Always-On Web of Things Infrastructure using Dynamic Software Updating

Martin Alexander Neumann Christoph Tobias Bach Andrei Miclaus Till Riedel Michael Beigl

Karlsruhe Institute of Technology
Vincenz-Priessnitz-Str. 1
Karlsruhe, Germany
{firstname.lastname}@kit.edu

ABSTRACT

Applications in the Internet of Things require security, high availability and real-time communications for reliable operation. But their software contains issues that need to be fixed. Timely installation of software updates allows securing vulnerable software quickly but conventionally disrupts availability and communications. Rolling update schemes prevent disruptions, but have to be performed carefully.

Dynamic software updating significantly shortens the installation duration of updates by implementing them in-memory, allowing timely hot fixing and installation of new features without service disruption or degradation in soft real-time communications. As the Web of Things settles on common technologies, we see the need for quick hot fixing of security vulnerabilities in widespread components.

To demonstrate the benefits, we present a case study in which the moquette message broker has been retrofitted for dynamic updating with our update system. We provide dynamic patches for all three releases of moquette and perform these updates on moquette at saturated load stressed by a 1:10 fan-out benchmark with 100 simulated publishers. While no connections or messages are lost, it demonstrates that the throughput drops only for 1-2s and that average message latency peaks up to 1000ms during this time.

1. INTRODUCTION

Today, things are becoming intelligent using a wide range of technologies and platforms, forming the Internet of Things (IoT). To simplify intelligent thing development and interoperability in an evolving heterogeneous ecosystem of IoT technologies, the Web of Things (WoT) sets out to streamlining these potentials by settling on a common set of technologies that simplifies interoperability between varying IoT platforms and technologies. The Web of Things (WoT) enables device mashup and system integration by promoting web communication, platform and language standards that provide a robust, efficient and flexible application layer to developers [8]. It integrates heterogeneous devices, ranging from micro-controllers to cloud-based highly-scalable servers, and heterogeneous communications, ranging from small-bandwidth unreliable wireless links to high-bandwidth and robust cable links. Prominent standards and technologies for seamless interoperability in the WoT stack are for example HTTP (reliable client-server connections), REST (scalable communications), JSON (extensible marshalling), JSON-LD (type-safe marshalling and efficient device semantics) or WebSockets, CoAP and MQTT (efficient communications).

While things become more intelligent, the relevance of software in defining the value of things increases and the software system becomes more distributed and complex. As any other software, the software of these things and their infrastructure needs to be updated in the field, either to fix bugs and security features, or to add new features and improve the user experience. Updating the large number of devices in the Internet of Things timely and securely has been identified as an important issue [11]. For example, a recent two-week delay in fixing a bug in the smart nest thermostat prevented heating in homes during winter[1].

In most use cases, e.g. when connecting vehicles, there is always work to do at the infrastructure, raising real-time communications and high-availability needs. Although updating cloud and edge devices is considered easier than large-scale resource-constraint devices [2], we argue that updating the software in the cloud and on edge devices is still challenging in face of these real-time and availability requirements.

In this paper, we present Dynamic Software Updating (DSU) as an approach for updating core components in the Web of Things. It uniquely enables *timely* and *seamless* updating, featuring updating of highly available and real-time communications systems. The goal of DSU is to produce results equivalent to conventional updating but performing updates at runtime in the applications' memory to speed up the installation significantly. Timely means that providers can immediately push security fixes or new features into the systems without having to wait for scheduled maintenance windows or wait for the initiation and completion of carefully designed *rolling updates* or *big flips* (see [4]).

Timely updating is vital to large systems incorporating many components with similar software stacks, such as the Web of Things, to make it practically secure against exploitation of vulnerabilities. Scheduled maintenance for performing conventional updates (fast reboot) allows straight-

[1] http://goo.gl/GIK3tk

forward update installation but disrupts the system, which we think is hardly an option with smart devices being widely adopted and integrated into business systems and processes and alike. In contrast, big flips or rolling updating allows non-disruptive installation of updates to redundant systems behind a load balancer, but it has to be designed carefully to prevent crashing the system during the stretched update period in which both programs versions are running [1]. In addition, the software update must be backwards-compatible to allow interaction between old and new program version while both versions are running concurrently during the rolling update schedule [4]. DSU provides a simpler update installation procedure as it does not require such backwards-compatibility or complex update scheduling when installing updates to multiple machines.

Seamless updating means that the update is low-disruptive to machine-to-machine communications, especially real-time, and to user interactions, even at highly saturated load. Updates can be performed at any time and do not have to be postponed to situations of low load. The goal of the DSU installation procedure is to perform maintenance for hot fixing and larger updating of distributed machines that lasts very shortly and therefore can be scheduled immediately as it does not disrupt runtime as regular scheduled maintenance.

We firstly discuss software updating for the different kinds of devices in the Internet of Things and related work in Section 2. Afterwards in Section 3, we present a healthcare monitoring setting and present our version of the moquette MQTT broker which has been enhanced for dynamic updating (on our Java DSU system Lusagent. The case study demonstrates programming efforts for enabling timely and seamless dynamic updates. In Section 4, we evaluate the effects on message throughput and latency when updating a moquette instance while the it is stressed by a benchmark.

2. UPDATING THE INTERNET OF THINGS

Communication of smart objects is differentiated into two models: *device-to-device* (e.g. over Bluetooth or ZigBee), *device-to-cloud* (e.g. over IPv4). The latter may be extended into *device-to-gateway* in which gateways close to the devices proxy access between cloud services and devices which are short on resources [3]. Gateways enhance security, provide protocol translation for efficiency, and provide caching of device state to lower stress on devices.

Software on devices, gateways and cloud services requires updating to fix bugs and provide new features. Security patches to widespread components, such as secure communications libraries, are particularly important to deploy quickly as large parts of a system become vulnerable to the same issue when a vulnerability is disclosed. This is challenged by uptime-sensitive services leading to delayed installation and long periods of vulnerability. For example, in case of the heartbleed bug, a duration of 2 weeks has been estimated for its fix to reach 50% of the vulnerable web servers in the public IPv4 space which amounted to 5% of all IPv4 web servers [6]. The timely installation of security patches is an important issue in software updating already today, with such common technologies and implementations being deployed at large-scale. This issue is becoming more relevant with the increase of adoption of common implementations in the context of the Web of Things, raising the need for mechanisms for timely updating of all three kinds of devices.

2.1 Gateways

Large applications with integrated smart objects, especially interactive ones such as payment or health monitoring, require their gateways and cloud services to be highly available for uninterrupted operation and provide real-time communications. But updating software on gateways usually requires to shut them down as they are designed for cost-efficiency and are usually not featuring live updating, e.g. by a redundant design with a hot standby. For fast continued operations, critical state is persisted before shutdown and reloaded after restart. Even though it may not challenge availability as the state is rather small, it may still challenge real-time communications. Established communications are usually dropped on shutdown and have to be re-established afterwards which adds to the time for persisting and reloading state. Shutdown may also mean that state may become invalid and is thrown away and reacquired after restart, especially, if caches are flushed and have to be warmed up again in usual application operation.

2.2 Cloud Services

Updating software on cloud services may either induce downtime at non-redundantly designed systems or maybe enabled hot by rolling updates or application-specific solutions. Rolling updates extend the idea of reliable computing of switching to a hot standby at runtime: such updates low-disruptively switch to a seperate instance of the new program version whose state has already been warmed-up. In contrast to dynamic updating, real-time communications may be challenged by re-establishing (or re-routing) connections to the warmed-up instance. Furthermore, rolling updates require backwards-compatibility of any updates to the application and careful distributed update scheduling. This is also the reason why this scheme is generally found rather complex to get right [5].

Alternatively, application-specific approaches to live update the parts of an application that make a conventional update take time could be adapted to speed up updating. For example, Facebook's version of memcached is a prominent example that migrates the cache content hot between program versions, effectively hot-swapping the program around cache [12]. Such schemes head towards generic DSU systems by using dynamic updating techniques to perform time-intensive parts of off-the-shelf updates in-memory.

2.3 Dynamic Software Updating

DSU systems go a step further in providing generic updating services to an application to update parts of it in-memory [10]. Such systems stop the control-flow in program parts affected by an update, transform control-flow and state to the new version and afterwards release the control-flow in the new version. Systems try to provide safety properties such that performed transformations at the point where the control-flow has been stopped are correct–usually forms of type safety such that no transformations or new program code may interact with any unknown/old types [18].

Entire-program DSU offers tested and efficient generic updating features that aim for high update flexibility in allowing performing any update to any part of a running program. It may complement application-specific updating approaches or be used as a standalone feature for new applications or be retrofitted into off-the-shelf applications. The high flexibility aimed for in the provided generic mechanisms

may be less efficient than tuned application-specific solutions (e.g. compared to Facebook's version of memcached) but it still is a good complement offering to update even those parts the application-specific ones do not.

Besides such flexibility, DSU systems try to be timely in performing updates right away when updates are released. But entire-program DSU systems offering such flexible and timely updates usually require programming: timely updating requires an application to be instrumented to reach a safe point for updating quickly when an update is available [9], and flexible updating requires update code to implement the transformations in-memory which are required. But when regarding that such instrumentations and update codes have to be developed for wide-spread applications in the context of the Web of Things, we think, that the necessary efforts outweigh the benefits: any instances on the Internet could be updated immediately without added rolling complexity and affecting real-time communications only little.

2.4 Smart Objects

Smart objects, i.e. small devices with severe constraints on power, memory, and processing (in the RFC 7228 classes 0 to 2), could also benefit from the dynamic updating approach. But as the individual devices usually carry little transient state and keep only few active connections at a time, conventional updating (i.e. fast reboot) is likely to be sufficient. Updating a resource-constraint device over-the-air in the field means to download a new firmware while the old one is still running, to shut down the old, install the new using a bootloader and restart the firmware [14]. Critical state may be written to NVRAM before updating and be quickly reloaded afterwards, and the few disconnected connections are likely less of a challenge to real-time communications from a device perspective. If many devices update simultaneously, the burst of disconnecting and reconnecting devices is more relevant to handle at the infrastructure.

This procedure is quite fast such that major concerns on this device type are functional updating (i.e. to not "brick" the device) and secure updating (i.e. to prevent installing tampered-with software). This is for example addressed by updating on a finer granularity of components that allows reverting a component update in case of failure, and by verification of cryptographic signatures on downloaded binaries [7]. Another challenge is to simultaneously update many of such distributed devices. They may be connected to the infrastructure by links of differing quality, maybe even via challenged networks, which challenges to distribute an update and to coordinate evolving the program version in lockstep [14]. They may be used in a many-tenant scenario such that legal and ethical issues (e.g. user veto's[17]) also have to be taken into account. Updates may even be complex and distributed requiring standardization for coordination [16].

3. CASE STUDY: MESSAGE BROKER

Envision a setting in which numerous sensors send their data to one or many message brokers frequently as shown in Figure 1. For example, in a hospital healthcare scenario, hundreds to thousands of patients are monitored simultaneously. Patients might be in-house, at home or on the road. Their vital signs are continuously tracked at high frequency and reported into the hospital infrastructure via gateways. Depending on the use case the data dropped off in bulk loads infrequently, e.g. in case of long-term heart rate or blood pressure diaries, or each taken measurement is uploaded instantly, for example, to analyze collapsing patients.

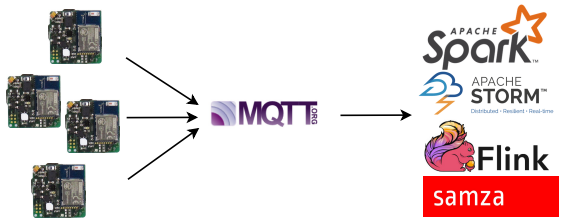

Figure 1: Schematic of Dynamic Updating Case Study

Moquette.

We suppose the scenario be driven by one to many MQTT[2] message brokers providing reliable real-time communications. Moquette[3] is a MQTT message broker of about 8k Lines of Code (LoC) in pure Java, implementing MQTT v3.1. Moquette has three releases so far: 0.7 in 2015 and 0.8 and 0.8.1 in 2016. We have instrumented these three and we provide update code for updating $0.7 \rightarrow 0.8$ and $0.8 \rightarrow 0.8.1$ dynamically[4]. DSU is performed while Moquette is benchmarked by our custom MQTT fan-out benchmark. The DSU preserves all in-flight messages and client connections. Correct DSU is validated by successful completion of the benchmark.

Lusagent DSU.

With our Java DSU system Lusagent, the programmer has to take care of two DSU related tasks: firstly, to instrument the vanilla application with control-flow migration, and secondly, to implement update code for state migration on every update to the application. The control-flow API is adopted from the DSU system Rubah, as illustrated in [15]: by replacing the application threads by specific Lusagent threads and instrumenting the application with *update points* that unwind the stack before update and rebuild it after update, this approach offers to update any code of an application, especially long-running methods. The programmer places update points to ensure timely initiation of an update: they have to be inserted such that all threads visit any of them frequently. The API is also used to make sleeps, networking and file I/O interruptible by updates.

3.1 Dynamic Update Instrumentation

Moquette is an event-driven server using the netty IO library[5] which is based on pipelines of different types of I/O handlers. Its operation is characterized by short-lived transactions of handling incoming MQTT messages in the context of netty handlers and forwarding them to subscribers.

The control-flow instrumentation API of Lusagent contains a generic factory that can be used to generate proxy objects to any given interface. The proxies are implemented by dynamically building Java proxy classes [13]. A proxy implements all methods of the given interface and can proxy an object implementing this interface to: by default, a generated proxy transparently forwards all method calls to the

[2]MQTT has been adopted in various open source and commercial IoT platforms as it provides reliable fast messaging with exactly-once delivery. Details: `http://mqtt.org`

[3]`https://git.io/viHrE`

[4]Our instrumented releases of moquette and update codes are at: `https://git.io/vPPCT` and `https://git.io/vPPWN`

[5]`https://netty.io`

proxied object when no update is requested, and it acts as a barrier to any call if an update is in progress. The factory is used to add such *update barriers* in all netty IO pipelines of Moquette to pause incoming netty events before updating. Moquette contains 4 of such pipelines, for TCP, SSL, HTTP and HTTPS, which are all instrumented by the identical change. Listing 1 shows an example for the TCP pipeline. The required change consists of adding line 2 and 9. In line 2 a proxy object is generated for the `ChannelInboundHandler` interface which wraps a freshly created `ChannelInboundHandlerAdapter` object. This object is a default implementation of a pipeline processing step that simply forwards to the next pipeline step. The wrapped object is inserted in front of the netty pipeline in line 9.

As Moquette supports MQTT QoS 0, 1 and 2, it contains an additional thread[6] that is triggered regularly by a timer to push the world forward. This thread's super class has been changed from `java.lang.Thread` to `lus.LusThread` which allows the DSU system to wait for the thread's completion before update. There is no additional instrumentation of this code to make this thread quickly reach a safe state as it is never performs long-running operations.

The control-flow of the three moquette releases have been instrumented by quite similar patches adding 11 and deleting 1 line (in versions 0.8 and 0.8.1) respectively adding 19 and deleting 3 lines (in version 0.7). In addition, to allow successful transformation of program state in the update code in the following section, two classes from version 0.7 for handling MQTT messages using the have been copy-and-pasted as inner-classes into the code of 0.8–this part of the patch amounts to 91 added and 7 deleted lines.

3.2 Dynamic Update Code

We have implemented update code in the Java-like language of our DSU system for transforming the program state at runtime from 0.7 → 0.8 and 0.8 → 0.8.1. The language allows to handle classes and objects from the old and new program simultaneously in code to access old and new fields and methods: the programmer can transform values and call functions, e.g. to perform an initialization sequence.

As just mentioned, two classes had to be manually copied from 0.7 to 0.8 to allow state transformation at runtime between these two versions. The 0.8 release changes the handling of MQTT messages in moquette. In consequence, the update code for 0.7 → 0.8 is considerably larger than the update code for 0.8 → 0.8.1 (237 vs. 33 LoC). We think that both updates are manageable and summarize their operation in the following, starting with the shorter update code.

The programmed code works in the context of the following: when the code is executed all new classes have already been loaded and their static initializations have been executed. The programmer may disable class-initialization on a per-class basis though. Furthermore, after automatic intialization, all values of old class fields that are also existing in the new program version get their values automatically copied over. In this step, values of fields deleted in a class are not copied and must be manually migrated by the programmer. New fields do not get a value copied over but keep their value after class-initialization. The programmer may disable class-copy on a per-class basis though.

Furthermore, all values of old object fields that are also existing in the new program version get their values automatically copied over. This cannot be disabled. Values of deleted object fields are analogously not copied over and new fields are initialized by generic defaults defined by the DSU system: primitive numbers are set to `0`, primitive booleans are set to `false` and object references are set to `null`.

3.2.1 Update Code 0.8 to 0.8.1

This update code (1) copies values of two object fields in the class `moquette.SubscribeMessage$Couple` that have been renamed, (2) it changes the location (i.e. object field) in which the `moquette.NettyChannel` of a client session is stored, (3) it copies a few string constants from the old into the new version which are used as keys in hash-maps (the strings in old and new version are `equal()` but not the same instance), and (4) it disables the static initializers of 4 classes in which the automatic class initialization fails as these initializers construct singleton objects which fails if the singleton has already been constructed in the old program.

3.2.2 Update Code 0.7 to 0.8

This release update changes the namespace of the project from `org.eclipse.moquette` to `io.moquette`: the update code renames the classes accordingly. Furthermore, the netty pipelines on the asynchronous communication channels have changed in this release. Therefore, the update code iterates all `moquette.NettyChannel` objects and modifies the pipelines accordingly. The message handling code has been redesigned in moquette in general, which is cared for by the large part of this update code. In the old version a combination of a LMAX disruptor ring-buffer and a MapDB[7] database are used to dispatch and store message objects. The message objects in 0.8 have changed their structure and their handling has been moved stronger into MapDB dropping the attached ring-buffer. The transformer uses about 150 LoC to read the messages in the old ring-buffer and the old MapDB, transform them into their new format, and finally, store them in the new MapDB instance.

4. EVALUATION

We evaluate the dynamic updating capability of our DSU-retrofitted version of moquette by measuring the impact of release-level updating on throughput and latency of MQTT messages. Our results are preliminary in that we have performed an experiment only with DSU on moquette to study its characteristics. We have not yet performed a conventional update (fast reboot) for baselining.

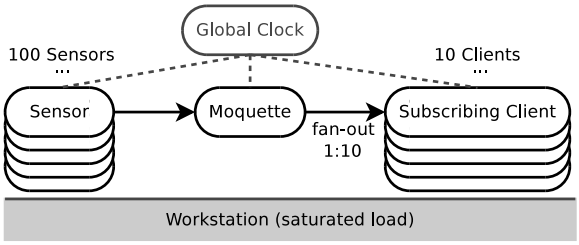

Figure 2: Moquette Benchmark Setting

[6]It's a `java.lang.Thread` in releases 0.8 and 0.8.1. In version 0.7, this part is an event handler using the LMAX disruptor messaging library (`https://git.io/viQtD`)–this has been instrumented analogously to the netty pipeline.

[7]`http://www.mapdb.org/`

```java
private void initializeTCP(IMessaging messaging, Properties props) throws IOException {
        final ChannelInboundHandler lus = LusProxy.create(new ChannelInboundHandlerAdapter(),
    ↪ ChannelInboundHandler.class);
        final NettyMQTTHandler handler = new NettyMQTTHandler();
        // ...
        initFactory(host, port, new PipelineInitializer() {
            void init(ChannelPipeline pipeline) {
                // ...
                pipeline.addLast("handler", handler);
                pipeline.addFirst(lus); // Ensure that this lus-barrier is in front of pipeline
            });
        }
```

Listing 1: Example of Moquette Control-Flow Instrumentation in `org.eclipse.moquette.server.netty.NettyAcceptor`

Our benchmarking setting is implemented in a custom script and is depicted in Figure 2: 100 sensors are simulated continuously transmitting MQTT messages at fixed rate onto the same topic of the moquette instance. Each sensor finishes transmission after sending $10k$ messages. Each message carries 128 bytes payload also containing the timestamp of their emission. 10 clients are subscribed onto the same topic, such that each incoming messages is fan out by factor $1:10$. All messages are sent and received at QoS level 2. All sensors, moquette and all subscribers are executed on the same workstation to share a common wall clock.

All sensors are equally rate-limited (1) to keep all CPU cores at saturated load to allow measuring all work necessary for dynamic updating in message throughput and latency, and (2) to keep in-flight messages in moquette at a stable level such that all transmitted messages in our benchmark expose quite short latencies and any negative impact on latency due to dynamic updating becomes evident. Keeping the number of in-flight messages at a stable level in moquette prevents their steady increase during the benchmark which would otherwise change the performance behavior in our fan-out setting considerably over time during the benchmark. The more messages are in-flight, the higher latencies in delivering MQTT messages become in the broker, and it eventually makes its performance collapse due to high memory pressure. The setting's goal is to stress moquette under normal operation without overloading it.

The performance experiments ran on a workstation with an Intel i7-2600 CPU (64-Bit SMP, 4 cores) at 3.4 GHz and dual-channel 16 GiB of RAM at 1.33 Ghz (each channel equipped with two 4 GiB modules). The workstation was disconnected from any networks and the CPU's dynamic frequency scaling and hyper-threading have been disabled. The machine ran 64-Bit Debian 4.5.3-2 (SMP support) and Oracle Java 1.8.0_92. The JVM heaps were limited to 8 GiB.

4.1 Message Throughput

We measure messages delivered per second by moquette to its subscribers (called *throughput* here) during the benchmark. At first, either moquette 0.7 (or 0.8) runs. Next, the update to 0.8 (or 0.8.1) is performed. Afterwards, measurements continue until the benchmark has finished. The dynamic update causes a pause of the broker's operation which becomes evident in its message throughput over time.

Figure 3 shows the average throughputs during the benchmarks. In both, at $10s$ the update is triggered and the throughput drops for about $1-2s$. Afterwards, the throughput returns to its previous level after a short increase from $95\% \rightarrow 100\%$ to finish messages accumulated during the dynamic. The increase for about 1-2 for the small patch ($0.8 \rightarrow 0.8.1$) and about $7-8$ for the large patch ($0.7 \rightarrow 0.8$).

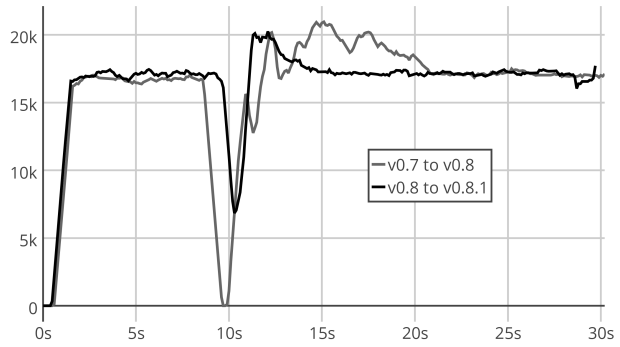

Figure 3: Message Throughput when Updating Moquette

Even though clients keep sending at fixed-rate during the update, the throughput is only negatively affected for a relatively short period of time, only for the time the dynamic update lasts, and afterwards quickly climbs back to its previous performance. The experiment demonstrates this for the worst case of normal operation.

4.2 Message Latency

To assess how the dynamic update affects the latency of the messages coming in while moquette is paused, and thus how it affects real-time communications (also when the broker is heavily loaded), we have also measured message end-to-end latency. All messages sent by the sensors are stamped by the global wall clock, whenever a subscribes receives a message, the message and its latency are tracked in a log.

Figure 4 shows the average of latencies over a sliding window of $1000ms$ with step size of $100ms$ during the benchmarks when updating moquette $0.7 \rightarrow 0.8$ and $0.8 \rightarrow 0.8.1$. This window size and step size samples the messages delayed during the update quite well as the pause for dynamic updating in our benchmarks lasts between $500ms$ to $700ms$.

In both figures, at $10s$ the update is triggered and the average latency shortly peaks for about 1-$2s$ and immediately returns to its previous level. In contrast to the previous throughput measurements, no significant difference in aver-

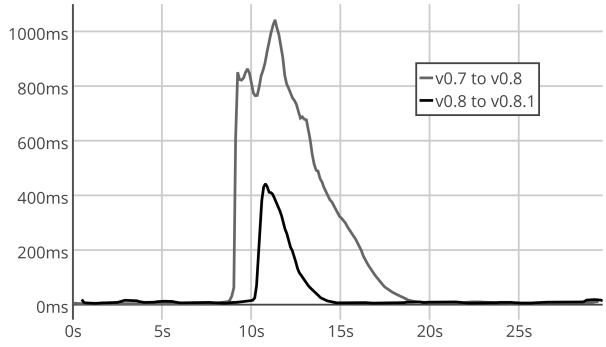

Figure 4: Message Latency when Updating Moquette

age latency before and after update are visible in updates $0.7 \rightarrow 0.8$ and $0.8 \rightarrow 0.8.1$. We think this demonstrates the potential of DSU for timely and low-disruptive updating of infrastructure components in the Internet of Things (which also has requirements on real-time communications besides high-availability on its cloud services).

5. CONCLUSION

We have discussed that applications in the Web of Things require timely hot fixing of core components while these also provide high availability and real-time communications. Timely installation of software updates allows securing vulnerable systems quickly but conventionally disrupts availability and in consequence communications. Dynamic updating has been presented as an approach significantly shortening the installation duration of updates to cloud services and gateway software by implementing them in-memory.

Dynamic updating requires programmers to implement additional update code on every regular update. But as the Web of Things sets out on settling on common standards and technologies we have argued that the low-disruptive updating benefits outweigh the programming efforts necessary: update code is implemented once per update and immediately applicable to a large number of installations. In a case study on the moquette message broker retrofitted for dynamic updating, our benchmarks demonstrate the benefits of this approach to for systems serving many clients.

Acknowledgments

This research has been partially funded by the German Federal Ministry of Education and Research as part of the UHUS project (grant no. 01IS12051).

6. REFERENCES

[1] S. Ajmani, B. Liskov, and L. Shrira. *Modular Software Upgrades for Distributed Systems*, pages 452–476. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[2] J. Arkko. Architectural Considerations with Smart Objects and Software Updates. In *Internet of Things Software Update Workshop (IoTSU)*, Dublin, Ireland, Mar. 2016.

[3] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A Survey. *Comput. Netw.*, 54(15):2787–2805, Oct. 2010.

[4] E. A. Brewer. Lessons from Giant-Scale Services. *IEEE Internet Computing*, 5(4):46–55, July 2001.

[5] T. Dumitraş and P. Narasimhan. Why do upgrades fail and what can we do about it?: Toward dependable, online upgrades in enterprise system. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 18:1–18:20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[6] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.

[7] J.-P. Fassino. Secure Firmware Update in Schneider Electric IOT-enabled offers. In *Internet of Things Software Update Workshop (IoTSU)*, Dublin, Ireland, Mar. 2016.

[8] D. Guinard, V. Trifa, and E. Wilde. A Resource Oriented Architecture for the Web of Things. In *Proceedings of Internet of Things 2010 International Conference (IoT 2010)*, Tokyo, Japan, Nov. 2010.

[9] C. M. Hayden, E. K. Smith, E. A. Hardisty, M. Hicks, and J. S. Foster. Evaluating dynamic software update safety using systematic testing. *IEEE Trans. Softw. Eng.*, 38(6):1340–1354, Nov. 2012.

[10] M. Hicks and S. Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, Nov. 2005.

[11] M. Kovatsch, A. Scholz, and J. Hund. Why software updates are more than a security issue. In *Internet of Things Software Update Workshop (IoTSU)*, Dublin, Ireland, Mar. 2016.

[12] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

[13] Oracle. Java SE Specification. WWW page, 2014. `https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html`.

[14] M. Orehek. Summary of existing firmware update strategies for deeply embedded systems. In *Internet of Things Software Update Workshop (IoTSU)*, Dublin, Ireland, Mar. 2016.

[15] L. Pina, L. Veiga, and M. Hicks. Rubah: DSU for Java on a Stock JVM. In *Proceedings of the 2014 ACM Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 103–119, New York, NY, USA, 2014. ACM.

[16] N. Smith. Toward A Common Modeling Standard for Software Update and IoT Objects. In *Internet of Things Software Update Workshop (IoTSU)*, Dublin, Ireland, Mar. 2016.

[17] R. Sparks. Avoiding the Obsolete-Thing Event Horizon. In *Internet of Things Software Update Workshop (IoTSU)*, Dublin, Ireland, Mar. 2016.

[18] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 29(4), Aug. 2007.