures. The architecture incorporates the mechanism for local updates and a middleware for the coordination of updates on multiple nodes as parts two and three of our solution, respectively.

We assume our approach to be implemented on more computationally powerful IoT devices, e.g., IoT Connectors or Cloud Platforms, that are capable of hosting applications that can contribute different services to the system. Devices characterized by strict resource constraints, e.g., sensor nodes or certain mobile devices, might not be able to provide the required computational power to cater neither role-based IoT applications nor our proposed update mechanism, but may rely on specialized solutions for such resource-constraint devices. Data providers, such as the *Wind Turbine* or the *Photovoltaic system* (PS) in our introductory example, can be considered such resource-constraint devices that would require specifically tailored update mechanisms. We further assume all involved nodes to belong to a single management domain, which could be, for instance, a vendor specific IoT ecosystem, e.g., Google Nest[1] or Samsung ARTIK[2].

A. Architecture

Our approach follows a three-layer architecture as depicted in Fig. 1. It is closely related to the feedback loop presented by Oreizy et al. [7], which is split into the *Adaptation Management* sub-loop responsible for calculating all necessary steps to change the software system and the *Evolution Management* sub-loop that ensures a consistent modification of the system.

The uppermost layer in our architecture consists of the *Update Initiator* and the *Repository* and can be mapped to the *Adaptation Management* control loop by Oreizy et al [7]. The *Repository* serves as a globally available hub to distribute available updates within the system. The *Update Initiator* is assumed to maintain knowledge on the distributed service deployment and triggers the system to update by calculating an *Update Prescription* which contains a set of update operations that prescribe the parts of the system to be updated.

[1]https://nest.com

[2]http://developer.samsung.com/artik

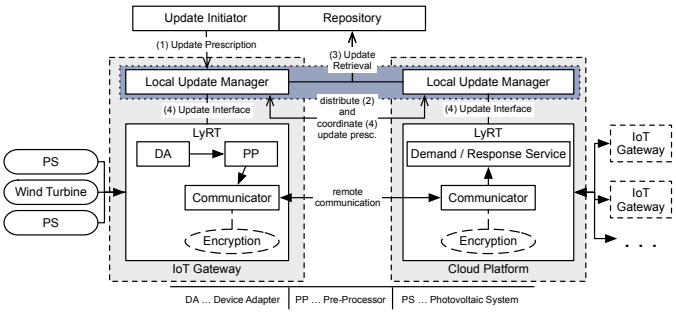

Figure 1. Architecture to enable dynamic software updates in an exemplary IoT infrastructure

The set of *Local Update Manager*s in the system represents the second layer in our architecture: the *Coordination Middleware Layer*. The subset of update managers collaborate to coordinate their respective local updates in order to ensure a consistent update of the system. A consistent update is coined by either performing all updates specified in the issued update prescription successfully or none at all.

The lowest layer in our architecture is the *Application Layer* which represents the actual application the user is interacting with. This layer hosts an updatable application that is implemented using *LyRT*[3] [5], [6], which provides basic functionality to allow for local dynamic software updates. The *Local Update Manger* and *LyRT* share a common interface to perform the update and control it locally. Since *Coordination Middleware* and *Application Layer* collaborate to update the system consistently, both layers can be mapped to the *Evolution Management* sub-loop presented by Oreizy et al. [7].

The general update procedure works as follows: the *Update Manager* triggers the update process by issuing an *Update Prescription* to a random *Local Update Manager* (LUM) hosted on one of the devices affected by the update. Subsequently, the LUM examines the received update prescription and forwards it to all LUMs on devices specified by the update operations contained in the *Update Prescription*. All involved local update managers download the updated role information from the globally available repository (cf. steps (1) through (3) in Fig. 1). After all local update managers acknowledged the download, the coordination of the local update procedures is started (cf. step (4) in Fig. 1).

B. Consistent Local Update

LyRT is a role-based run-time framework for context-dependent applications. In accord with the concept of roles, software implemented with LyRT is separated into static parts, modeled as *players*, also referred to as *core objects*, and dynamic parts, modeled as *roles*. In our demand response scenario we modeled the *Device Adapter*, *Pre-Processor* and *Communicator* in the IoT Gateway and the Demand Response Service and the *Communicator* in the cloud platform as players that are depicted as rectangles in Fig. 1. The updates for both nodes are modeled as roles that are represented as ellipsis.

[3]Prototype is available at https://github.com/nguonly/lyrt-with-transaction

Final edited form was published in "IEEE World Forum on Internet of Things (WF-IoT). Reston 2016", S. 171 - 176. ISBN: 978-1-5090-4130-5
https://doi.org/10.1109/WF-IoT.2016.7845450

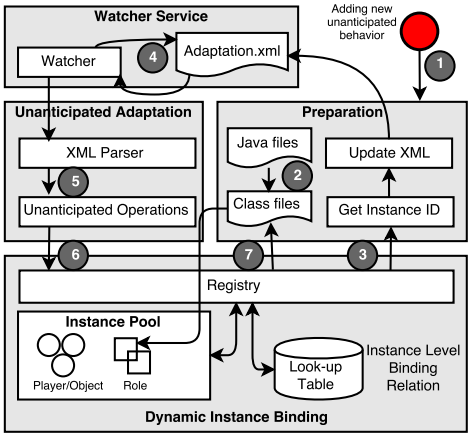

Figure 2. LyRT's Run-time Architecture

The framework's design is based on the concept of *dynamic instance binding*, which allows for arbitrary binding of players and roles. At design time, players and roles are modeled as unrelated types being initialized independently at run time. To correlate these two entities at run time, a centralized look-up table is used to store the role binding information for the dynamic method dispatch. Method calls to the player are intercepted by the dispatcher that delegates the invocations to the responsible role instance using the binding information in the look-up table. Roles can be reloaded, replaced or transferred forcing the player's run-time behavior to change.

1) LyRT's Runtime Architecture: The architecture of *LyRT* that is depicted in Figure 2 comprises four main components: *Preparation*, *Watcher Service*, *Unanticipated Adaptation* and *Dynamic Instance Binding Runtime*. To add unanticipated behavior to a player, seven steps implemented by these components have to be performed: unknown, new behavior, which is implemented as a role is taken (step 1) and is subsequently compiled in step 2 (*Preparation*). New roles are going to be bound to existing core objects, so that the core instances' identities must be queried from the runtime (step 3). Subsequently, the required (un-)bind operations are configured in the *Adaptation.xml* (step 4). The *Watcher Service*, a daemon executing in a separate thread, monitors the change of the XML file and fires events to the *Unanticipated Adaptation* component, which parses the XML file (step 5). In step 6, the runtime prepares for the adaptation with respect to the operations defined in the XML file. Finally, in step 7, classes are reloaded and bound to particular running objects granting new behaviors. In *LyRT*, dynamic parts or roles are adaptable entities that can be replaced or reloaded.

2) Consistent Unanticipated Adaptation: *LyRT* is designed to safely feature both anticipated and unanticipated adaptation applicable to the instance level. Supporting unanticipated adaptation consequently results in *Dynamic Software Update* (DSU) as the static parts change their behaviors [1]. The watcher service triggers the asynchronous (un-)binding of roles and players thus changing the behavior of the running instance of the player. As a consequence, a system may behave unexpectedly, which is undesirable. To solve this issue, *LyRT* introduces the notion of transaction at object level, inspired by the concept of tranquility [2], to safeguard the consistency of objects' behaviors inside a particular transaction [6]. This allows for disruption-minimal behavioral updates of the same instance that is executed in a different thread.

A transaction in *LyRT* works on top of the centralized dynamic method dispatch that propagates the right invocation to the valid roles of a particular executing transaction. We illustrate two showcases — binding and unbinding roles. (1) Assume there is an object instance running inside a transaction T_1 when later new roles are bound asynchronously to that instance caused by an update. In this scenario, the role is disregarded by the method dispatcher for the resolution of the method invocation target. If another transaction T_2 runs in parallel but starts after the role binding process finished, the same core object instance now shows the new behavior because the role is now used by the method dispatcher for the invocation resolution. (2) As a result of another evolution, roles may be removed while they are performing computation tasks within a transaction. Since the transaction is still ongoing, these roles cannot be removed immediately. Instead, they will be marked as phantom roles in order to ensure consistency. These phantom roles will be garbage-collected as soon as all transactions executing these roles have been finished.

C. Coordinated Adaptation Management

LyRT offers the possibility to dynamically update applications but is limited to a local runtime. In the case of our introductory example, we add a new `Encryption` role to the `Communicator` as a software update due to changed requirements. Imagine the update would be disseminated and deployed to each computing device individually without further coordination. The transmission times of the update prescription and the download speed of the actual update surely differs between the computing devices. As described in the introduction, uncoordinated installation of related updates on two nodes can lead to a mismatch between encryption and decryption at sender and receiver IoT nodes. Consequently, data would be lost. With respect to our introductory example, Listing 1 depicts an excerpt of an *Update Prescription* showing one update operation to add the encryption functionality to the communicator core object on the cloud platform, which is represented by the `Address` attribute of the operation.

Listing 1. Example of a Transaction showing one update operation

```
Transaction:T01:command
...
OperationID:O05:update
OperationType:ADD
TargetNode:
    Player:de.tud.rosi.da.iot.Communicator
    Role:de.tud.rosi.da.iot.evolution.Encryption
    Address:10.211.55.4
...
```

The LUMs use `Report` messages to indicate the status of the download, i.e., whether it could be finished successfully or not (cf. *Distribute Update* in Figure 3). A `Report` message contains information about the (1) transaction it belongs to

Final edited form was published in "IEEE World Forum on Internet of Things (WF-IoT). Reston 2016", S. 171 - 176. ISBN: 978-1-5090-4130-5
https://doi.org/10.1109/WF-IoT.2016.7845450

and (2) a set of pairs of unique IDs of the operation and the actual status information (i.e., true or false), cf. Listing 2. LUMs incorporate all previously received status information from updates on other devices and share it when propagating their own reports to increase the coordination protocol's tolerance against loss of report messages. If a LUM still misses status information about other updates, it starts requesting the information proactively from the LUM, from which a report is still missing, and other peers.

Listing 2. Example of a Report message for an Update operation

```
Transaction:T01:report
AdaptationOperation:O05
SuccessState:true
...
```

Besides lost coordination messages, devices or local parts of the application may fail during the update process. Thus, LUMs might be unreachable or might not reach their local *LyRT* runtime. Our coordination protocol tries to continue the update process on nodes that are not affected from the failure, i.e., roles and players do not communicate with the peers on the failed device. Please note that affected peers continue their service due to updates being inactive at first when performed. When the device or local runtime is available again, compensation mechanisms provided by the LUM eventually finish the update process. Due to space restrictions, we focus on a failure-free scenario in the remainder of the section.

Having confirmed the distribution of the update, the local update managers start to prepare the actual update procedure. In a first step, the updated role information is injected to the local application's runtime and the update is performed as described previously. However, changes are still inactive unless they are explicitly activated by the LUM, which means the system continues to work as before the update. Accessing the local run-time model of the application, LUMs determine local and remote roles the role under update is collaborating with and force them into a passive state. In the case of remote collaborations, LUMs send `Passivate` commands to remote LUMs to enforce the passivation of the remote role. Remote LUMs respond with `Report` messages to indicate that their local roles are now in a passive state. Other approaches were proposed that reach such a safe state to perform updates or adaptations without explicit messages to inactivate parts of the system [3], [2], but they rely on knowing the communication patterns within the system's components to determine inactive components.

In order to determine, when all local updates could be performed successfully, the LUMs exchange `Report` messages following the pattern previously described. Having confirmed the successful update of all application parts, the LUMs start activating their locally performed updates and all remote roles and players they passivated. Transmitted `Activate` and `Passivate` operations between LUMs contain the same set of information about roles and players that is to be activated or passivated, respectively, as the update operation depicted in Listing 1. Fig. 3 exemplarily displays the coordination of the update process among two LUMs.

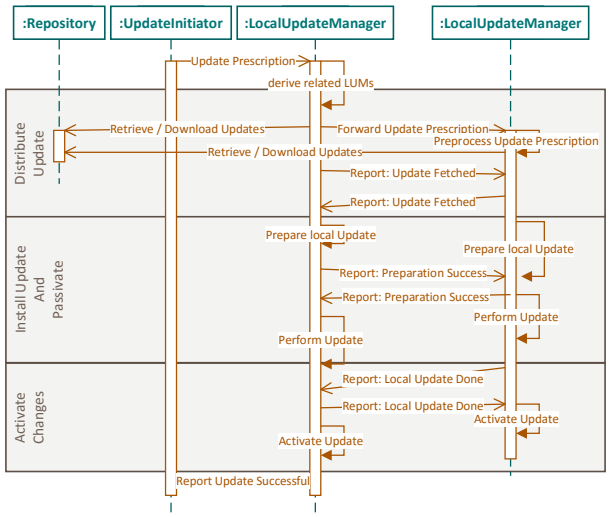

Figure 3. Message Exchange of Local Update Managers to coordinate the distributed update

III. Implementation

We implemented the *Local Update Manager* using Java 1.8 without relying on any additional frameworks or libraries. Each local update manager maintains two interfaces: (1) to communicate with other update managers on remote devices, and (2) with its local application runtime, i.e., *LyRT*. Both interfaces use a socket-based communication channel that provides transport layer functionality to the update manager. In order to receive messages from other update managers, each local update manager listens to a specific port for inbound messages, e.g., new update prescriptions or coordination messages throughout the update procedure. The overall exchange of messages between the local update managers is depicted in Figure 3. The Local Update Managers transition through the three stages during the update procedure is depicted in Figure 3. The most significant phase is the second (*Install Update and Passivate*) since LUMs coordinate the transition of the application to a safe state using the previously described protocol messages, which are currently implemented in plain text format as depicted in Listings 1 and 2.

The *Update Initiator* was implemented as a plain Java application which we provided with a predefined update definition in order to generate the transaction script that triggers the update. Since the planning and (semi-)automated generation of such update prescription is not part of our work, we only provided a simple implementation to simulate the update initiator for evaluation purposes.

IV. Evaluation

We used our introductory example to validate the feasibility of our approach. Our experimental setup involved four devices each hosting an instance of the *Local Update Manager* and an updatable application implemented using *LyRT*.

A fifth device simulated the *Update Initiator* that calculates and transmits the *Update Prescription*. One device, which represents the cloud-platform, hosted the *demand response*

Final edited form was published in "IEEE World Forum on Internet of Things (WF-IoT). Reston 2016", S. 171 - 176. ISBN: 978-1-5090-4130-5
https://doi.org/10.1109/WF-IoT.2016.7845450

Table I
AVERAGE TIME MEASUREMENTS OF THE DISTRIBUTED DYNAMIC SOFTWARE UPDATE. ALL VALUES IN MILLISECONDS (MS).

Node	Overall	Start–Finish	Local Adapt	Activation
DS	244	182	12	5
PS 1	168	140	11	4
PS 2	105	53	42	6
WT	168	131	51	5
Across	171	126	26	5

service (DS) that is responsible for balancing the supplied and demanded energy. The other devices representing IoT Gateways hosted energy producers, namely two *Photovoltaic System*s (PS) and one *Wind Turbine* (WT). In order to conduct our experiments, we assumed the updates to be already available locally on each device, since the distribution of updates does neither affect the running system nor its interrupt time.

The DS and PS 1 were hosted on a 3.3GHz Intel Core i5-4590 with 8GB RAM running Ubuntu 15.10, whereas PS 2 was hosted on a 2.53GHz Intel Core 2 Duo P8700 with 4GB RAM running OS X 10.10.5, and WT was hosted on a 2.6GHz Intel Core i5-4278U with 16GB RAM running OS X 10.11.5 devices. We performed 10 iterations of the described test setup in our lab's local area network. The results of the experiment are summarized in Table I. Evidently, the addition of new and previously unforeseen behavior to the distributed application can be achieved in a feasible amount of time for small examples because the system only needed 26ms in average to add the new behavior. The majority of the time consumed for the update (171ms on average across all platforms) is consumed by the coordination of the update procedure to ensure a consistent activation of the added behavior. The average time the coordination middleware required to determine and propagate the successful addition of the new behavior amounted to 126ms across all devices. We randomly chose the cloud-platform hosting the *demand response service* to receive the update prescription first. On this local update manager, the time that was required to actually forward the update prescription is included in the overall time the update took place, which leads to a slightly higher mean time for the entire update process. Please note that the *demand response service* is still up and running during the time the local update manager takes to forward the update prescription. The application does thus not suffer from significantly longer system interruption due to being the initial receiver of the update. On average, the interrupt time of the system amounted to 5ms across all devices, which is the time in which the `Communicator` players have to halt their communication temporarily to activate the previously accommodated new behavior.

V. RELATED WORK

This section discusses related approaches of two domains: local runtime supporting dynamic software updates and coordination techniques to push updates in distributed settings.

A. Run-time Support for Dynamic Software Update

LyRT [5], [6] comprehensively supports dynamic behavior of software systems at the instance level, which is comparable to applications leveraged by *dynamic software updates* (DSU). We discuss LyRT's relation to context-dependent systems at the programming language level and DSU techniques.

Context-oriented Programming (COP) [8] and Role-oriented Programming (ROP) [9] significantly contribute to building dynamic run-time systems that enable objects to change their behavior through adaptation techniques. Generally, both do not support unanticipated adaptation which introduces new behavior later during execution. Some languages, such as ContextJS [10] and Context Traits [11], underpinning the meta-object protocol, provide open mechanisms for possible unanticipated adaptation. Yet, these mechanisms have not dealt with inconsistencies arisen from newly incorporated behavior.

LyRT is closer related to adaptive software systems than to DSU techniques. DSU is being transparent to programmers allowing them to almost completely update the live system without the knowledge of DSU [1] whereas in LyRT programmers need to adopt the concepts of adaptive software systems, i.e., explicitly define static and dynamic parts. This extra work is beneficial for mobile and IoT applications that dynamically adapt to the surrounding environment.

The existing Java-based DSU techniques [12], [13], [14] which provide complete transparency to programmers regardless of their architecture preferences modify a particular Java Virtual Machine (JVM). As a result, these JVM-dependent solutions break compatibility with other JVM releases. Moreover, engineering efforts and exhaustive tests are necessary as there were crashes documented [12]. JavAdaptor [15] does not modify the JVM but still relies on Java HotSwap to change the method body. The application states must be mapped manually by the programmer. Rubah [16] also works on the stock JVM and supports state transition. However, it is not completely transparent to programmers as they need to know how to inject Rubah's code for future updates. LyRT works on the standard JVM, too, and programmers need to conform to the framework design. (Re-)loading roles at run time relies on the simple dynamic class loader that has been adopted in all major frameworks, e.g., OSGi[4]. Since OSGi supports hot swapping of components, the overhead to update only one class is much higher since the whole component has to be reloaded. LyRT allows fine-grained modifications through object instance adaptations.

B. Coordination

Dynamic software updates focus on the realization of updates at run time using appropriate mechanisms and techniques to achieve the desired degree of variability. Performing multiple updates on different devices in a coordinated manner has not yet been addressed by this community. However, the issue of changing the behavior of a software system at run time is subject to the self-adaptive software systems community.

[4]www.osgi.org

Final edited form was published in "IEEE World Forum on Internet of Things (WF-IoT). Reston 2016", S. 171 - 176. ISBN: 978-1-5090-4130-5
https://doi.org/10.1109/WF-IoT.2016.7845450

Works of Kramer & Magee [3] and Vandewoude et al. [2] propose mechanisms to determine a safe point in time to perform the adaptation but rely on complete knowledge of the component's interaction pattern to detect idle hence updatable components. Relying solely on dynamic object binding techniques, we do not have this knowledge of the interaction pattern between parts of the system. However, performing updates only in safe states of the system is a necessary requirement to prevent inconsistent behavior and data loss.

Therefore, we restrict the system's behavior (i.e., the passivation of roles) before we perform updates with the aim to reach a safe state to perform the update in an reasonable amount of time. As soon as a safe state is reached we conduct the update at a given time t, and experience the updated system's behavior at a time $t+1$. In their formal specification of adaptation semantics, Zhang et al. [17] considered adaptations that restrain the system's functionality to reach a safe state *guided adaptation*, which is also applicable for our approach.

Approaches like FlashMop [18], DecAp [19] or Clonal Plasticity [20] consider the adaptation of distributed software systems but focus on the decentralized generation of change prescriptions and how independent agents have to collaborate to derive those plans without complete knowledge of the system. Coordinating the actual adaptation or update of the software system imposes different requirements on the protocol, e.g., ensuring a safe state to update or perform related updates simultaneously or in an ordered manner.

Other approaches focus on the design and architecture of self-adaptive software systems. Kramer & Magee, for example, proposed a widely acknowledged reference architecture for self-adaptive software systems, called 3L [3]. Our coordination middleware and application layer are a conceptual refinement of their component control layer, which provides mechanisms for the modification of the component-based applications.

VI. Conclusion

In this paper we addressed the issue of software updates in IoT infrastructures. From the domain of self-adaptive systems we adopt a mechanism for consistent unanticipated adaptations to safely update software on local IoT nodes in a disruption-minimal manner. To extend our solution to scenarios where software is running on a set of distributed nodes, we introduced a decentralized middleware layer that coordinates such software updates transactionally to ensure a consistent transition of the application throughout the update process.

In certain setups it is still possible for data to get lost through mismatched communication endpoints as a consequence of performed updates. For instance, in case the encryption on a sender is activated before the receiver, data would be lost since it cannot be processed by the receiver. We intend to continue our research to prevent such inconsistencies from occurring without sacrificing the unawareness of the application's internal communication pattern. A first intuitive approach would utilize NTP On-Wire[5] or the SNTP[6] protocol to synchronize clocks between devices participating in the update process and subsequently agree on a common point in time to activate the changes using a decentralized decision making or voting protocol. Eventually, only a logical point in time, e.g., a point in the instruction flow of the distributed application, is required for simultaneous activation of changes across multiple devices. The approach proposed in [21] could serve as a foundation for further investigation.

[5]https://tools.ietf.org/html/rfc5905

[6]https://tools.ietf.org/html/rfc4330

Acknowledgements

This work is partially funded by the German Research Foundation (DFG) within the Research Training Group "Role-based Software Infrastructures for continuous-context-sensitive Systems" (GRK 1907) and the Erasmus Mundus Program.

References

[1] H. Seifzadeh *et al.*, "A survey of dynamic software updating," *Journal of Software: Evolution and Process*, vol. 25, no. 5, pp. 535–568, 2013.

[2] Y. Vandewoude *et al.*, "Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 856–868, Dec. 2007.

[3] J. Kramer and J. Magee, "Self-Managed Systems: an Architectural Challenge," in *FOSE '07*. IEEE, May 2007, pp. 259–268.

[4] H. Gomaa *et al.*, "Software adaptation patterns for service-oriented architectures." *SAC*, pp. 462–469, 2010.

[5] N. Taing *et al.*, "A dynamic instance binding mechanism supporting run-time variability of role-based software systems," in *Modularity Companion*. ACM, 2016, pp. 137–142.

[6] N. Taing *et al.*, "Consistent unanticipated adaptation for context-dependent applications," in *COP'16*. ACM, 2016, p. to appear.

[7] P. Oreizy *et al.*, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 54–62, May 1999.

[8] G. Salvaneschi *et al.*, "Context-oriented programming: A software engineering perspective," *Journal of Systems and Software*, vol. 85, no. 8, pp. 1801–1817, 2012.

[9] T. Kühn *et al.*, "A metamodel family for role-based modeling and programming languages," in *SLE*. Springer, 2014, pp. 141–160.

[10] J. Lincke *et al.*, "An open implementation for context-oriented layer composition in contextjs," *Science of Computer Programming*, vol. 76, no. 12, pp. 1194–1209, 2011.

[11] S. González *et al.*, "Context traits: dynamic behaviour adaptation through run-time trait recomposition," in *AOSD'13*. ACM, 2013, pp. 209–220.

[12] A. R. Gregersen and B. N. Jorgensen, "Dynamic update of java applications-balancing change flexibility vs programming transparency," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21, no. 2, pp. 81–112, 2009.

[13] T. Würthinger *et al.*, "Dynamic code evolution for java," in *PPPJ'10*. ACM, 2010, pp. 10–19.

[14] T. Gu *et al.*, "Low-disruptive dynamic updating of java applications," *Information and Software Tech.*, vol. 56, no. 9, pp. 1086–1098, 2014.

[15] M. Pukall *et al.*, "Javadaptor – flexible runtime updates of java applications," *Software: Practice and Experience*, vol. 43, no. 2, pp. 153–185, 2013.

[16] L. Pina *et al.*, "Rubah: Dsu for java on a stock jvm," in *OOPSLA'14*, vol. 49, no. 10. ACM, 2014, pp. 103–119.

[17] B. H. C. Cheng and J. Zhang, "Specifying adaptation semantics," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, 2005.

[18] D. Sykes *et al.*, *FlashMob: distributed adaptive self-assembly*, ser. distributed adaptive self-assembly. ACM, May 2011.

[19] S. Malek *et al.*, "A Decentralized Redeployment Algorithm for Improving the Availability of Distributed Systems," in *Component Deployment*. Springer Berlin Heidelberg, 2005, pp. 99–114.

[20] V. Nallur *et al.*, "Clonal plasticity: a method for decentralized adaptation in multi-agent systems," in *SEAMS'16*. ACM, May 2016, pp. 122–128.

[21] B. Ensink and V. Adve, "Coordinating Adaptations in Distributed Systems," in *ICDCS'04*. IEEE Computer Society, 2004, pp. 446–455.