Low-Disruptive and Timely Dynamic Software Updating of Smart Grid Components

Martin Alexander Neumann(✉), Christoph Tobias Bach, Yong Ding, Till Riedel, and Michael Beigl

Karlsruhe Institute of Technology,
Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
{martin.neumann,christoph.bach,
yong.ding,till.riedel,michael.beigl}@kit.edu

Abstract. Components in the power grid require security, high availability and real-time communications for reliable operation. But these components are based on software that contains issues that need to be fixed. Timely installation of software updates allows securing vulnerable software quickly but conventionally disrupts availability and communications. Rolling updates on redundant systems prevent such disruptions but delay update installations as they need to be prepared carefully to update reliably. Dynamic Software Updating 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 Smart Grid settles on standardization and common technologies for interoperability, the need for timely hot-fixing and updating of software applications and libraries which are in widespread use increases.

In this paper, we discuss requirements of Smart Grid components and their updating opportunities. Afterwards, we present Lusagent, our dynamic updating system for Java 6 to 8 that is based on a novel eager program state transformation approach. We illustrate its programming efforts in a case study on an open-source Java control system framework and on several other server applications. Furthermore, we present performance measurements of dynamically updating these applications. The results demonstrate the potential of our dynamic updating approach in enabling low-disruptive and timely updating of highly available and widespread components at low and only one-time programming efforts.

1 Introduction

The power grid is becoming more connected and intelligent, forming the infrastructure of the Smart Grid. It allows highly frequent measurements at power producers, consumers and in power transmission for real-time monitoring and control. The system can immediately respond to volatile changes in generation and demand, and diagnose faults at high resolution to prevent surges. The goal is to maximize the utilization of the power infrastructure and yet improve

E.T. Lau et al. (Eds.): SmartGIFT 2017, LNICST 203, pp. 155–171, 2017.
DOI: 10.1007/978-3-319-61813-5_16

its reliability. This requires all grid components to be highly available and interconnected by real-time communications. Heterogeneous devices, ranging from micro-controllers to cloud servers, and heterogeneous communications, ranging from small-bandwidth unreliable wireless links to high-bandwidth robust cable links are interconnected by standardization and are redundant if applicable.

With the grid becoming more intelligent, the relevance of software in its components increases. As any other software, the software in the grid infrastructure needs to be updated in the field, either to fix bugs and security features, or to add new features and improve the reliability. These updates must be *low-disruptive* and carefully implemented to not affect the systems' availability and real-time communications. Besides this closed new infrastructure, the grid components are opened up to the Internet forming an open general-purpose platform for services on the Smart Grid. This raises the need to perform updates *timely* to fix bugs and security issues, especially security-critical vulnerabilities.

We discuss Dynamic Software Updating (DSU) as an approach for updating components in the Smart Grid. The goal of DSU is to produce results equivalent to conventional updating but performing updates at runtime in the application's memory to speed up their installation. It uniquely features timely and at the same time low-disruptive updating of highly available applications. Updates can be performed at any time, even at saturated load. System providers and operators are able to immediately push security fixes or new features into the systems without having to wait for low computing load, scheduled maintenance windows or wait for the setup of carefully designed *rolling updates* or *big flips* [2].

Timely updating is vital to open systems and large systems incorporating many components with similar software stacks to make it as a whole practically secure against the exploitation of vulnerabilities. Scheduled maintenance for performing conventional updates (i.e. *fast reboots*) allows straightforward update installation but disrupts the system, which is hardly an option for the components of the Smart Grid. In contrast, big flips or rolling updates allow non-disruptive installation of updates to redundant systems behind load balancers.

But big flips and rolling updates have to be designed carefully to prevent crashing the system during the stretched update period in which old and new program version are running simultaneously—leading to delayed update installations [1]. In addition, with these approaches, the software update must be backwards-compatible to allow both program versions to run concurrently during the update schedule [2]. DSU provides a simpler update installation procedure as it does not require such general backwards-compatibility or comparably complex update scheduling when installing updates to multiple machines.

In the following, we firstly discuss software updating in the Smart Grid and related work about Dynamic Software Updating in Sects. 2 and 3. Afterwards in Sect. 4, we present our DSU system for Java 6 to 8, called Lusagent. Section 5 discusses programming efforts of our approach in a case study on a highly available Java component of an open-source Supervisory Control and Data Acquisition (SCADA) system. Finally in Sect. 6, we evaluate programming efforts for

enabling low-disruptive and timely dynamic updates in general, and we assess updating performance on industry-grade Java virtual machines (JVM).

2 Updating Smart Grid Components

Complex SCADA systems for real-time data acquisition, monitoring and control are at the core of Smart Grid infrastructure. As depicted in Fig. 1, nodes at producers, consumers and in the transmission grid connect grid components to each other and to data centers. For example, the nodes implement phasor networks, consisting of PMUs (Phasor Measurement Units) and PDCs (Phasor Data Concentrators) that acquire fine-granular grid state at high frequency. Furthermore, RTUs (Remote Terminal Units) integrate decentralized energy sources into monitoring and control. And, IEDs (Intelligent Electronic Devices) integrate various grid components, such as voltage regulators and circuit breakers. Besides this, the system incorporates highly available generic computing infrastructure, such as databases or message queues and brokers in its nodes and data centers.

Software on these nodes, message queues/brokers, databases and central controllers require updating to fix bugs and to provide new features. Bugs must be fixed quickly to ensure reliability of the grid; security issues need quick care if the components are not entirely sealed in a closed environment; and new features might be interesting to deploy quickly for business or regulatory reasons. Especially, security patches to widespread components, such as cryptographic libraries, are particularly important to deploy quickly, as large parts of a system become vulnerable to the same issue when a vulnerability is disclosed. With the power grid opening up and fostering standardization, the need for timely installation of bug and security patches and other updates increases.

But quick deployment is challenged by high availability requirements, leading to delayed installation and periods of vulnerability. Even if installation is only delayed on a fraction of deployed systems, this may sum up to a large number of vulnerable systems. 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 [5].

Updating software on Smart Grid components may either induce downtime at non-redundantly designed systems or maybe enabled hot by rolling updates or application-specific solutions. Rolling updates build on the idea of reliable computing of switching to a hot standby at runtime: such updates low-disruptively switch to a separate instance of the new program version whose state has already been warmed-up. But in contrast to DSU, 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 complex to get right [4].

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

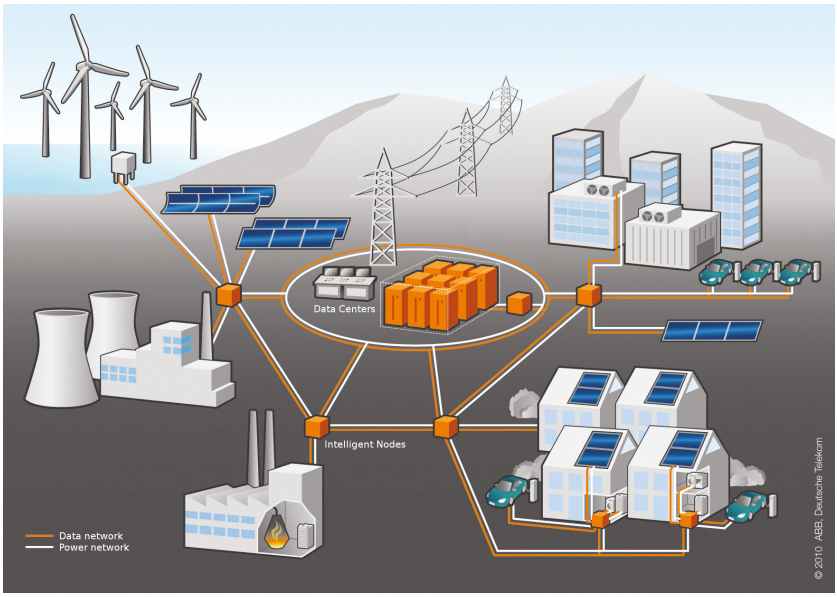

Fig. 1. Power and Computing Components in the Smart Grid (http://www.abb.com/cawp/seitp202/77a7e74be1ea8904c12577050030ab14.aspx)

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 vanilla updates in-memory.

3 Dynamic Software Updating

Systems for Dynamic Software Updates (DSU) may be used by developers when designing new software or to retrofit off-the-shelf applications. They go a step further than application-specific updating approaches by providing generic updating services to an application to update parts of it in-memory [10]. Such DSU systems usually 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—forms of type safety such that no transformations or new program code is able to interact with any unknown/old data structures are usual [16].

DSU systems should require minimal to no programmer intervention to reduce the cost for replacing conventional updates by dynamic ones. For example, no intervention is demonstrated by the DCE VM [18] that aids in Java debugging by applying small changes dynamically. Beyond that, release-level 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. For example, as implemented by JVolve [17], Javelus [8] or Rubah [13] for Java. DSU 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 DSU mechanisms may be less efficient than tuned application-specific solutions (e.g. Facebook's version of memcached), but DSU mechanisms complement such solutions by offering to update even those parts the application-specific ones cannot update.

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 that implements the necessary transformations in-memory. But when considering that such instrumentations and update codes have to be developed for widespread components, as in the context of the Smart Grid, we think, that the benefits outweigh the necessary efforts: all installations could update immediately without additional preparation efforts of the individual installations.

4 Lusagent System

DSU systems perform an atomic swap of an old into a new program in a stop-the-world pause. The transformation of program state is either implemented and finalized *eagerly* (during the stop) or *lazily* (after the stop—while the new program is already running, but before the program is accessing the state).

We present an efficient eager dynamic updating approach for Java. It is based on a novel parallel linear scan of the JVM heap. As previous eager approaches presented in [13,14,17,18], the entire program state is updated in a stop-the-world pause such that no transformation work is left after update which may affect the application performance. In contrast to previous approaches, the heap is sequentially iterated in-between Garbage Collections (GC) instead of traversing its object graph in GC-style. The approach is implemented in our DSU system Lusagent[1]. It is a native plugin to stock Oracle and OpenJDK JVMs for Java 6 to 8 that causes no steady-state overhead by design.

Vanilla applications need an explicit programmer-provided instrumentation with update points [11,13] for enabling dynamic updates but are not modified any further. Update code to transform programs during DSUs can be programmed in our Domain-Specific Language (DSL) for Object Transformers (OT).

[1] Code of Lusagent and evaluated applications: https://github.com/lusagent

4.1 Parts of Lusagent

- *Static analyses of old and new program*
  (1) determine mappings and categorizations for classes and fields (Sects. 4.2)
  (2) determine efficient auto-transformations (Sects. 4.3 and 4.5)
  (3) build type universe for type-safe OT programming/execution (Sect. 4.4)
- *Programming environment for object transformers* (Sects. 4.6 and 4.7)
- *Updating runtime* (Sect. 4.8).

4.2 Class Mapping

Classes in the new program, i.e. in namespace $V_1 = \{t_1^1 \dots t_m^1\}$, are firstly mapped to classes in the old program, i.e. in namespace $V_0 = \{t_1^0 \dots t_n^0\}$. Lusagent by default does so by their name: equally-named classes are mapped. The programmer may add custom mappings to change the defaults or add renamed classes. Let these **mapped classes** be defined by a relation $\mathbf{M_t} := (V_0 \times V_1) \cup (V_1 \times V_0)$. Furthermore, classes in external and standard libraries form namespace C_F (called foreign classes). Let our type universe be closed and consist of V_0, V_1 and C_F, such that $T = V_0 \cup V_1 \cup C_F$ holds. M_t is not allowed to define mappings on C_F.

Fields in old classes are mapped to fields in new classes. Lusagent by default does so by their name and type: equally-named fields are mapped if their types are mapped by M_t. The programmer may add custom mappings to change the defaults or add renamed fields to a class mapping. Definition of **mapped fields** ($\mathbf{M_f}$): let F be the set of all fields; let $\pi(f)$ be the name of field f; let $\tau(f)$ be the type of field f; let $\tau_{def}(f)$ be the defining type of field f; let $\epsilon : T \to F^P \quad \epsilon(t) := \{f \in F | \tau_{def}(f) = t\}$; let $x \preceq y :\Leftrightarrow x$ `instanceof` y; let $\xi : T \to F^P \quad \xi(t) := \bigcup_{s \in T, t \preceq s} \epsilon(s)$; let $x, y \in F$; $(x, y) \in \mathbf{M_f} :\Leftrightarrow \pi(x) = \pi(y) \wedge (\tau(x), \tau(y)) \in M_t \wedge (\tau_{\text{def}}(x), \tau_{\text{def}}(y)) \in M_t$. A mapping between two classes is defined as successful if all fields are mapped between them. Definition of **successful type mappings** ($\mathbf{M_{st}} \subset M_t$): let $(s, t) \in M_t$; $(s, t) \in \mathbf{M_{st}} :\Leftrightarrow \forall\, x \in \xi(s) \exists y \in \xi(t) : (x, y) \in M_f$.

4.3 Class Categorization and Auto-transformation

Afterwards, any classes loaded into the JVM, i.e. in our type universe T, are categorized as follows. Definition of **foreign types** ($\mathbf{C_F} \subset T$): $\mathbf{C_F} := T \backslash (V_0 \cup V_1)$. Definition of **deleted types** ($\mathbf{C_D} \subset V_0$): let $x \in V_0$; $x \in \mathbf{C_D} :\Leftrightarrow \forall\, y \in T : (x, y) \notin M_t$. Definition of **unmodified types** ($\mathbf{C_U} \subset V_0$): let $x \in V_0$; $x \in \mathbf{C_U} :\Leftrightarrow \exists y \in T : (x, y) \in M_{st} \wedge (y, x) \in M_{st}$. Definition of **modified types** ($\mathbf{C_M}$): $C_M := V_0 \backslash (C_U \cup C_D)$. Classes in C_F and C_D and their objects are not transformed as they are not part of the old program or will not be part of the new program. Classes in C_U and C_M and their objects will be transformed.

Objects of classes in C_U can be directly used in the new program as their memory layout does not change. Lusagent copies classes in C_U into the new namespace, but their objects are transformed *in-place* by *patching* their *class-pointers*. Classes in C_M have changed memory layouts which requires to copy their field values. Lusagent copies classes in C_M and their objects *out-of-place*.

The layout of objects may be directly or indirectly affected by modified object fields: the modification of object fields is inherited in stock JVMs as the memory layout of an object contains all values of its own fields but also of all object fields of its superclasses. Figure 2 shows an example type hierarchy. It contains the generic superclass `java.lang.Object`, the subclasses `A` (abstract), `C1`, `C2` and `C3`, and the interfaces `I1`, `I2` and `I3`. Given modified class fields of `A`: this affects `A` only, no other classes would be affected. Given modified object fields of `A`, the objects of `C1`, `C2` and `C3` would all be affected by this modification too.

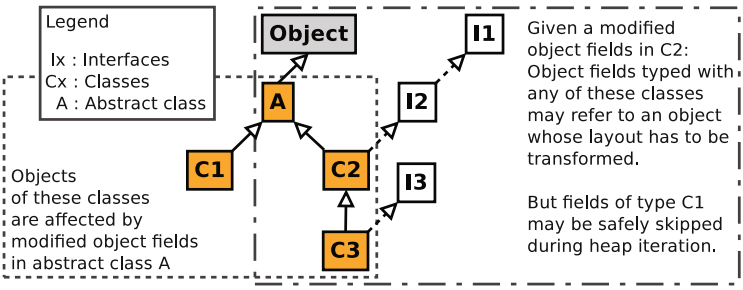

Fig. 2. Example on classes with modified object fields

4.4 Type Universe for Object Transformers

We use a type universe such that programmers can safely (1) access deleted fields in old objects, (2) access old objects of deleted classes, and (3) access new objects which are incompatible to the new fields they are stored in. We leave V_1 as is such that the programmer can safely interact with the new program but we modify V_0 as follows to provide type-safe access in any cases of (1), (2) and (3).

Definition of **foreign-typed fields** ($\mathbf{F_f} \subset F$): let $\tau(f)$ be the type of field f; let $\epsilon : T \to F^P$ $\epsilon(t) := \{f \in F | \tau_{def}(f) = t\}$; let $x \preceq y :\Leftrightarrow x$ `instanceof` y; $\mathbf{F_f} := \{f \in \bigcup_{x \in V_0} \epsilon(x) | \tau(f) \in C_F\}$. Foreign-typed fields in V_0 are not rewritten and always provide type-safe access to old and new objects (assumes that types of new objects are never narrowed by a foreign type which is currently a limitation of our system). Definition of **V_0-typed fields** ($\mathbf{F_0} \subset F$): let $st(t)$ be the set of all subtypes of type t including t; $\mathbf{F_0} := \{f \in \bigcup_{x \in V_0} \epsilon(x) | \tau(f) \in V_0 \wedge \forall t \in sty(\tau(f)) : t \in C_D\}$. Fields in V_0 are V_0-typed fields and therefore not rewritten if the field can only refer to old objects. Definition of **V_1-typed fields** ($\mathbf{F_1} \subset F$): $\mathbf{F_1} := \{f \in \bigcup_{x \in V_0} \epsilon(x) | \tau(f) \in V_0 \wedge \forall t \in M_t(\tau(f)) : t \in sty(\tau(M_f(f)))\}$. Field f in V_0 is V_1-typed and therefore rewritten to its mapped new type $\tau(M_f(f))$ if f can only refer to new objects that are compatible to the new type of the field. Definition of **unknown-typed fields ($\mathbf{F_U} \subset F$)** $\mathbf{F_U} := \{f \in \bigcup_{x \in V_0} \epsilon(x) | \tau(f) \in V_0 \wedge ((\exists t \in M_t(sty(\tau(f))) : t \notin sty(\tau(M_f(f)))) \vee ((\exists t \in sty(\tau(f)) : t \in C_D) \wedge (\exists t \in M_c(sty(\tau(f))) : t \in sty(\tau(M_f(f))))))\}$. Field f in V_0 is unknown-typed and

therefore rewritten to `java.lang.Object` if f can refer to old and new objects, or if f can refer to new objects that are incompatible to its mapped new type $\tau(M_f(f))$. These rewrites allow programmers to access all objects which are incompatible to the typing of the new program safely via fields of old objects.

4.5 Skippable Fields During Heap Iteration

Our heap iteration copies changed objects out-of-place and fixes up any references to these objects. This requires to iterate all fields in all objects and patch values if affected references are found. To minimize costly memory operations when iterating fields, Lusagent uses a novel technique to statically determine which object fields cannot refer to objects copied during an update such that the runtime can safely skip these fields during heap iteration. In Fig. 2, given modified object fields of `C2`, also `C3` is a class with modified object fields. In this case, firstly all fields typed `C2` or `C3` have to be visited. Secondly, all fields typed `I1`, `I2`, `I3`, `A` and `Object` have to be visited too as these are all supertypes of the classes `C2` and `C3`. But in this situation all fields typed `C1` may safely be skipped as they cannot refer to an object of a class with modified object fields.

Lusagent's linear heap scan ensures that all objects on the heap are still reached. In contrast, when performing a heap traversal (from the heap roots), no object fields could be skipped, as parts of the heap would become unreachable.

4.6 Programming Model

The programmer can implement OTs, i.e. Java-like code snippets registered on classes in V_1. The OT is called once per object of that class: it provides local variables `o0` and `o1` to access transformed old and new objects. The OT programmer in general interacts with the new program version only and reads fields of the old version to rescue values. The type universe T is defined as outlined in Sect. 4.4, i.e. types in V_1 are unmodified and types in V_0 have been rewritten for type-safe access. To integrate the classes in V_0 and V_1 into one Java namespace, their names are prefixed by pseudo packages `v0` and `v1`. New fields contain the value that has been transformed into it, except they contain the value `null` if they refer to old or incompatible new objects after transformation. Old fields contain old values if they are still V_0-typed and they contain new values if they are V_1-typed. They can contain old or new values if they are foreign- or unknown-typed (`java.lang.Object`). To prevent additional side-effects between the old and new program, fields of old classes/objects are read-only and old methods are erased.

4.7 Programming Framework

The programmer has to take care of two tasks using the Lusagent IDE as depicted in Fig. 3: firstly, instrument a vanilla application with control-flow transformation code, and secondly, implement OTs on every update to the application.

Control-Flow Instrumentation. The control-flow API has been adopted from Rubah [13]: by replacing the application threads by specific Lusagent threads

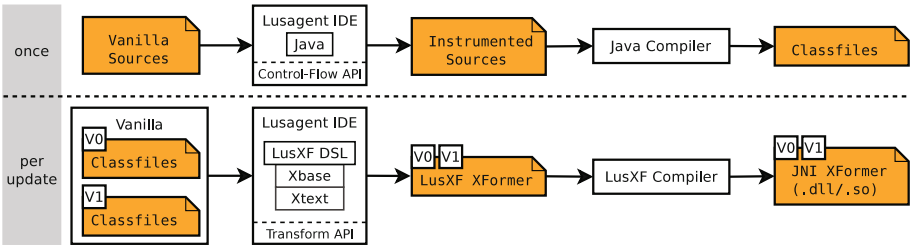

Fig. 3. Lusagent DSU development workflow

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. Furthermore, it can be used by the programmer to ensure timely initiation of an update: update points have to be inserted such that all threads visit any of them frequently. Our API can be used to make sleeps, networking and file I/O interruptable by an update.

Object Transformers. To apply a conventional update dynamically, the programmer uses Lusagent's IDE to specify OTs for it in *LusXF*. LusXF is a Java-like language featuring static type-inference and type-checking based on the Xbase [6] language and the Xtext infrastructure. LusXF provides class and object transformations as first-class citizens, allows to specify their execution sequence and implements our programming model. Its implementation covers an Eclipse-based IDE and a standalone compiler which also features validation of OT-specific semantics. Mappings and auto-transformations can be inspected and stubs for OTs of modified and new classes can be generated by Lusagent.

4.8 Updating Runtime

Updating with Lusagent works as follows. The DSU-instrumented application is launched on the JVM using the Lusagent runtime. The runtime replaces the JVM's default classloader by a Lusagent one that manages all *updatable classes*. When an update is performed, a new program version is loaded using a new Lusagent classloader and afterwards control-flow and state is transformed.

As depicted in Fig. 4a, the transformation consists of analysis components to prepare the heap iteration, the heap iteration with automated transformations itself (phase #1), and the subsequent execution of OTs (phase #2). The analyses are independent of platform specifics (neither JVM nor operating system). While phase #2 depends on JNI only, phase #1 operates directly on the internal JVM data-structures (i.e. class, object and heap layouts) to iterate the entire heap efficiently. This part is kept portable using Oracle's Serviceability Agent (SA) interface [15] which holds symbol locations and definitions of data structures in the JVM itself which is usually used by debugging and profiling tools.

The workflow of these components is depicted in Fig. 4b. Given the classfiles of V_0 and V_1, the static analyses are performed while V_0 is still running.

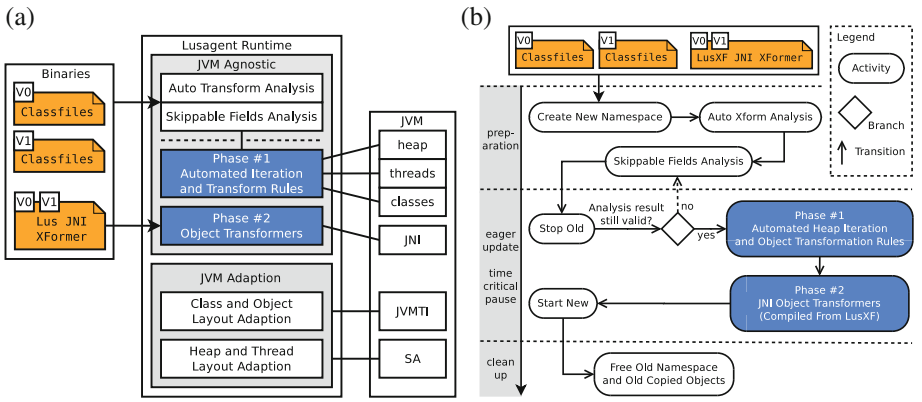

Fig. 4. (**a**) Lusagent runtime architecture Lusagent eager DSU procedure (**b**) Lusagent eager DSU procedure

Afterwards, the JVM is stopped, it is validated that no new classes have been loaded since the static analyses have been finished (which would invalidate the analysis results) and the heap iteration is initiated. The heap is partitioned into n equally-sized chunks, each chunk being iterated by an individual thread. Finally, in phase #2, all OTs for the update are executed: to enable this, Lusagent collects all objects OTs have been registered for during phase #1.

5 Case Study: Dynamically Updating Tango Controls

Among other frameworks, Eclipse NeoSCADA and Tango Controls [7] are two actively maintained open-source SCADA frameworks that are at least partially implemented in Java. Eclipse NeoSCADA is based on isolated Java modules using dedicated classloaders which is currently not supported by Lusagent. Instead, we have used Lusagent to add dynamic updating to the JTango[2] library of Tango Controls. Tango Controls' basic architecture consists of sensors and actuators being connected to a central monitoring and control server. Sensors and actuators are entirely proxied by *device servers*. While the central server is exclusively implemented in C++, device servers can either be implemented in C++, Python or Java. JTango is used to implement device servers in Java.

We illustrate the programming efforts necessary to enable dynamic updating of the JTango library in device servers. So far, device servers had to be restarted to update them. As device servers usually do not keep large data structures in main memory, restarts are be possible within a few seconds. They are not instant though, as the Java runtime environment and the JTango boot-up takes several seconds which we have measured on commodity desktop hardware. This procedure is speed up by Lusagent resulting in considerably less disruption: we

[2] https://github.com/tango-controls/JTango

have measured only few milliseconds to update device servers which have been continuously reporting sensor values at about 20 Hz to a central server.

5.1 Control-Flow Instrumentation

JTango does not use any own application threads but is exclusively driven by events from the JacORB CORBA [3] library that interconnects it to the Tango Controls server. We have instrumented the request processing threads in recent JacORB releases with Lusagent: v3.6, v3.7 (both released in 2015) and v3.8 (released in 2016). These releases are also used by recent JTango releases.

```
1 public class RequestProcessor extends LusThread /* before: Thread */
2              implements InvocationContext, Configurable {
3     /* ... */
4     public void run() {
5         while (true) {
6             Updater.updatePoint("request-processor"); /* added line */
7             /* ... */ wait(); /* ... */ process(); /* ... */
8         } } }
```

Listing 1: Lusagent Instrumentation in `org.jacorb.poa.RequestProcessor`

We have a single patch for the three JacORB releases. It is depicted in simplified form in Listing 1: to allow each request processor to reach an update point, its standard Java thread has been replaced by a Lusagent thread and an update point has been added into the thread's event-loop. Now, JTango can be updated in-between processing of any two JacORB events. As JTango is not performing long-running operations when processing JacORB events, no code has been added into JTango to interrupt operations and reach update points. JTango itself has not been instrumented: its vanilla releases can be used directly.

5.2 Object Transformers

We have looked at the update code necessary to update 5 subsequent recent minor releases of JTango dynamically: from v9.0.8 to v9.0.11 and v9.1.0 (all released in 2016). Table 1 lists the Lines of Code (LoC) changed between subsequent vanilla releases and lists the LoC of Lusagent OTs we programmed to implement the subsequent releases as dynamic updates. The updates v9.0.9 → v9.0.10 and v9.0.10 → v9.0.11 are fully covered by Lusagent's automatic transformations. The update v9.0.8 → v9.0.9 requires an OT for objects of 2 classes in which a renamed object field and a class field that has become and object field are initialized by copying over the values of their old counterparts. The update v9.0.11 → v9.1.0 requires an OT for objects of 3 classes in which newly added object fields are initialized analogously to the new object constructors.

Table 1. Programming efforts with Lusagent for updating JTango

Release		Instrumentation	Vanilla update	OTs
Version	LoC	+/− LoC	+/− LoC	LoC
JTango				
9.0.8	16514	0, 0	10681	12
9.0.9	16532	Same	12899	0
9.0.10	16558	Same	2926	0
9.0.11	16563	Same	166102	18
9.1.0	16628	Same		

6 Evaluation

Besides the previous case study, we have evaluated the *programming efforts* with Lusagent and its *updating performance* on 7 Java server applications: 2 SQL databases (H2 and HSQLDB), the Voldemort key-value-store, CrossFTP, the Moquette MQTT broker, JavaEmailServer and the Glowstone game-server. We measure pauses for updating instances of H2, HSQLDB and Voldemort with in-memory data-stores which have been warmed-up to at least 4 GiB and while benchmarks on them are executed. The pauses of Lusagent consist of synchronizing all control-flows in the DSU runtime and the subsequent heap iteration. For baselining our performance studies, we measure durations of updates with the Rubah [13] DSU system on H2 and Voldemort using equivalent control-flow instrumentations. Furthermore, we measure durations of Garbage Collections (parallel full GC in standard configuration). To illustrate control-flow migration performance with many threads, CrossFTP is stressed by many clients.

6.1 Programming Efforts

Table 2 shows the Lines of Code (LoC) to instrument the applications with Lusagent, the LoC affected by the vanilla updates and the LoC required to implement these vanilla updates by OTs in LusXF. Our experiences confirm that the control-flow instrumentation proposed by Rubah is manageable and a one-time effort. We have extended the instrumentation API by a programming pattern we call *update barriers*: dynamically generated proxies to synchronize external threads with the DSU that cannot be replaced by our LusThreads. This resulted in concise instrumentation solutions for Moquette and HSQLDB.

The first update to Moquette and the last update to HSQLDB are quite complex: the OTs contain many `instanceof` tests to determine the situation in the program state at time-of-update and follow various old fields to access proper values for transformation. Our type-checked language demonstrates its feasibility particularly on these updates as the code is concise and its typing apparent.

Table 2. Programming efforts with Lusagent

Release		Instr.	Van. update	OTs	Release		Instr.	V. update	OTs
Version	LoC	+/– LoC	+/– LoC	LoC	Ver	LoC	+/– LoC	+/– LoC	LoC
H2					*Moquette*				
1.2.121	78738	331, 38			0.7	8468	19, 3		
1.2.122	79185	Same	1184, 610	21	0.8	9691	102, 8	3109, 2629	267
1.2.123	79274	333, 34	2188, 1911	12	0.8.1	9670	11, 1	170, 307	33
HSQLDB					*CrossFTP*				
2.3.0	168130	196, 147			1.07	18082	408, 247		
2.3.1	168212	196, 139	285, 195	4	1.08	18109	Same	97, 46	0
2.3.2	168563	Same	2666, 1871	34	1.09	18174	420, 233	718, 702	33
2.3.3	167638	Same	10526, 11970	201	1.11	18468	Same	615, 189	28
Voldemort					*JavaEmailServer*				
1.5.3	58474	69, 13			1.3.3	2429	262, 85		
1.5.4	58497	Same	82, 24	6	1.3.4	2508	Same	137, 17	0
Glowstone					1.4	2590	Same	134, 7	11
1.8.4	45781	41, 10							

6.2 Pauses for Updating

We have measured the pauses for updating with Rubah and Lusagent on H2, HSQLDB, Voldemort and CrossFTP. The results are listed in Table 3. Both, Lusagent with and without its field skipping technique, outperform parallel full Garbage Collection (GC). On average, the skipping technique improves

Table 3. Pauses (in secs) for updating large memory applications with Rubah and Lusagent. (avg. & std. dev. from 15 samples.)

Update	Mem scale	Lusagent no skip	Lusagent with skip	GC	Eager Rubah	Lazy Rubah
		$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
H2						
121 → 122	**64**[a]	3.6 ± 0.04	3.6 ± 0.21	4.6 ± 0.05	21.9 ± 3.89	0.6 ± 0.09
122 → 123	**64**	3.7 ± 0.06	3.6 ± 0.05	4.6 ± 0.04	25.0 ± 0.50	0.7 ± 0.15
HSQLDB						
2.3.0 → 1	**200**[a]	1.5 ± 0.01	1.3 ± 0.01	5.0 ± 0.16	–[b]	–[b]
2.3.1 → 2	**200**	1.5 ± 0.01	1.3 ± 0.01	5.0 ± 0.23	–	–
2.3.2 → 3	**200**	1.8 ± 0.02	1.7 ± 0.02	5.2 ± 0.18	–	–
Voldemort						
v1.5.3 → 4	**M5**[a]	1.0 ± 0.03	0.9 ± 0.02	4.1 ± 0.03	24.6 ± 1.47	0.3 ± 0.03
	M10	1.9 ± 0.02	1.7 ± 0.02	7.9 ± 0.08	80.6 ± 5.52	0.4 ± 0.04
	M15	2.8 ± 0.04	2.5 ± 0.03	14.4 ± 0.48	204.4 ± 12.08	0.4 ± 0.06

[a] *64* is a scale factor for the Dacapo TPC-C benchmark; *200* is a scale factor for the HSQLDB TPC-B benchmark; *M5* is a scale factor for the Voldemort benchmark. All three factors result in processes of >4 GiB unshared memory given by Unique Set Size (USS) on Linux for x64.

[b] We have not yet ported our Object Transformers for HSQLDB to Rubah.

performance by 9.2%. The performance improvements differ significantly between the databases: 1.4% for H2, 11.4% for Voldemort, 15.4% for HSQLDB.

We expect the type hierarchy in the software to be a relevant factor for the performance differences. But as the update pause is dominated by the copying overhead for out-of-place object transformations after heap iteration, this performance improvement during heap iteration becomes less evident with larger updates. We preliminary conclude that our field skipping improves linear heap scanning performance significantly (depending on the type hierarchy) and improves updates significantly that affect only a moderate fraction of heap objects (Fig. 5).

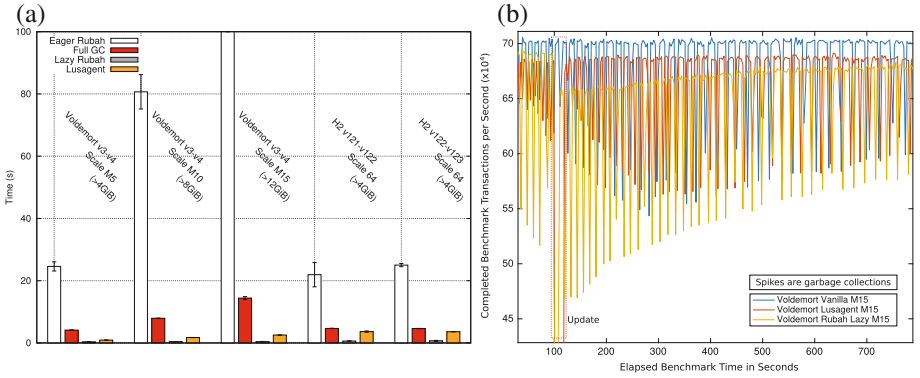

Fig. 5. (**a**) Update pauses compared to GC. (avg. & std. dev. from 15 samples) (**b**) Benchmark performance on update. Vanilla application is baseline.

Table 4. Pauses (in secs) for updating many threads with Rubah and Lusagent. (avg. & std. dev. from 30 samples)

Update	Threads	Eager Rubah	Lusagent
	#	$\mu \pm \sigma$	$\mu \pm \sigma$
CrossFTP			
v1.07 → v1.08	**64**	0.27 ± 0.03	0.18 ± 0.00
	128	0.32 ± 0.05	0.30 ± 0.00
	192	0.38 ± 0.06	0.35 ± 0.00
	256	0.45 ± 0.04	0.43 ± 0.01
v1.08 → v1.09	**128**	0.27 ± 0.05	0.30 ± 0.00
	256	0.45 ± 0.04	0.43 ± 0.01
v1.09 → v1.11	**128**	0.27 ± 0.04	0.28 ± 0.00
	256	0.46 ± 0.08	0.42 ± 0.01

Figure 4a also displays the pause durations of the DSUs on H2 and Voldemort for Lusagent and Rubah (parallel eager and lazy). The three Voldemort benchmarks demonstrate that pause times of GC, Lusagent and eager Rubah scale proportionally to memory size. In contrast, lazy Rubah causes a constant pause. Pauses induced by lazy Rubah and Lusagent both outperform full GC.

Table 4 depicts the pauses induced by Rubah and Lusagent on CrossFTP. In this benchmark, memory of the application is only in the range of a several MiB but the DSU instrumentation has to wait for one thread per connected ftp client before updating. Both systems cause similar pauses of <1s which scale by the number of threads. Lusagent exposes a significantly lower variance.

Finally, except for the warmup of the JVM's Just-In-Time Compiler (JIT), Lusagent does not induce short-term overhead after updating, as exemplary depicted in Fig. 4b for the Voldemort benchmark at scale M15.

7 Conclusion

In this work, we have discussed the need for low-disruptive and timely hot-fixing and updating of software components in the Smart Grid infrastructure which is becoming an open system and will be largely standardized for interoperability. Dynamic Software Updating is an approach that tries to enable such updates for highly available software in general. As Smart Grid components are primarily based on software, DSU allows to immediately update them whenever security issues are disclosed that would otherwise put all installations in the system at risk. This particularly becomes an issue if many openly accessible components share the same implementations, e.g. a secure communications library in a distributed system such as the Advanced Metering Infrastructure.

We have presented Lusagent: a system for eager dynamic software updating of Java applications on release-level which has been implemented as a native plugin to industry-grade JVMs. It uses a novel heap iteration algorithm to efficiently update an entire Java application in-memory. It allows to perform hot-fixes, requiring almost no additional programming effort, and it features release-level updates, which can be flexibly programmed using object transformation code.

We have illustrated our updating approach in the context of the Smart Grid in a case study performing 5 subsequent release-level updates on JTango device servers in the Tango Controls SCADA framework. The programming efforts were considerably low: only an event-driven communications library of JTango had to be instrumented to enable dynamic updating of JTango and its device servers.

We have furthermore demonstrated the feasibility of our updating approach by performing 13 release-level DSUs, ranging from small to quite large ones, on 7 server applications. Furthermore, we studied its efficiency in updating 3 database servers with large transient memory and 1 highly multi-threaded file server, showing that update pauses are significantly shorter than the parallel full garbage collections which usually are part of the normal application runtime.

References

1. Ajmani, S., Liskov, B., Shrira, L.: Modular software upgrades for distributed systems. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 452–476. Springer, Heidelberg (2006). doi:10.1007/11785477_26
2. Brewer, E.A.: Lessons from giant-scale services. IEEE Internet Comput. **5**(4), 46–55 (2001)
3. Brose, G.: JacORB: Implementation and Design of a Java ORB, pp. 143–154. Chapman & Hall, Cottbus (1997)
4. Dumitraş, T., Narasimhan, P.: Why do upgrades fail and what can we do about it? In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 349–372. Springer, Heidelberg (2009). doi:10.1007/978-3-642-10445-9_18
5. Durumeric, Z., Kasten, J., Adrian, D., Halderman, J.A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., Paxson, V.: The matter of heartbleed. In: Proceedings of the 2014 Conference on Internet Measurement Conference (IMC 2014), pp. 475–488. ACM, New York (2014)
6. Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., Hanus, M.: Xbase: implementing domain-specific languages for Java. In: Proceedings of the 11th International Conference on Generative Programming and Component Engineering (GPCE 2012). ACM, New York (2012)
7. Götz, A., Taurel, E., et al.: TANGO V8-Another turbo charged major release. In: Proceedings of ICALEPCS, San Francisco (2013)
8. Gu, T., Cao, C., Xu, C., Ma, X., Zhang, L., Lu, J.: Javelus: a low disruptive approach to dynamic software updates. In: Proceedings of the 2012 19th Asia-Pacific Software Engineering Conference (APSEC 2012), vol. 01, pp. 527–536. IEEE Computer Society, Washington, DC (2012)
9. Hayden, C.M., Smith, E.K., Hardisty, E.A., Hicks, M., Foster, J.S.: Evaluating dynamic software update safety using systematic testing. IEEE Trans. Softw. Eng. **38**(6), 1340–1354 (2012)
10. Hicks, M., Nettles, S.: Dynamic software updating. ACM Trans. Program. Lang. Syst. **27**(6), 1049–1096 (2005)
11. Neamtiu, I., Hicks, M., Stoyle, G., Oriol, M.: Practical dynamic software updating for C. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2006), pp. 72–83. ACM, New York (2006)
12. Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H.C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., Venkataramani, V.: Scaling memcache at Facebook. In: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI 2013), pp. 385–398. USENIX Association, Berkeley (2013)
13. Pina, L., Veiga, L., Hicks, M.: Rubah: DSU for Java on a stock JVM. In: Proceedings of the 2014 ACM Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA 2014). ACM, New York (2014)
14. Pukall, M., Kästner, C., Cazzola, W., Götz, S., Grebhahn, A., Schröter, R., Saake, G.: JavAdaptor-flexible runtime updates of Java applications. Softw. Pract. Exp. **43**(2), 153–185 (2013)
15. Russell, K., Bak, L.: The hotspotTM serviceability agent: an out-of-process high level debugger for a JavaTM virtual machine. In: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium (JVM 2001), vol. 1, p. 16. USENIX Association, Berkeley (2001)

16. Stoyle, G., Hicks, M., Bierman, G., Sewell, P., Neamtiu, I.: Mutatis mutandis: safe and predictable dynamic software updating. ACM Trans. Program. Lang. Syst. **29**(4), 22 (2007)
17. Subramanian, S., Hicks, M., McKinley, K.S.: Dynamic software updates: a VM-centric approach. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009), pp. 1–12. ACM, New York (2009)
18. Würthinger, T., Wimmer, C., Stadler, L.: Unrestricted and safe dynamic code evolution for Java. Sci. Comput. Program. **78**(5), 481–498 (2013)