Updating Service-based Software Systems in Air-Gapped Environments*

Oleksandr Shabelnyk[1,2], Pantelis A. Frangoudis[2], Schahram Dustdar[2], and Christos Tsigkanos[2]

[1]Preparatory Commission for the Comprehensive Nuclear-Test-Ban Treaty Organization, Austria
[2]Distributed Systems Group, TU Wien, Austria

Abstract. Contemporary component-based systems often manifest themselves as service-based architectures, where a central activity is management of their software updates. However, stringent security constraints in mission-critical settings often impose compulsory network isolation among systems, also known as air-gap; a prevalent choice in different sectors including private, public or governmental organizations. This raises several issues involving updates, stemming from the fact that controlling the update procedure of a distributed service-based system centrally and remotely is precluded by network isolation policies. A dedicated software architecture is thus required, where key themes are dependability of the update process, interoperability with respect to the software supported and auditability regarding update actions previously performed. We adopt an architectural viewpoint and present a technical framework for updating service-based systems in air-gapped environments. We describe the particularities of the domain characterized by network isolation and provide suitable notations for service versions, whereupon satisfiability is leveraged for dependency resolution; those are situated within an overall architectural design. Finally, we evaluate the proposed framework over a realistic case study of an international organization, and assess the performance of the dependency resolution procedures for practical problem sizes.

Keywords: Software Updates · Air-gapped Environments · Service-based Architectures
Disclaimer: The views expressed herein are those of the authors and do not necessarily reflect the views of the CTBTO Preparatory Commission.

1 Introduction

Contemporary software architectures reflect decades-long software engineering research and practice, where separation of concerns with respect to the wide-ranging functionality available throughout a software system is strongly emphasized. This leads to systems formed via composition of loosely coupled independent software components, which are also often distributed. The trend towards

* Research partially supported by Austrian Science Foundation (FWF) project M 2778-N "EDENSPACE".

breaking down software into increasingly smaller pieces introduces numerous advantages, however, it increases overall system complexity, including over its maintenance and managed evolution. This component-based view has culminated in service-orientation, where service-oriented architectures (SOA) have seen wide applicability.

Software systems however are not static, but rather *evolve*, undergoing continual change, with software maintenance thus constituting a major activity [1]. This is evident also in service-oriented component-based architectures, where software is designed, developed and maintained by different teams in often agile processes. As such, software updates are a central theme, something exacerbated in mission-critical settings in highly regulated, mission critical environments where stringent security constraints impose compulsory network isolation among distributed systems, also known as *air-gap*. Even though network isolation does not counter all security concerns [2–4], such a design is a prevalent choice in different sectors involving critical systems, be it within private, public or governmental organizations. Air-gap isolation generally imposes challenges in the lifecycle management of service-based software systems, the lack of constant availability of resources being a major issue, and is in contrast with the spirit of modern DevOps practices [5]. In working environments where an air-gap is in place, the lack of Internet connectivity also has a negative impact on productivity [6]. Challenges arise especially when there is a need to initially provision and later update distributed component-based software systems – an update of a software component may introduce breaking changes to other dependents. Naturally, software updates, their modelling and dependency resolution are problems that have been treated by the community extensively and in several forms [7–10].

However, updating air-gapped systems raises several issues from a software architecture perspective, especially given the overall mission-critical setting; those include: (i) the configuration of components produced to update the system should be verifiably correct, since there is significant cost-to-repair for incorrect updates, (ii) service-based architectures entail containerized services, with support of different runtime environments, and (iii) update actions should be recorded in a traceable manner, in order to support auditability and regulatory compliance. As such, we adopt an architectural viewpoint and present a technical framework for updating distributed software systems in air-gapped environments. Our main contributions are as follows:

- We detail the domain characterized by network isolation and identify requirements, update workflow and modelling notations for service versions;
- We leverage satisfiability for dependency resolution, providing alternative strategies with correctness guarantees and address the trade-off between their execution time and resolution quality;
- We describe an architectural design to instrument updates for air-gapped service-based systems, which we implement end-to-end, and finally
- We evaluate the proposed framework over a realistic case of an air-gapped update elicited from the Comprehensive Nuclear-Test-Ban Treaty Organi-

zation (CTBTO[1]). We further assess the performance of the dependency resolution procedures for practical problem sizes.

The rest of this paper is structured as follows. Sec. 2 gives an overview of the proposed approach along with the challenges brought by the air-gapped setting. Sec. 3 describes an architecture and workflow instrumenting air-gapped updates, while Sec. 4 elaborates on characteristic dependency resolution strategies. Sec. 5 provides an assessment over a case study along with a performance evaluation. Related work is considered in Sec. 6, and Sec. 7 concludes the paper.

2 Updating Service-Based Air-Gapped Systems

An *air-gap* is a security measure employed to ensure that a computer system is physically network-isolated from others, such as the Internet or other local area networks. The air-gap design may manifest in computers having no network interfaces to others, while residing in a physically isolated location. This is because a network – often used to update software – represents a security vulnerability or regulatory violation. To transfer data (or programs) between the network-connected world and air-gapped systems, one typically uses a removable physical medium such as a hard drive, while access is regulated and controlled [11]. The key concept is that an air-gapped system can generally be regarded as closed in terms of data and software, and unable to be accessed from the outside world. However, this has implications regarding contemporary systems, which may need to be upgraded as part of software maintenance activities. Although existing package management solutions can be theoretically used (e.g., by storing an entire repository on a physical medium), this may be inefficient and not readily applicable in a service-based setting; repositories can be sizable and snapshotting to removable media may be impractical or even infeasible.

Figure 1 illustrates a birds-eye view of the domain and proposed approach. On the problem domain (left part of Fig. 1) a series of air-gapped systems host software services, each having some version. Those comprise the service version configuration state of each air-gapped system, which is assumed to be known or adequately communicated. Software development takes place off-site, and services may need to be updated. Services – as software components – have dependencies, specified at development time. To perform an update on an air-gapped system (right part of Fig. 1), the process entails resolution of service dependencies per air-gapped system, building a valid service configuration taking into account its current configuration state, and pulling of appropriate artifacts (such as containers) from development repositories, storing them in a removable physical medium. Subsequently, the air-gapped system is visited, the service configuration is verified against the local state and the update is performed; services are then provisioned accordingly based on execution environments.

[1] CTBTO Preparatory Commission, www.ctbto.org.

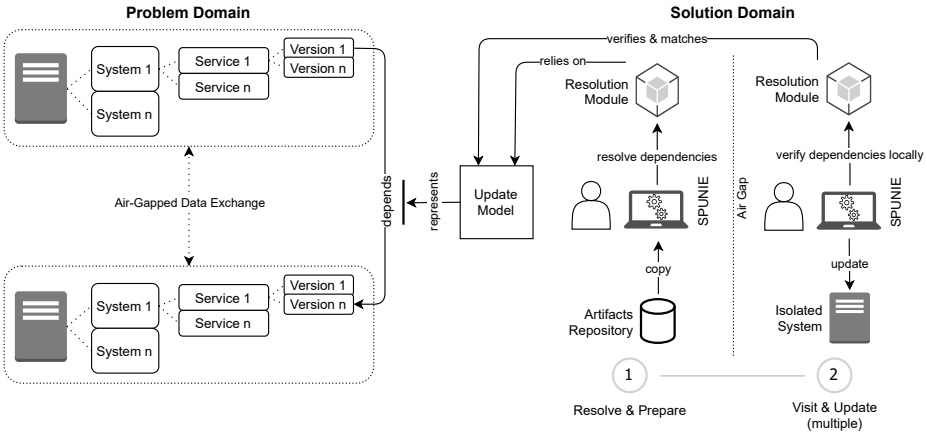


Fig. 1: Updating service-based air-gapped systems – Overview.

3 Architecture for Instrumenting Air-Gapped Updates

As evident from Fig. 1, updating service-based air-gapped systems requires a dedicated software architecture, capable of coping with the particularities of the domain, workflow, and execution environments. To this end, this section first outlines design requirements a software architecture for instrumenting air-gapped updates should fulfil, before presenting its materialization and accompanying update workflow.

3.1 Design Requirements

As air-gapped systems are typically employed in mission-critical settings, key themes regarding the design of a software architecture concern dependability of the update process performed, interoperability with respect to the software supported and auditability regarding update actions performed. Specifically:

DR1 Dependability & Verifiability: The configuration of components produced in order to update a system should be verifiably correct. Given the criticality of the domain and the network isolation, there is significant cost-to-repair incorrect updates, something exacerbated by the fact that a physical visit to the air-gapped site is required to apply the update. Moreover, the air-gapped system should be able to verify locally that the update configuration to be applied is correct before installing (recall Fig. 1), as rollbacks induce further cost.

DR2 Interoperability & Extensibility: Since in service-based systems, software is designed, developed and maintained by different teams in often agile processes, loose coupling is desired, in practice realized by pluggable (and interchangeable) components. Functional blocks should manifest themselves as containerized services, different runtime environments of which may be

supported (e.g., Docker). Components should expose interfaces, in order to be agnostic of underlying programming languages and other internals.

DR3 Traceability & Auditability: User and system access and update actions should be recorded in a traceable manner, in order to both aid the development and deployment lifecycle and to ensure regulatory compliance. This is key to support required forensic processes for external regulators or inspectors of the update workflow, that are typically in place in mission-critical administrative domains, and is a fundamental requirement in our particular case study which we present in Sec. 5.1.

3.2 Functional Components

Figure 2 illustrates functional components of the architecture we advocate for updating air-gapped systems. The architecture is itself service-based, in order to support interoperability as per DR2; components address separation of concerns such that their development can be supported by different teams or processes, something which is the typical case in large organizations involved with mission-critical systems. The Gateway provides a Web UI and is responsible for routing (authenticated) users' requests to the right service, making use of a Service Registry which records and discovers available services available to consist an update. Additionally, a collection of air-gapped systems is maintained along with their current service version configuration – this amounts to book-keeping of their remote state, and may be instrumented in case-specific ways which are out of the scope of the present paper. Thereupon, the Dependency Resolver service is responsible for resolving version dependencies according to DR1, yielding update configurations that are verifiably correct, to be shipped to target air-gapped systems. Upon the physical visit, the update configuration is validated against the service configuration already present in the system.

The rightmost part of Fig. 2 illustrates functional components outside the core architecture, namely interaction with different execution environments the system may employ – Docker is assumed to be the main service containerization technology, but mobile application containers or images may be also be included. In practice, Docker Swarm implements an interface to ensure loose coupling and avoid vendor lock-in (DR2). The depicted artifact repositories provide, for example, container images. Finally, logging and monitoring facilities address traceability and auditability according to DR3; to this end, all user actions, update operations and system interactions are recorded. Concrete technological choices for implementation of the functional components are illustrated in grey in Fig. 2; those represent contemporary technologies that can be adopted for implementation.

3.3 Update Workflow

Given the architecture of Fig. 2 and upon a user's request, the update workflow for an air-gapped system is comprised of the following steps, which are sufficiently recorded per session, in order to ensure traceability and auditability:

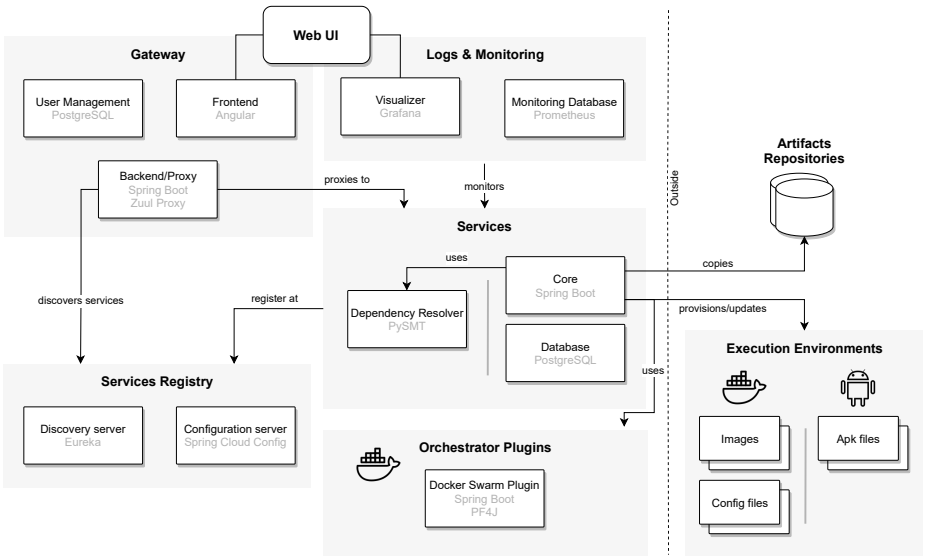


Fig. 2: Service-based architecture supporting updates of air-gapped systems.

1. The user responsible for the preparation and deployment of an update (e.g., the designated release engineer) selects an available system and defines which of its services should be updated and to which versions (DR3).
2. The Dependency Resolver yields a satisfiable combination of service versions (ref. DR1).
3. The Core service, accordingly invoking the Dependency Resolver prepares the corresponding artifacts by copying them to a physical medium from external repositories, and creates a Deployment Plan.
4. Given the Deployment Plan produced, the target air-gapped system is physically visited, the update configuration is verified against the local service configuration (as per DR1), and the update is applied; the services can be provisioned in the target host.

4 Service Dependency Resolution

At the heart of the update workflow lies a dependency resolution step. In this section we first discuss how to model service versions, including how the problem of resolving their dependencies can be formulated in order to enable its automatic resolution. This regards an implementation of the critical *Dependency Resolver* component of Fig. 2. As this component amounts to a black box, we outline a characteristic manner in which it can be implemented; recent literature on the topic can further extend it to cover more specialized cases.

4.1 Problem Formulation

Dependency resolution entails finding the right combination of software components while preserving certain constraints such as version compatibility; the un-

derlying problem is NP-Complete [12]. Dependency resolution is particularly pertinent in component-based software architectures, where it is manifested in various forms; in free and open source software for instance, the components are often called packages and are handled by package managers. Although package managers differ in how they handle dependency resolution, dependencies/packages usually have certain common traits [8, 9]: i) name and version which are uniquely identifiable, ii) dependencies to other components (also called *positive* requirements), iii) conflicts expressing absence of certain other components (also called *negative* requirements), and iv) features, identifiers of "virtual" components that may be used to satisfy dependencies of other components.

For our service-based setting, we adopt semantic versioning [13], where three numbers separated by a dot are used, e.g. "2.4.1." The first number indicates a *major* release typically introducing breaking changes. The middle number reflects a *minor* version change, signalling that new functionality has been added but with full backwards compatibility preserved. The last number stands for *patch or micro* changes which indicates bug fixes; patch changes are also fully backwards compatible. Dietrich et al. [14] propose a comprehensive classification of version constraints, including describing fixed, soft, variable dependencies, or typical version semantics such as at least, at most, or latest; the approach we advocate can be further extended to support them.

Resolving which versions are needed to perform a valid update of a system has been approached in several ways including boolean satisfiability (SAT), Mixed Integer Linear Programming (MILP), Answer Set Programming (ASP), or Quantified Boolean Formulae (QBF) [9, 8, 15, 16]. A typical way is working within Satisfiability Modulo Theories (SMT), where solving consists in deciding the satisfiability of a first-order formula with unknowns and relations lying in certain theories; formulas are constructed over usual boolean operators, quantifiers over finite sets, as well as integer linear arithmetic operators. In the following, we informally describe the construction of such a formula which integrates known facts about a system, along with certain constraints; the interested reader can consult technical literature on the topic [17]. The intuition is as follows. Facts capture the current state of the system as well as its desired state; for example, consider that services A and B with versions "1.3" and "18.2.1" respectively are installed, and A is sought to be upgraded to version "1.4." Constraints encode dependencies between services; for example, service B of version "18.2.*" requires service C of fixed version "1.2.5". In essence, given (i) a configuration of already installed services of certain versions, (ii) a service of a newer version which is sought to be updated, (iii) dependency relations between services, and (iv) a set of available versions per each service, we seek to identify which versions of services are required to perform a valid update. A valid update is one that satisfies all service dependencies, and transitions the system to an upgraded (resp. for some service) state. Specifically, the components of the problem regard the desired state, dependencies and the current service configuration:

– **upgrade-versions**: One or more versions of different services that should be upgraded. For example, the user may desire to upgrade service A to version

n and service B to version m due to newly introduced features in the first one and a recently fixed bug in the second.
- **available-versions**: Versions of services that are available to be installed, sourced for instance from development repositories.
- **dependencies**: A dependency of a service to versions of another, for example service A with patch constraint "7.5.*" depends on service B with minor constraint "10.*.*."
- **installed-versions**: Versions of services which are already installed in the system; this is the current state of the system.

We abstain from providing a formal representation; existence of versions amounts to assignment to variables (e.g., within linear arithmetic in SMT), while dependencies consist of implications (e.g., selection of service A version 2.1 requires B version 3.4). As such, informally, the dependency resolution problem amounts to: "Given service versions the user wants to update to, versions available and services already installed, derive a set of service versions that adhere to version dependencies, if such a set exists".

Algorithm 1 ALL-VER

Input upgrade version(s), dependency constraint(s), available version(s) per component, installed version(s)
Output set of all valid versions per component

```
1:  /* Construct domain */
2:  domain ← dependencies
3:  /* Construct facts */
4:  hardFacts ←upgrade-versions,exactly-one
5:  hardFacts ←+ available-versions
6:  softFacts ← installed-versions
7:  /* Construct problem */
8:  problem ← domain, hardFacts, softFacts
9:  /* Iterate over service versions */
10: for i ← 1, versions do
11:     /* pr is partial result */
12:     for pr ← solve(problem,i) do
13:         /* add partial to results */
14:         result.add(pr)
15:         /* negate partial result */
16:         problem.not(pr)
17:     end for
18: end for
19: Return result
```

Algorithm 2 MAX-VER

Input upgrade version(s), dependency constraint(s), available version(s) per component, installed versions
Output set of maximum versions per component

```
1:  /* Construct domain */
2:  domain ← dependencies
3:  /* Construct facts */
4:  hardFacts ←upgrade-versions,exactly-one
5:  hardFacts ←+ available-versions
6:  softFacts ← installed-versions
7:  /* Construct problem */
8:  problem ← domain, hardFacts, softFacts
9:  components ← available version(s)
10: /* Iterate over components */
11: for i ← 1, components do
12:     /* Apply ALL-VER to component i */
13:     versions ← all_ver(problem, i)
14:     max_ver ← max(versions)
15:     result.add(max_ver)
16:     /* Version holds for component i */
17:     problem.add(max_ver, i)
18: end for
19: Return result
```

4.2 Dependency Resolution Strategies

The generic setting previously presented amounts to a generic problem formulation; thereupon, one can build further strategies for dependency resolution, with different objectives, beyond merely being satisfiable. In particular, we advocate two reference strategies: the MAX-VER strategy determines the most recent versions that constitute a valid update, while ALL-VER discovers all feasible versions. The latter is intended to enable some other selection criterion – for

example, selecting a version which has been more widely deployed (thus perhaps more bug-free) – but naturally, at a higher computational cost. We note that the strategies describe the general process – optimizations (taking into account solver particularities, for example), are further possible.

The strategies employed are illustrated in Algorithms 1 and 2. Both take as input the desired service version(s), the dependency constraints, the available versions per component, and the current installed configuration. The ALL-VER strategy consists of three steps: (i) construct the domain using dependency constraint(s), (ii) construct facts using available and installed version(s), (iii) find solutions. The latter step entails considering each version variable, querying the solver for a partial model (a set of valid results), storing the results, negating the partial model, and querying the solver again until all partial models are delivered. Conversely, the MAX-VER strategy identifies first all versions for the current service, subsequently obtains the partial model and negates the intermediary results. Thereupon, all valid options for the component are found, and the maximum (most recent) is selected. The iteration continues – each time the problem is increasingly constrained.

5 Evaluation

To provide concrete support for our air-gapped update framework, we realized a prototypical end-to-end system; technological choices made for implementation of the functional components are the ones illustrated in grey in Fig. 2. Thereupon, we evaluate our approach over a characteristic scenario elicited from the Comprehensive Nuclear-Test-Ban Treaty (CTBT) Organization. Subsequently, we assess performance aspects. We conclude with a discussion. Our evaluation goals are two-fold; we seek to investigate (i) *applicability* of the proposed solution, in terms that the architecture and system used are able to be used in practice (Sec. 5.1), and (ii) *performance* in realistic settings (Sec. 5.2). The former entails considering a realistic scenario in a mission-critical setting, where the workflow and architecture advocated are employed end-to-end. The latter requires assessing dependency resolution over typical problem sizes.

5.1 Applicability

The Comprehensive Nuclear-Test-Ban Treaty Organization (CTBTO) is an international body tasked with verifying the ban on nuclear tests, operating a worldwide monitoring system and, after the treaty's entry into force, conducting On-Site Inspections (OSIs). Being the final verification measure under the treaty, the purpose of an On-Site Inspection is to collect evidence on whether or not a nuclear explosion has been carried out. The inspection team consists of scientific experts, while the strict regulatory compliance framework in place enforces stringent security requirements on data handling, which imposes strict air-gapped isolation upon all software systems involved. The scenario we consider was elicited via interviews from key stakeholders, and concerns a characteristic case where air-gapped update is required.

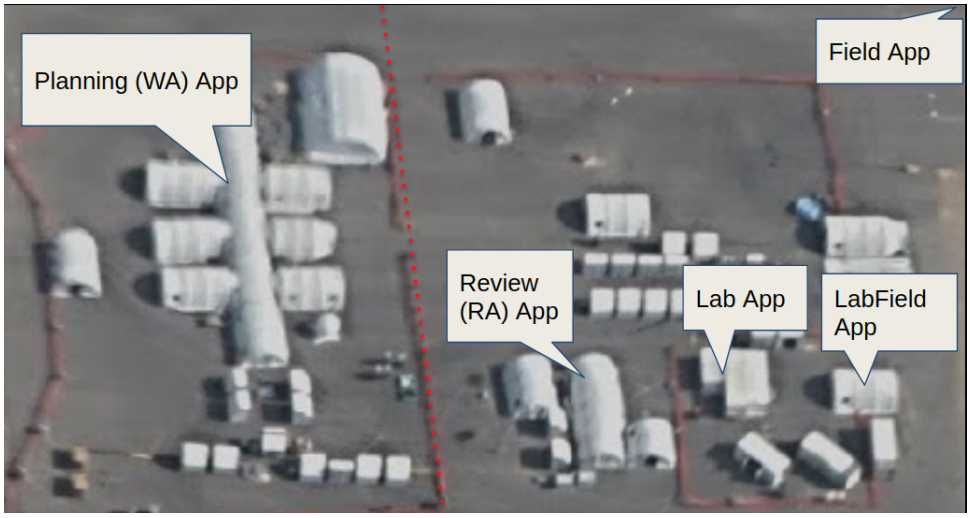


Fig. 3: Fragment of an On-Site-Inspection Exercise illustrating air-gapped sites hosting applications (Image credit: CTBTO OSI Division).

When an On-Site Inspection is dispatched to a remote location, all software systems are carefully prepared, configured and shipped fully provisioned during the launch phase. Fig. 3 illustrates an airborne photograph of this setup during an Integrated Field Exercise. The shown software systems help the inspection team to conduct an inspection by: planning field missions (Planning WA App), collecting data and metadata in the field depending on the used inspection technique (Field App), field data review and classification (Review RA App), conducting radionuclide measurements of environmental samples (LabApp), and receiving or handing over samples (Lab Field App). Observe that the site is network-isolated – there is no uplink, and furthermore sub-systems are not connected to each other; for instance, there is a strict air gap between the Working Area (WA) and Receiving Area (RA) applications, indicated with the red dotted line in Fig. 3. The software setup is comprised of particular versions as shown in Fig. 4. However, they may need to be updated; this may be a case where the Inspected State Party (ISP – the nation or state in which inspection is performed) provides a piece of equipment to conduct e.g., radionuclide measurements in the mobile laboratory, requiring new software to be deployed. As such, the Laboratory Application must be updated to a newer version. In such a case, an officer assesses the change requested and plans the update procedure. Fig. 4 highlights in red the application which uses the equipment directly, and it highlights in orange four other components which must now incorporate changes as well. The diagram also shows how communication between components is implemented, and indicates the air-gap with dotted lines between applications.

Observe that in order to update the Laboratory Application to version 3.9.1, new versions of others are required: Field Application (A component), LabField

Application (A component), Planning (WA) App PWA (W component), Lab App (W component). The procedure amounts to the following phases:

- **Bootstrapping**. Versions, dependency constraints and artifacts are obtained. Those are defined beforehand by the respective software teams managing them: for the case considered, those are Lab App W "3.9.*" requires LabField App A "3.1.0", Lab App W "3.9.*" requires Field App A "7.9.*," Lab App W "3.9.*" requires Planning (RA) App W "20.1.*," and Lab App W "3.9.*" requires Planning (WA) App PWA "20.1.*." Recall Sec. 4.1 and observe that the dependee version is restricted to only a fixed one of LabField App A, while in the rest only major and minor versions specified and the patch one is open.
- **Update Plan**. The plan to perform needed updates is calculated, and the relevant service artifacts are obtained – those include appropriate container images, data or other binaries. Specifically, the officer selects Lab App W (current installed version "3.8.2"), which is intended to be updated to version "3.9.1." The framework resolves dependencies and returns a list of mandatory updates for other components: LabField App A "3.1.0," Field App A "7.9.0," Planning (RA) App W "20.1.0," and Planning (WA) App PWA "20.1.0." Subsequently, the appropriate artifacts corresponding to the components are obtained from development repositories.
- **Update Execution**. Updates are performed by physically visiting each air-gapped host – details of this step involve protocols outside the scope of this paper.

Overall, we observe that design requirements DR1-DR3 regarding dependability of the update process performed, interoperability with respect to the software supported and auditability regarding update actions are featured in the case performed, pointing to increased applicability of the architecture and end-framework.

5.2 Dependency Resolution Performance

The strategies employed to resolve dependencies represent the most computationally intense activity of the update process; as such, our quantitative evaluation concerns performance assessment with typical problem sizes as elicited from stakeholders (see previous section). Recall that the dependency resolution strategies employ multiple calls to an SMT solver; moreover, they do so differently with MAX-VER deriving (conservative) maximum version resolution only, while ALL-VER targets resolution of all versions, at higher computational cost.

Experiments Setup. Our experiment setup entails (i) generating a suitable dataset and (ii) deploying the proposed framework on a commodity laptop computer typically used in the setting described in the previous section. To obtain a suitable dataset for our experiments, we automatically generate versions in predefined ranges, while constraints and components are set manually to ensure satisfiable solutions. Thereupon, problem instances are synthesized, varying the number of components and the number of versions per component. Experiments

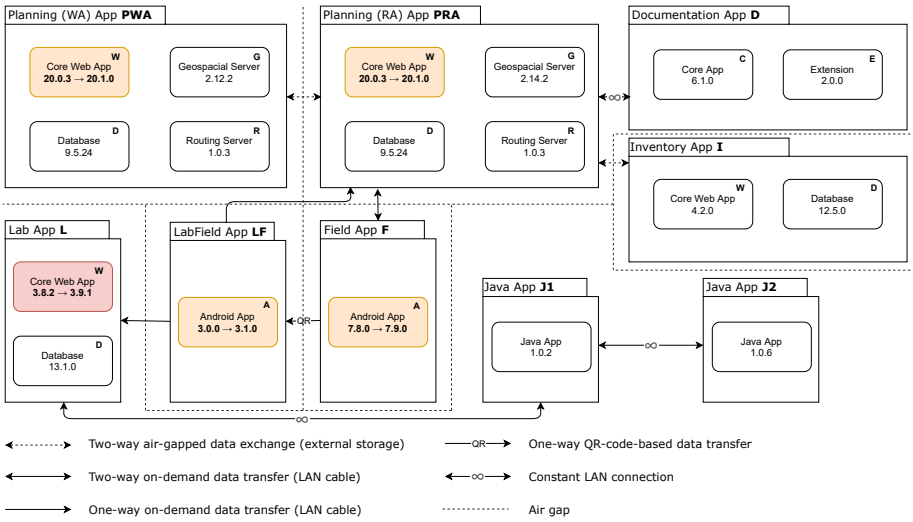


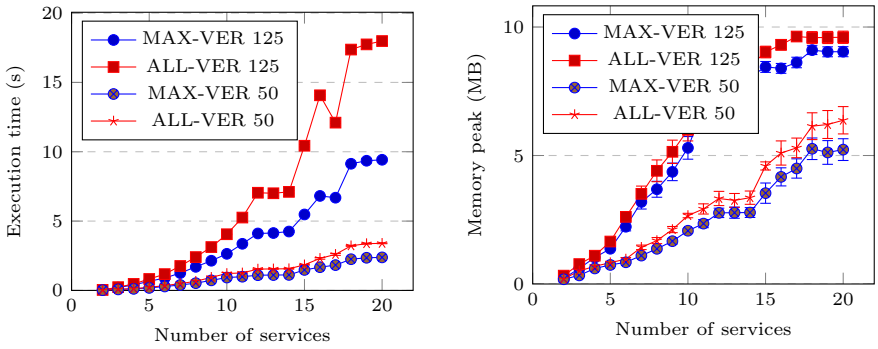
Fig. 4: Service-based software architecture of the OSI case study, illustrating applications comprised of services, with versions (before and after the update) in bold. Applications' air-gapped deployment is denoted with dotted lines.

were performed on a laptop computer featuring an Intel® Core™ i7-6820HQ CPU clocked at 2.70GHz, using PySMT 0.9.0 for the programmatic formula construction and MathSAT 5.6.1 as the underlying solver.

Experiments Results. Quantitative results are illustrated in Fig. 5, for configurations of 50 and 125 versions and increasing the number of services considered. We are interested to investigate i) execution time, and ii) the memory footprint, due to the size of the formulas constructed. Results show that strategies MAX-VER and ALL-VER can handle a realistic number of versions per component within acceptable time budgets. For a system with 15 components and 50 versions for each, the execution time is in the order of seconds, depending on the applied strategy. Although the memory footprint does not grow linearly with the size of the problem, it is arguably insignificant for modern systems.

5.3 Discussion

Based on our evaluation results, we believe to have demonstrated that our framework facilitates the update process for air-gapped systems. A typical scenario was elicited and modelled in Sec. 5.1, demonstrating applicability; we successfully modelled a realistic scenario elicited from stakeholders without running into any conceptual issues with regard to our notions of service versions, dependency management and architecture materializing air-gapped updates. On a functional level, a satisfiable combination of versions is computed and a deployment plan is formed, to be installed via the physical visit where the service artifacts are pushed to the target host. Furthermore, the architecture illustrated in Fig. 2 provides for an end-to-end solution, including configuration management, user



(a) (b)

Fig. 5: Dependency resolution performance (a) and memory footprint (b) of MAX-VER and ALL-VER strategies, over number of services for 50 and 125 versions each.

authentication and container management. Since we followed versioning best practices (tailored for contemporary service-based systems) and employed satisfiability which is widely applied for version management, we believe internal threats to validity of our results to be minimal. However, we note that the case study, although realistic and catering to the needs of an international organization, implied certain type and number of service components, as well as certain design choices in the overall service-based architecture. This is additionally relevant to the quantitative analysis of the dependency resolution; vastly different systems or with different update procedures would imply changes to the workflow and dependency resolution strategies. This would point that results of the case study may not apply to highly diverse cases, which is a threat to external validity. We believe identifying variation points in the architecture presented as a promising avenue of future work.

6 Related Work

The architectural framework proposed is founded on the general area of updating software systems. Accordingly, we classify related work into software evolution management and related approaches dealing with air-gapped environments.

The process of updating service-oriented software systems has been extensively studied, especially concerning so-called dynamic updates [18–20], aiming for reducing downtime while an update is performed. Panzica et al. [19] present a model-driven approach to support software evolution of component-based distributed systems. It requires to build a model, interface automata, to automatically identify the specific class of update. The class is derived based on information locally available in the component and indicates in which state and under which environment condition the system can be correctly updated. In our case, due to the air-gapped nature of the environment, (i) distributed transactions do not take place, and (ii) updates need not happen at runtime. This makes our

problem more akin to an offline one; the runtime context of distributed components is thus not relevant and need not be maintained. Importantly, works on dynamic updates focus on *when* to update, while our focus is on *what* to include in an update.

Software update usually relies on dependency resolution (also known as dependency solving) to identify suitable components and compatible versions. Dependency resolution has been approached by using various types of solvers such as boolean satisfiability (SAT), Mixed Integer Linear Programming (MILP), Answer Set Programming (ASP), or Quantified Boolean Formulae (QBF) [9, 8, 15, 16]. Abate et al. [9] argue that dependency solving should be treated as a separate concern from other component management concerns, proposing a modular software construct to decouple the evolution of dependency solving from that of specific package managers and component models, with a Domain Specific Language (DSL) called CUDF as the interface – the DSL can be used to encode component metadata and user update requests. We identify integrating such advanced features as an interesting avenue of future work.

An alternative approach to updates over a network is to take advantage of mobile agents [21, 22]. Software packages are updated on a central server, then mobile agents installed on a client receive the update. However, this approach does not target air-gapped networks, as it relies on at least occasional network connectivity for transfer of updates. Gravity [23] is a delivery system for provisioning cloud-native applications in regulated, restricted, or remote environments. It allows packaging complex Kubernetes clusters into portable images for later delivery to a cloud-hosted provider-agnostic environment. Among others, deployment in air-gapped environments is additionally targetted allowing to package a whole cluster, including dependencies, to a tarball, eliminating the need for utilizing a network connection during installation. Gravity aims at packaging and transferring of complex Kubernetes clusters, abstaining however from addressing versions – our approach further provides plugin-style support for different target runtime environments, addressing Docker support out-of-the-box while also similarly allowing Kubernetes for orchestration.

Azab et al. [24] target an isolated infrastructure for storing and processing sensitive research data, providing procedures to provision Docker containers in isolated environments. Additionally, security-related disadvantages of Docker containers are identified, and mitigations are proposed. We believe this reinforces the technological choices made in the architecture of Fig. 2 – furthermore [24] showed that Docker can be indeed successfully applied to provision software in network-isolated environments. The study of security limitations of containers and their secure deployment [25, 26] are also important aspects to be taken into account in the overall air-gapped context.

7 Conclusion and Future Work

A central activity within the lifecycle of service-based systems is management of their software updates. Although it is a problem that has been widely tackled

by the community in the past, settings where security constraints impose compulsory network isolation call for specialized treatment. To this end, we adopted an architectural viewpoint and presented a technical framework for updating service-based systems in air-gapped environments. After describing the particularities of the domain, we provided suitable modelling notations for service versions, whereupon satisfiability is used for dependency resolution; an overall architecture was presented in an end-to-end solution. We evaluated the applicability of the framework over a realistic case study of an international organization, and assessed the performance of the dependency resolution procedures for practical problem sizes. As for future work, we identify providing a complete reference architecture that engineers and organizations can use for air-gapped updates of service-based systems as per ISO/IEC/IEEE 42010. Moreover, within the update workflow, dependency constraints defined between components and their versions are left to be set by developers without any verification whether the specified constraint is valid; automatic discovery of dependencies could mitigate this problem, which we identify as future work. Similarly, architecture description languages may be used to ensure that the provided interfaces of services match the required ones of their dependents. Adequately keeping track of the remote state of air-gapped systems is an adjacent problem as well. Finally, explicitly considering the criticality of the domain, security is often a key concern that permeates processes, software architectures and software construction and maintenance and as such warrants further investigation.

References

1. M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.
2. E. Byres, "The air gap: Scada's enduring security myth," *Communications of the ACM*, vol. 56, no. 8, pp. 29–31, 2013.
3. M. Guri, G. Kedma, A. Kachlon, and Y. Elovici, "Airhopper: Bridging the air-gap between isolated networks and mobile phones using radio frequencies," in *2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*, pp. 58–67, IEEE, 2014.
4. M. Guri, B. Zadov, and Y. Elovici, "Odini: Escaping sensitive data from faraday-caged, air-gapped computers via magnetic fields," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1190–1203, 2019.
5. J. A. Morales, H. Yasar, and A. Volkmann, "Implementing devops practices in highly regulated environments," in *Proc. 19th International Conference on Agile Software Development (XP 2018), Companion*, 2018.
6. S. Wong and A. Woepse, "Software development challenges with air-gap isolation," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, (New York, NY, USA), p. 815–820, Association for Computing Machinery, 2018.
7. F. Mancinelli, J. Boender, R. D. Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen, "Managing the complexity of large free and open source package-based software distributions," in *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*, 2006.

8. P. Abate, R. Di Cosmo, J. Boender, and S. Zacchiroli, "Strong dependencies between software components," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pp. 89–99, IEEE, 2009.
9. P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli, "Dependency solving: a separate concern in component evolution management," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2228–2240, 2012.
10. P. Abate, R. D. Cosmo, G. Gousios, and S. Zacchiroli, "Dependency solving is still hard, but we are getting better at it," in *Proc. 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2020)*, 2020.
11. C. Tsigkanos, L. Pasquale, C. Ghezzi, and B. Nuseibeh, "On the interplay between cyber and physical spaces for adaptive security," *IEEE Trans. Dependable Sec. Comput.*, vol. 15, no. 3, pp. 466–480, 2018.
12. C. Russ, "Version sat, research.swtch.com/version-sat, accessed: 22.10.2020," 2016.
13. T. Preston-Werner, "Semantic versioning 2.0.0. 2013," *Online: http://semver.org*, 2019.
14. J. Dietrich, D. Pearce, J. Stringer, A. Tahir, and K. Blincoe, "Dependency versioning in the wild," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 349–359, IEEE, 2019.
15. D. Le Berre and A. Parrain, "On sat technologies for dependency management and beyond," 2008.
16. F. Lonsing and A. Biere, "Depqbf: A dependency-aware qbf solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, no. 2-3, pp. 71–76, 2010.
17. C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*, pp. 305–343, Springer, 2018.
18. L. Baresi, C. Ghezzi, X. Ma, and V. P. La Manna, "Efficient dynamic updates of distributed components through version consistency," *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 340–358, 2016.
19. V. Panzica La Manna, "Local dynamic update for component-based distributed systems," in *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, pp. 167–176, 2012.
20. S. Ajmani, B. Liskov, and L. Shrira, "Modular software upgrades for distributed systems," in *ECOOP 2006 – Object-Oriented Programming* (D. Thomas, ed.), (Berlin, Heidelberg), pp. 452–476, Springer Berlin Heidelberg, 2006.
21. L. Bettini, R. De Nicola, and M. Loreti, "Software update via mobile agent based programming," in *Proceedings of the 2002 ACM symposium on Applied computing*, pp. 32–36, 2002.
22. D. B. Lange, "Mobile objects and mobile agents: The future of distributed computing?," in *European conference on object-oriented programming*, pp. 1–12, Springer, 1998.
23. J. Marin, "Deploying applications into air gapped environments, goteleport.com/blog/airgap-deployment, accessed: 24.03.2021," 2019.
24. A. Azab and D. Domanska, "Software provisioning inside a secure environment as docker containers using stroll file-system," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 674–683, IEEE, 2016.
25. A. Martin, S. Raponi, T. Combe, and R. D. Pietro, "Docker ecosystem - vulnerability analysis," *Comput. Commun.*, vol. 122, pp. 30–43, 2018.
26. Q. Xu, C. Jin, M. F. B. M. Rasid, B. Veeravalli, and K. M. M. Aung, "Blockchain-based decentralized content trust for docker images," *Multimedia Tools and Applications*, vol. 77, no. 14, pp. 18223–18248, 2018.