Automatic Deployment of Distributed Software Systems: Definitions and State of the Art.

Jean-Paul Arcangeli, Raja Boujbel, Sébastien Leriche

Automatic deployment of distributed software systems: Definitions and state of the art

Jean-Paul Arcangeli [a,*], Raja Boujbel [a], Sébastien Leriche [b]

[a] Université de Toulouse, UPS, IRIT, 118 Route de Narbonne, F-31062 Toulouse, France
[b] Université de Toulouse, ENAC, 7 Avenue Édouard Belin, F-31055 Toulouse, France

A B S T R A C T

Deployment of software systems is a complex post-production process that consists in making software available for use and then keeping it operational. It must deal with constraints concerning both the system and the target machine(s), in particular their distribution, heterogeneity and dynamics, and satisfy requirements from different stakeholders. In the context of mobility and openness, deployment must react to the instability of the network of machines (failures, connections, disconnections, variations in the quality of the resources, etc.). Thus, deployment should be an uninterrupted process which also works when software is running and requires adaptiveness in order to continually satisfy the constraints and the requirements. Originally managed "by hand", software deployment demands an increasing level of automation and autonomy.

This article first provides up-to-date terminology and definitions related to software deployment. Then, it proposes an analytical framework and reviews recent research works on automatic deployment with reference to this framework, and synthesizes the results. The review shows that existing solutions are incomplete, and possibly inefficient or unusable, when distribution, heterogeneity, scalability, dynamics and openness are primary concerns. In particular, they barely support dynamic reactions to unforeseeable events. Additionally, abstraction level and expressiveness at design time are rather limited regarding deployment complexity.

1. Introduction

Software deployment is a complex post-production process that consists in making software available for use and then keeping it operational. It concerns a number of inter-related activities such as release, installation, activation, update, etc.

Current software products are no longer monolithic but organized as a set of *components* assembled as a *system* and operating together. The term "component" refers to (weak definition) an element of a system in a context of modularity, or to (strong definition) a unit of composition with contractually defined interfaces that is subject to composition by third parties (Crnkovic et al., 2011; Szyperski, 2002). In the latter case, interfaces specify both the functions or services provided by the component and those required by the component from other ones or its environment. Many models of software components exist such as JavaBeans (Oracle, 2013b), Corba Component Model (Object Management Group, 2006a), or Fractal (Bruneton et al., 2006; OW2 Consortium, 2009). Component-based technologies ease deployment (among other things, components may provide interfaces dedicated to administration and configuration) and components can be deployed independently. Therefore, the use of software components as units of packaging, delivery and deployment, becomes a common practice. Note that component-based technologies are not limited to the application level but may operate at the system level (Coulson et al., 2008). This approach has been proposed in order to build an open configurable and dynamically reconfigurable grid middleware, GridKit (Coulson et al., 2006), where overlay pluggable components adapt the system to the application and/or execution context.

Deployment of distributed component-based software systems must take into account dependencies both between the components themselves (software dependencies) and between the components and their runtime environment (system and hardware dependencies) which provides resources for the execution. Thus, deployment must satisfy a set of constraints (e.g. a logging component needs Linux and 50 Gb of memory to work) and requirements (e.g. a GUI component should be present on every smartphone connected to a given WiFi network) from different stakeholders.

Originally, software deployment was managed "by hand". Nowadays, deployment requires increasing level of automation and autonomy mainly due to distribution, to mobility and pervasiveness, to the increasing number of devices (and users) and their heterogeneity,

* Corresponding author. Tel.: +33 561 556 349.
E-mail addresses: Jean-Paul.Arcangeli@irit.fr (J.-P. Arcangeli), Raja.Boujbel@irit.fr (R. Boujbel), Sebastien.Leriche@enac.fr (S. Leriche).

to the evolutivity of software systems, more generally to hardware and software increasing dynamics. As complexity of software systems grows, complexity of deployment grows too.

1.1. Background

As Weiser imagined more than 20 years ago (Weiser, 1999), the concept of invisible and ubiquitous computers seamlessly integrated into the physical environment has become a reality. Interconnecting spatially distributed devices, equipped with a minimal set of communication and computing capacities, has led lately to the concept of "Internet of Things" (IoT) (Miorandi et al., 2012). The meaning of IoT has evolved since the first use of the term by K. Ashton in 1999 in the context of supply chain management (Ashton, 2009). Here, we mean more or less smart devices (physical objects), possibly mobiles, which compose heterogeneous, unstable and open systems, but not purely Wireless Sensors Networks (WSNs). Nowadays, systems may combine the Internet of Things with mobility of devices and users, and due to hardware and software limitations, these systems demand remote computing resources which can be provided by cloud infrastructures (Gubbi et al., 2013). Thus, recent research works have identified the need to make WSNs, ubiquitous, mobile and cloud computing systems collaborate, so as to build "multi-scale" systems (Flinn, 2012; Kessis et al., 2009; van Steen et al., 2012). At the end of these systems, WSNs on one hand and cloud infrastructures on the other hand are connected through gateways machines. Multi-scale systems are heterogeneous, open, highly dynamic, decentralized, and may be distributed over wireless sensor networks, smart objects, mobile devices, gateway machines, fixed servers or clouds.

In such a context, deployment should manage a huge number of heterogeneous devices and network links and a mass of components and software versions as in the case of large-scale systems (Flissi et al., 2008). At the age of "many computers for everyone" (Weiser, 1996), in a context of high heterogeneity and variability, deployment requires the *push mode*. In the push mode, deployment is initiated and remotely managed by administrators (or by software producers), contrary to the traditional *pull mode* (which should remain available) where deployment is left to software users.

Faced with mobility, disconnections, and variations in the availability and the quality of the resources and the services, deployment must also change and adapt dynamically in order to permanently satisfy constraints and requirements (i.e. preserve expected properties): it should deal with the dynamics and openness of both the network of hosts (e.g. appearance and disappearance of devices) and the deployed software system (e.g. new components to integrate, removal or displacement of existing ones).

Therefore, while deploying applications in a satisfactory way commonly requires human intervention, deployment of modern distributed software systems demands automation: appropriate methods and tools are necessary to design the deployment (e.g. to express deployment constraints and requirements), control and automate the deployment process, and automatically resolve the problems related to instability and openness. However, to the best of our knowledge, there is no turnkey solution.

1.2. Aim and scope of the article

Automating deployment is not a new problem. Different technologies provide support for software deployment such as Linux Redhat Package Manager (Bailey, 2000), Microsoft Windows, Microsoft .Net, Enterprise JavaBeans (Oracle, 2013a), Corba Component Model (Object Management Group, 2006a), OSGi (OSGi Alliance, 2009), or for virtualization like VMware products (VMware Inc., 2008). However, these deployment technical solutions are often limited to the pull mode and installation only. Reviewing them is out of the scope of this article but readers can refer to Dearle (2007) or to Heydarnoori (2008) where several technologies are introduced.

In the industry, configuration management tools such as Chef (Chef Software, Inc., 2014), Puppet (Puppet Labs, 2014), Octopus Deploy (Octopus Deploy Pty. Ltd., 2014), and Ansible (Ansible, Inc., 2014) are widely used. They support software installation and update, with dependence solving, in enterprise networks i.e. on server machines, PCs or virtual machines. As they operate basically in a client–server mode, machines which enter the domain can require the (centralized) server for a configuration and then launch locally an installation script. While Chef and Puppet are not specialized for a particular operating system, Octoplus Deploy is dedicated to the .NET environment. With Ansible, unlike others, machines are not required to install and run background daemons which connects to the server; this reduces the network overhead by avoiding the machines to periodically poll the server. Finally, as these industrial solutions do not target deployment of component-based systems and suppose that the deployment domain is quite stable (otherwise, that its variations are controlled), we do not go on with their analysis.

Therefore, in this article, we aim at providing a state of the art of research works on automatic deployment.

Note that in WSN or cloud context, deployment has specific properties and constraints: for example, domain homogeneity and little openness, or resource limitations in the former case, dynamic resource allocation, transfer of computations or accounting requirements in the latter case. Deployment in WSNs must carefully consider resource consumption such as energy (Levis et al., 2004; Marrón et al., 2006). There, several solutions have been proposed but, generally, they are platform-specific and suppose domain homogeneity. In this article, we are interested in more "agnostic" solutions, which are not tied with a particular platform.

1.3. Plan of the article

Before reviewing the state of the art, we set an up-to-date terminology about software deployment and we propose a framework which supports the analysis of the different works. Then, 19 recent research works are presented and analyzed, and a synthesis is provided with reference to this framework. The review shows that, in a context of distribution, heterogeneity, scalability, dynamics and openness, existing solutions are incomplete, and possibly inefficient or unusable. Additionally, abstraction level and expressiveness in design proves to be limited regarding the complexity of the deployment problem.

The article is organized as follows. Next section provides a reference terminology and definitions related to software deployment. In Section 3, the analytical framework is presented. Section 4 reviews research works according to the analytical framework. Finally, Section 5 provides a synthesis of the review and concludes the discussion.

2. Definitions

The reference papers on software deployment are those of Carzaniga et al. in 1998 and Dearle in 2007. Generic concepts have also been specified by the OMG in the D&C specification (Object Management Group, 2006b).

Carzaniga et al. analyze the issues and the challenges, and propose a characterization framework for software deployment technologies (Carzaniga et al., 1998). The main issues identified are the management of changes, dependencies among components, content delivery, coordination between deployment and use of the software, heterogeneity, security, changeability of the deployment process, and integration with the Internet. The characterization framework is based on four criteria: deployment process coverage, deployment process changeability, interprocess coordination, and support for modeling. Given that framework, the authors review a set of technologies:

installers, package managers, application management systems, content delivery technologies, and standards for system description.

Dearle provides an overview of software deployment and identifies some issues such as binding, the use of containers and inversion of control, reflection, the use of metadata (Dearle, 2007). Then, he examines some issues in the field and how they might be addressed: the focus is placed on the granularity of containers, distributed deployment, middleware, adaptation and autonomicity, and architectural specification. Among other things, he underlines the potential complexity of autonomic deployment and of the dynamic adaptation of the process.

Our understanding of software deployment is based on the definitions of Carzaniga et al. and Dearle. In this section, we take these definitions as a basis for constructing up-to-date or new ones.

2.1. Roles

Stakeholders of the deployment process can play different roles. Dearle introduces two such roles: *software producer* and *software deployer*. For both design and management of deployment, many authors consider only the role of *deployment manager* (i.e. software deployer). Conversely, we consider separately the roles of *deployment designer* and *deployment operator*, which have their own requirements, skills and permissions. In the same way, two additional roles should be considered: *system administrator* and *software user*. In our opinion, refining roles helps to master the increasing complexity of deployment and to identify the requirements.

- **Software producer**. The software producer is the creator of the software system; he also provides the software installer. He acts upstream software delivery, but he has to take into account requirements from deployment.
- **Deployment designer**. The deployment designer expresses deployment commands or only a specification of the deployment (i.e. constraints and requirements), and may use a framework or a dedicated language for that purpose.
- **Deployment operator**. The deployment operator is responsible for the supervision and the execution of deployment. If necessary, he may interact with the system in charge of the deployment.
- **System administrator**. The system administrator manages the resources of the targeted device(s).
- **Software user**. The software user is the end-user of the deployed software. He acts downstream, but may add some requirements and preferences on the deployment process. The software user is at the core of ambient systems, and the software system has to, above all, take into account his preferences and requirements.

Obviously, one person may play several roles in the deployment process. For example, the owner of a smartphone may be the system administrator, the deployment designer and operator, and the user of the deployed software. Besides, automatic deployment supposes that the deployment operator role is played (possibly partially) by a non-human entity (a program). Conversely, several persons can play the same role : for instance, there may be as many system administrators as devices in the case of systems with multiple administration.

2.2. Distribution

Basically, deployment of a software product involves a *producer site* and a *consumer site* (Carzaniga et al., 1998): the producer site hosts the software elements including the installation procedures, the consumer site is the target site of the deployment process on which software is to be run later. A *deployer site* may be involved if deployment is managed by a third party.

Additionally, in case of distributed software systems, the deployment process is itself distributed over the network.

- **Deployment domain**. The deployment domain is the set of networked machines or devices (consumer sites) which hosts the components of the deployed software system.
- **Deployment plan**. The deployment plan is a mapping between a software system and the deployment domain, increased by data for configuration (and management of dependencies between components). Additionally, like for any architectural artefact, rationale at the origin of the plan can be included.

2.3. Activities

For Carzaniga et al., software deployment is a crucial step in the software life cycle. It refers to all the activities that make a software system available for use, such as release (at the end of the production process), installation in the execution environment, activation, deactivation, update, and removal of components (Carzaniga et al., 1998).

For Dearle, software deployment is a post-production activity which can be defined as the processes between the acquisition and the execution of software, consisting in a number of inter-related activities (Dearle, 2007).

We define **software deployment** as **a process which organizes and schedules a set of activities in order to make software available for use and to keep it up-to-date and operational**. From an operational point of view, activities range from software release to software retire on the producer site or to software removal from the consumer site. Some of them occur before or after software execution, while others occur when software is running on the consumer site(s).

There is no consensus about the set of activities and their names. Referring to Carzaniga et al. and Dearle, we set up unified definitions and select a representative name for each activity.

- **Release**. Release concerns all the operations needed to prepare the software component(s) for assembly and distribution (assembling into packages containing sufficient metadata to describe the resources on which the software depends).
- **Installation**. Installation is the initial "integration" of the software component(s) in the consumer site(s). It requires the software component(s) to be transferred (delivery) and configured in order to prepare it (them) for activation.
- **Activation**. Activation covers all the operations required to start the software system or to install triggers that will launch the software system at an appropriate time. For complex software systems, it may require other services and processes to be started.
- **Deactivation**. Deactivation is the reverse of activation, i.e. all operations required to stop the software system. It may require the dependencies to be taken into account in order to warn about the deactivation.
- **Deinstallation**. Deinstallation removes the software component(s) from the consumer site(s).
- **Retire**. Retire concerns all the operations done on the producer site, by the software producer in order to mark the software as obsolete. The main consequence of retiring a software is that no new version will be produced (however, current or previous versions may remain usable).

After installation, operations are necessary to overhaul the deployed software and provide evolutions.

- **Update**. Update is triggered by an event on the producer site: a release of a new version of a piece of the software system (a component) by the producer. It consists in replacing the old component by the new one. It is similar to installation, but less complex, because most dependencies have already been resolved.

The next operations are triggered by a modification in the software environment on a consumer site such as a change in the available

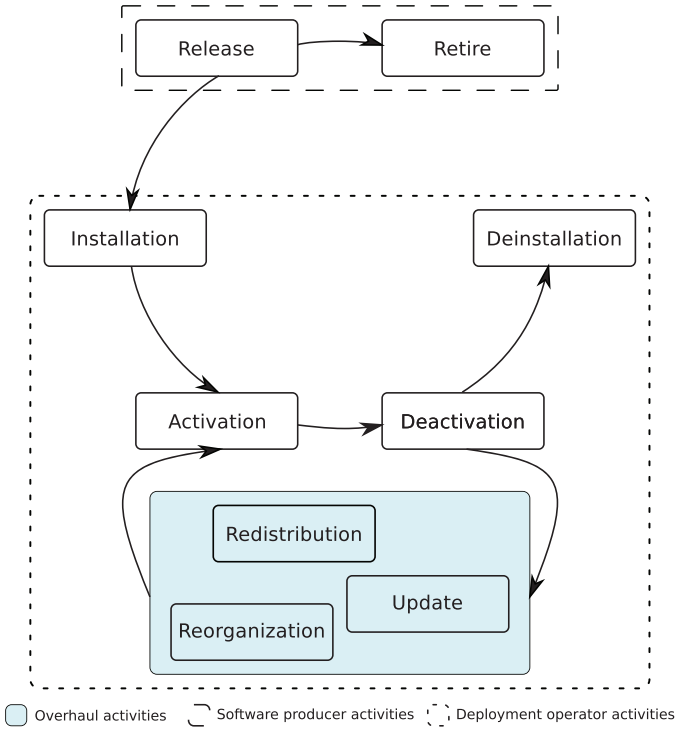

Fig. 1. Scheduling of the deployment activities.

resources in the runtime environment or a request from the operator or the user.

- **Reorganization**. Reorganization modifies the logical structure of the system of components, by replacing a component or modifying configuration parameters and/or links between components.
- **Redistribution**. Redistribution modifies the physical structure of the system of components, that is to say the deployment plan. It may be required when there is a change of the network topology. It possibly demands a new location for the component(s) to be chosen, and consists in moving the component(s) while preserving dependencies, constraints, and requirements. The term "redeployment" is sometimes used as a synonym for redistribution.

2.4. Process

The deployment process organizes the set of activities on the software system, some of them occurring when the system is running. Fig. 1 shows a standard order between the activities. It is worth pointing out that retire is not preceded by deinstallation: indeed, a software can be obsolete (retired on the producer site) without having been deinstalled (on the consumer site).

In a software system, any component has three possible states (after release): *deployable*, *inactive*, *active*. The component is *deployable* when the software producer has accomplished all the necessary operations in order to make it ready for deployment. The component is *inactive* when it is available for use but not yet running. Then, it is *active* when it is in use. The state machine illustrated in Fig. 2 shows how the activities impact the state of one component and when they can occur (depending on its current state). Update supposes previous deactivation of the component – here, we disregard the possibility of running the new version in parallel with the old one, for a more secure and dependable multi-version application, as proposed in Hosek and Cadar (2013) – as well as redistribution. Reorganization may occur in the active state, but if so, it can be limited to (individual) setting of new configuration parameters. In order to simplify the process, the retire operation is not considered: it does not impact the state of the currently deployed component except that it can no longer be updated.

Reorganization of a software system may demand reconfiguration of existing components, but also component addition or removal: adding a component to a system, or removing a component, consists in changing the bindings between components, possibly after installation and/or activation of the new component in case of addition, or deactivation and/or deinstallation in case of removal. Redistribution consists in moving one or several components from one site to another; deactivation (and possibly deinstallation) on the origin site should arrive before moving, and then (installation and) activation should be done on the destination site.

2.5. Design

The activities introduced above are related to an operational point of view on deployment, that is to say to the realization of the deployment plan. However, while Carzaniga et al. and Dearle place the focus on them, we consider that in a situation of growing complexity an essential activity concerning deployment is design.

- **Design.** Deployment design aims at building a deployment plan.

Design should consider different but inter-related activities and deal with both constraints concerning the deployment domain, the application and its components, and requirements which may come from different stakeholders.

The deployment plan may be constructed "by hand" or computed more or less automatically from a specification, namely the expression of a set of expected properties (constraints and requirements) and goals. Note that the deployment plan may be combined with a schedule for its operation, which defines when the deployment should be realized.

Planning is occasionally used as a synonym for design (Heydarnoori, 2008).

2.6. Timeline

Fig. 3 shows the timeline of deployment in relation to the deployed application. Basically, deployment activities also occur when the application is running. Before running, the application is installed and activated: this sequence of activities is commonly named **initial deployment**. After running, the application is deactivated and maybe deinstalled. Deployment runtime is the period when the deployment plan is run (and possibly modified), covering application runtime.

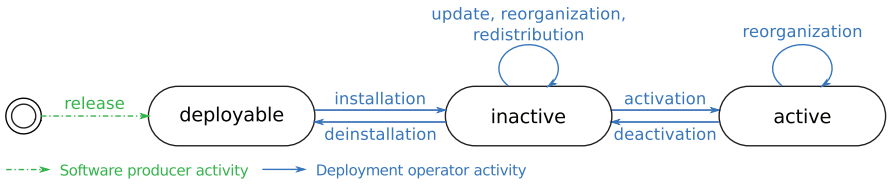

Fig. 2. Impact of the activities on the state of the software.

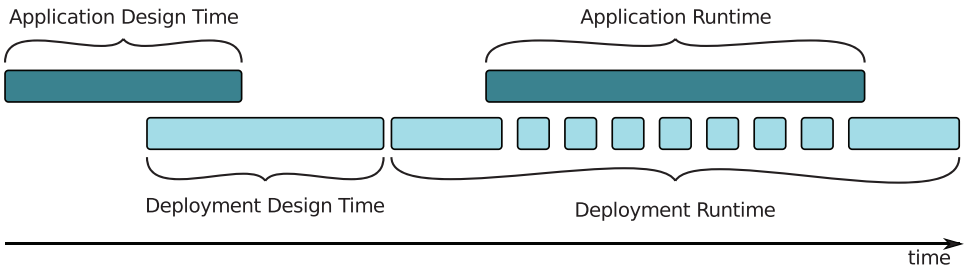

Fig. 3. Deployment timeline.

Especially in the context of open and unstable systems, deployment is not a time-bounded operation because of the dynamic evolution of both the application and the deployment domain (appearing or disappearing devices, loss of domain connexity, etc.). Thus, we define two special forms of deployment: incremental deployment and continuous deployment.

- **Incremental deployment**. Incremental deployment consists in deploying a new software component within an already deployed software system.
- **Continuous deployment**. Continuous deployment consists in managing the deployment of a software component on a device entering (or leaving) the deployment domain.

Incremental deployment concerns operations related to the dynamics of the deployed software system while continuous deployment concerns those related to the dynamics of the deployment domain. Both impact the deployment plan.

3. Analytical framework

The purpose of this section is to define a framework for the analysis and the characterization of the solutions for software deployment. In order to do that, four basic questions must be considered.

- *"What is deployed?"* The first question concerns the nature of the deployed software and of its components, and its changeability.
- *"Where is the software deployed?"* Symmetrically, this question concerns the nature of the deployment domain, its dynamics, openness, and size.
- *"How is the deployment performed?"* This question relates to the realization of deployment: activities, organization and architecture, constraints and assumptions, limits.
- *"How is the deployment designed?"* Since deployment complexity grows as distribution, heterogeneity, pervasiveness, mobility, dynamics, and openness grow, designing deployment is another challenging issue.

In order to analyze the state of the art, we refine the four questions in eleven main points.

First, for the deployed software, we consider two points:

1. ***Nature of the software***: Does the analyzed solution target a monolithic software or a component-based system? If so, is the number of components considered?
2. ***Software changeability***: Does the solution take into account the software dynamics i.e. component(s) evolution, addition or removal of components at runtime?

Concerning the deployment domain, our analysis focuses on the three following points, the two last ones being particularly important in the context presented in Section 1:

3. ***Nature of the domain***: What kind of deployment domain does the solution target? May it be heterogeneous?
4. ***Number and scalability***: Does the solution take into account the number of devices, and is it scalable?
5. ***Dynamics of the domain***: Is the domain open, does its composition change over time as connections, disconnections or failures occur? Is the variation of the quality and the availability of resources taken into account?

In order to analyze the solutions from a realization perspective, we highlight three points:

6. ***Activities***: What are the handled deployment activities?
7. ***Control***: Is deployment controlled in a centralized or decentralized way? Decentralizing the control is a major requirement in a context of large-scale distribution.
8. ***Bootstrap***: On what kind of bootstrap[1] does the solution rely on. This question relates to the specificity of the solution and its dependency to particular technologies.

There are other points related to the achievement of deployment, as for example reliability or efficiency or, more generally, non-functional properties of the deployment. Automatization is the first answer to the reliability requirement. Efficiency is a basic requirement in the context of resource-limited devices. Then, few specific proposals exist (we mention them in the survey when it is relevant). Another issue is the scheduling of deployment operations, and the impact of deployment on the running application. Since dealing with non-functional properties or scheduling is not central in the state of the art, we do not highlight these points in our framework. But we discuss them in Section 5.2.

The last part of the framework concerns deployment design, which becomes an essential activity and demands abstraction and expressiveness. It is worth to notice that directly expressing the deployment plan is not always desirable or even possible: with mobility and openness, devices may be unknown at design time but nevertheless be discovered at runtime and be part of the deployment domain. Additionally, in large-scale domains, it could be useful to be allowed to designate devices or subdomains by means of an abstract characteristic property.

Our analysis focus on how design is supported i.e. on the level of expressiveness and on the skills expected from the designer.

9. ***Nature of the specification***: What must the designer(s) express? Is he forced to construct the deployment plan "by hand" and give it (to be interpreted as a set of deployment commands)? Or can he only express an abstract specification of the expected properties from which the plan is computed?
10. ***Domain transparency***: Can the designer specify deployment without designating explicitly the devices or with some level of abstraction, so that the binding between components and devices is delayed? This issue is strongly linked to the dynamics of the domain and the ability to discover devices dynamically.

[1] A bootstrap a basic executable program on a device, or a runtime environment, which the system in charge of the deployment relies on.

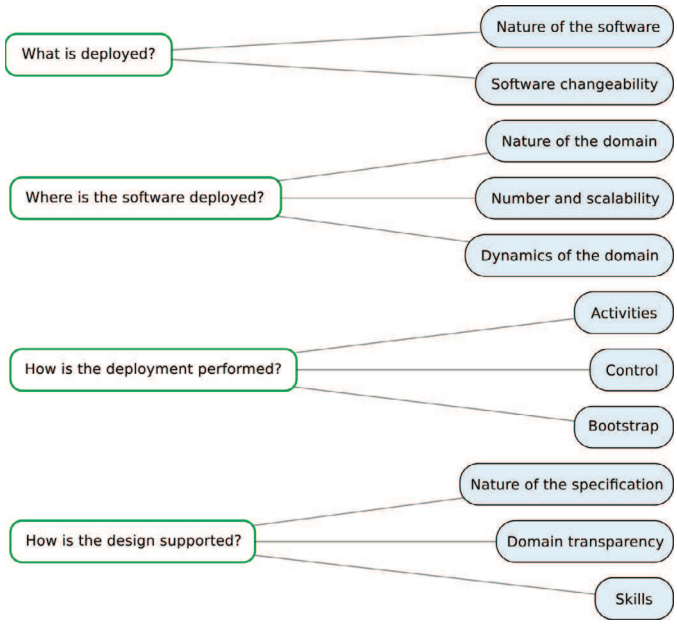

Fig. 4. Analytical framework.

11. ***Skills***: What are the skills required to use the solution and to express the deployment properties? Should stakeholders be experts, and what should be their level of expertise? This last question relates to the usability of the solution, as deployment design may involve different persons playing different roles.

Fig. 4 illustrates the analysis framework and the decomposition of the four questions in the 11 points.

Note that, in Heydarnoori (2008), Heydarnoori compares several deployment techniques proposed in the research community focusing on the deployment activities that are supported, and classifies these techniques in eight categories (QoS-driven, model-driven, agent-based, grid-oriented, etc.) but without relying on a consistent framework as we propose to do.

4. Review

In this section, we review the state of the art of automatic deployment. We consider research works and related projects in which solutions for automatic deployment have been designed. In order to organize the survey only, and independently from the analytical framework, we grouped them in five categories depending on their main goal:

1. extend OSGi or take advantage of OSGi facilities,
2. relieve humans of deployment tasks,
3. allocate resources dynamically to computations,
4. manage resource-limited and mobile devices,
5. provide abstractions in order to facilitate the design.

Every work is reviewed according to the analytical framework introduced in Section 3. For each work, the reader will find a presentation of the *problem*, a description of the *solution*, and a summary in relation to the *main points*. Additionally, a global synthesis of the results is given in Section 5.1.

4.1. OSGi-based automation

The OSGi specification (OSGi Alliance, 2009) defines different functionalities for deployment and remote administration of services. The OSGi framework has moved beyond the original focus of service gateways to provide a full deployment and runtime environment for services implemented through Java-based components called *bundles*. Bundles are the physical units of deployment. A bundle is a Java compressed JAR file that contains a manifest (textual meta-data describing, for instance, dependencies and runtime requirements) and a combination of Java class files, native code and associated resources. Deployment within OSGi allows to install, deinstall, activate ("start" in OSGi), deactivate ("stop" in OSGi), and update components at runtime without restarting the whole system. OSGi implementations (commercial and open source) exist on several heterogeneous devices ranging from smartphone (at least Android 1.5 or Symbian S60), tablet, ultra-mobile PC, car-PC, and laptop to personal computer. OSGi provides a standard way to deploy software while hiding from the users of the framework the heterogeneity of hosts and the low-level technical aspects of deployment, thus allowing them to consider and manage deployment at a high level of abstraction. However, the solution is restricted to a single host.

The OSGi framework has been used as a basis for the deployment of distributed components. For example, the UseNet project (USENET, 2007), which focuses on innovative scenarios and on experimental Machine-to-Machine (M2M) while aiming to enable ubiquitous use of M2M services, uses the OSGi technology for runtime deployment on several types of devices connected to heterogeneous communication networks, but without dealing with dynamic topologies of hosts. Here, we review works which take OSGi as a basis and aim at extending it for specific purposes such as deployment of Fractal components (Desertot et al., 2006) or distributed module management (Rellermeyer et al., 2007).

4.1.1. FROGi

Problem. The general-purpose hierarchical component model Fractal has several limitations regarding deployment: in particular, the concept of the deployment unit is out of the original Fractal specification, and dynamics of deployment is very restricted. Fractal could be enriched with efficient deployment abilities.

Solution. Desertot et al. suggest to combine the Fractal component model with OSGi (Desertot et al., 2006): on one hand, FROGi enhances Fractal by facilitating the deployment of Fractal (primitive or composite) components using the OSGi service platform. On the other hand, it provides the Fractal component-based capabilities to OSGi developers.

In FROGi, a component-based Fractal application is packaged within one or several bundles, and the OSGi platform makes components available as services. Fig. 5 shows a Fractal application packaged and deployed using OSGi. Fig. 5a shows a composite component made of two primitive ones, Client and Server (LC, BC, CC, and AC are names of control interfaces defined in the Fractal specification). Fig. 5b shows the three components (Comp is the composite one), each one packaged within a bundle (B0, B1, B2). Their provided and required interfaces are handled as bundle services. For the packaging of components, FROGi extends the Fractal ADL (Architecture Description Language) in order to allow the specification of bundles, of their version and their properties. Deployment is left to the OSGi platform. During bundle installation, OSGi automatically resolves code dependencies. Then, the bundles are activated, and a bootstrap creates the instances of the components. At runtime, FROGi supports management of the lifecycle of components and their bindings, and dynamic reconfiguration.

Main points. This solution is based on OSGi, Fractal, and the Julia platform, which is a Java-based implementation of the Fractal framework. It supports local deployment of Fractal components but not distribution: a system of components, integrated in one or several bundles, can be deployed but only locally (there is no deployment plan).

The activities covered are installation and deinstallation, activation and deactivation, update, and reorganization (i.e. reconfiguration

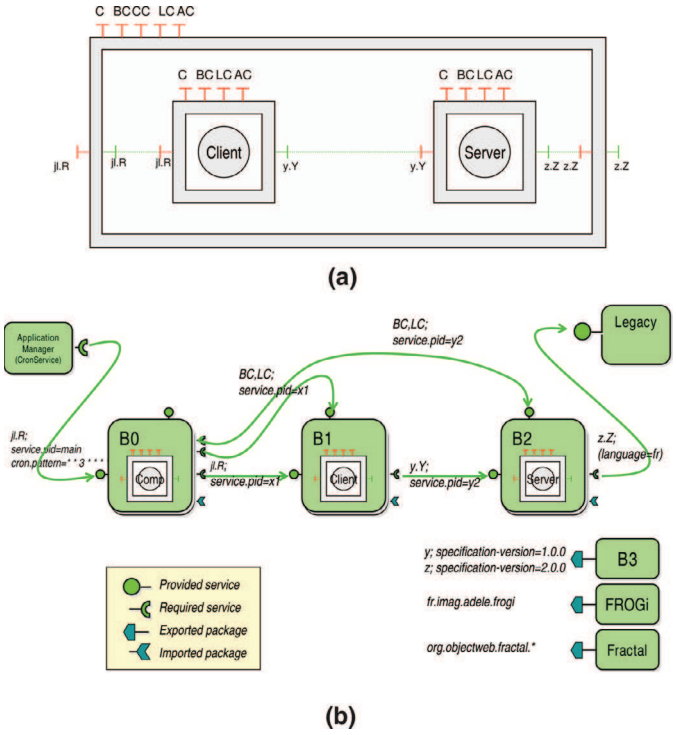

Fig. 5. A Fractal application (a) and its packaging as OSGi bundles (b) Desertot et al. (2006).

of components), while the level of dynamics is the one of OSGi. The targeted device must host the OSGi platform and a FROGi dedicated bootstrap.

The components and the device should be explicitly indicated, and the designer must have skills in Fractal ADL and be able to build OSGi bundles.

4.1.2. R-OSGi

Problem. OSGi is limited to remote but centralized module management. The difficulty is to allow an OSGi-based application to be distributed without loosing the properties of the OSGi model.

Solution. R-OSGi (Remoting-OSGi) is a distributed middleware platform that extends the OSGi specification to support distributed module management in a seamless and efficient way (Rellermeyer et al., 2007). It allows a centralized OSGi application to be automatically and transparently distributed, and run by using proxies.

R-OSGi relies on dynamic proxy generation and type injection to ensure type consistency. Proxies provide OSGi services locally, and hide service invocations across the network; the only difference from standard OSGi services is that they are aware of distribution in order to perform specialized operations, e.g. for system management. Service discovery is reactive and efficient. Services are registered and located by means of a distributed registry implemented with the Service Location Protocol, which complements the centralized OSGi service registry. At runtime, OSGi techniques developed for centralized module management, such as dynamic loading, unloading and bindings of modules are used to handle the dynamics of the domain (e.g. partial failures).

Main points. R-OSGi facilitates the deployment of systems made of OSGi components, in such a way that they can interact remotely at runtime over a network of devices which host the OSGi platform. R-OSGi inherits properties and abilities from OSGi (software changeability, deployment activities).

Here, the designer should make the deployment plan explicit, and must be expert in OSGi technology.

4.2. Relieving humans of deployment tasks

Several works postulate that deployment results in too complex and low-level decisions and actions for humans. Consequently, they target autonomy in deployment, that is precisely deployment without human support. It is the case for Software Dock (Hall et al., 1999) and QUIET (Manzoor and Nefti, 2010) which aim at deploying automatically over networks and whose both architectures are based on mobile agents. Disnix proposes another solution for automatic deployment based on models (van der Burg and Dolstra, 2014). Finally, RAC fully automates the installation and configuration of virtual machines (VM) for cloud computing, and removes the human from the loop (Liu, 2011).

Additionally, the Selfware project (SELFWARE, 2005) aims at limiting human involvement in system administration in order to reduce errors and to enable automatic reaction to special runtime situations. Selfware proposes a software platform which allows distributed systems to be built with autonomous administration (self-repair, self-healing): legacy Java EE applications are encapsulated into Fractal components (OW2 Consortium, 2009) which can be dynamically reconfigured in order to deal with failures or scalability issues. However, the solution is limited to Java EE applications deployed on devices which are known in advance.

4.2.1. Software Dock

Problem. At the end of the 1990s, with the rapid emergence of large networks such as Internet, installing software manually using a CD-ROM was superseded. From that time, network connectivity allowed software producers to offer remote high-level deployment services to consumers. Thus, Hall et al. proposed Software Dock, a distributed agent-based deployment framework which enables cooperation between software producers and software consumers (Hall et al., 1999).

Solution. The Deployable Software Description (DSD) is a main element of the solution. It is a standardized language used for the description of the software system through a collection of semantic properties mainly related to consumer-side features and constraints, and to configuration.

Software Dock architecture is distributed, and based on two sorts of elements: the *release dock* on the producer site which holds a repository of software releases, and the *field dock* on the consumer site which provides local information about resources and configuration, already deployed software systems, etc. Deployment relies on mobile agents which perform deployment activities. Agents move from a release dock to a field dock, while carrying the software release and the DSD description (agent mobility is, however, limited to one jump on a site decided in advance). On the consumer site, they interact with the field dock, and perform customized configuration and deployment by interpreting locally the DSD description (agents may be more or less specialized for one deployment activity). A distributed *event service* supports the connectivity between producers and consumers (see Fig. 6). Relying on the publish-subscribe pattern, release docks may generate events, for example in case of an update. Events are caught by the subscriber agents on consumer sites, which may start updates or reorganizations.

Note that Agilla (Fok et al., 2005) is another deployment solution based on mobile agents that has been developed for WSNs, in which agents support network programming and reprogramming.

Main points. The deployment process itself is distributed over the network but applications are deployed locally as a unit. Many activities are concerned: release and retire on the producer side, installation and deinstallation, update and reorganization on the consumer side.

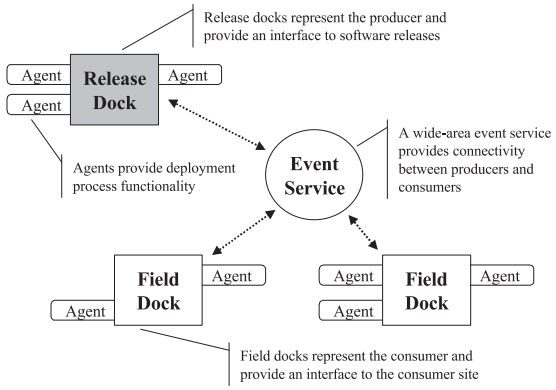

Fig. 6. Software Dock architecture (Hall et al., 1999).

Deployment is decentralized and customized on the consumer site thanks to the local interpretation of descriptors. By means of decentralization and the event-based interaction mode, the global overhead could be limited and the solution could scale and support domain openness, but these issues are not clearly addressed. The implementation relies on the mobile agent platform Voyager which hides the heterogeneity of the domain. In addition, devices must host the field dock (bootstrap).

A specific language (DSD) supports the specification of deployment properties. In this situation, the designer must have a strong expertise in deployment; he has to set up a configuration and to express constraints, dependencies and assertions in the DSD format.

4.2.2. QUIET

Problem. Traditional installation wizards oblige users to participate and express their preferences using a GUI. Manzoor and Nefti aim at automating software installation on networks of PCs and requiring minimal interaction with users.

Solution. QUIET is a framework for automatic installation and deinstallation over a network (Manzoor and Nefti, 2010), which supports "silent and unattended installation" that is to say installation without user interaction. Silent installation relies on the Silent Unattended Installation Package Manager (SUIPM) (Manzoor and Nefti, 2008). At the network level, a multi-agent system monitors the resources (bandwidth, memory, disc, etc.) of the domain and deploys smartly and efficiently the SUIPM on the consumer sites depending on network conditions. Agents provide autonomy and decentralization and their mobility helps in network coverage.

The network is divided into sub-networks, each of them corresponding to a set of consumer sites, managed by sub-servers. Fig. 7 describes how the multi-agent system operates and the behaviors of the different types of agents: each one is in charge of a specific operation and may create agents in order to execute underlying tasks. First, a *Master Controller Agent* (MCA) loads the application to be installed and the description of the deployment domain from XML files, and creates *File Transfer Agents* (FTA) to perform the transfer of configuration files on sub-servers. MCA also creates *Controller Agents* (CA) which migrate on sub-servers and are responsible for installation on sub-networks. On a sub-network, CA creates FTAs to perform the transfer of configuration files on the consumer site, and *Installer Agents* (IA) and *Verification Installer Agents* (VIA) which migrate to consumer site. CA are also responsible for sending to MCA observed context data concerning the network and the process, which helps MCA to build helpful knowledge to take smart decisions. Eventually, on the consumer site, IA installs the SUIPM package and VIA checks that the installation is correct according to the configuration file.

Then, SUIPM can install the application locally. Deinstallation relies on the same principles as installation.

A logging system is used for the deployment system recovery. Any agent is responsible for the supervision of the agents it has created: the creator agent assumes the child agent is dead when it does not receive a log message; then, it creates another (same) agent which resumes the deployment process, thanks to logs.

Main points. The QUIET framework supports distributed, decentralized and parallel deployment of a single component (SUIPM and then MS Windows application) over a network of PCs. Its implementation relies on the distributed agent-based platform Jade which serves as a bootstrap on the devices. Agents support the dynamics of the deployment process, but also its scalability as the domain grows, customization and autonomy. QUIET users may launch a release (*via* SUIPM), an installation or a deinstallation (but software changeability is not considered).

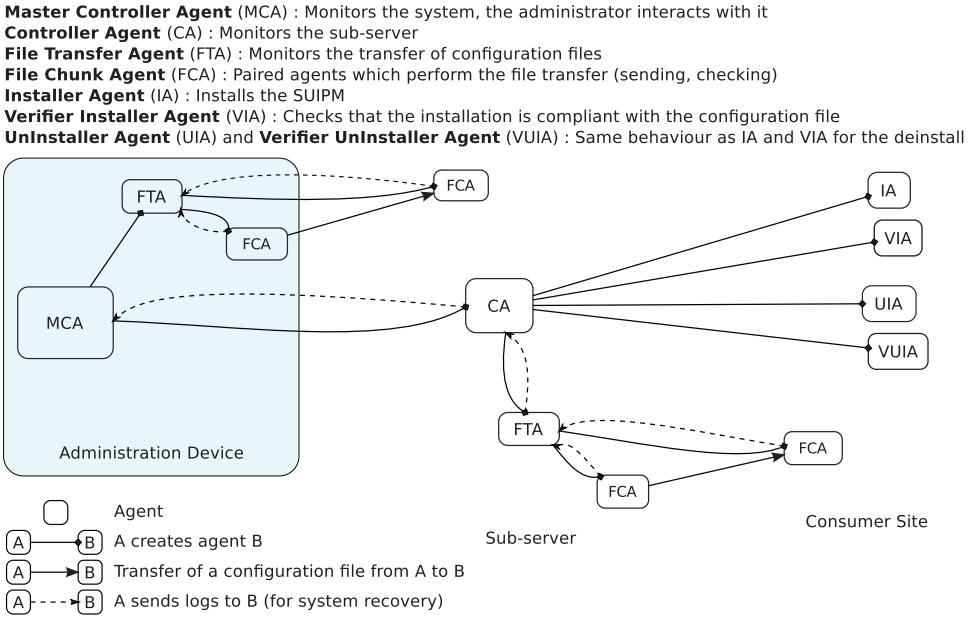

Fig. 7. QUIET multi-agent system.

For this to happen, the deployment manager needs to set up the software and the network configuration (paths, addresses, subservers, etc.) in an XML file, and must therefore have a true expertise in deployment.

4.2.3. Disnix

Problem. On one hand, many deployment tools are specific to a particular type of component (such as Enterprise JavaBeans) or domain. On the other hand, service-oriented systems are composed of inter-dependent, distributed and possibly heterogeneous components which provide the services. Deployment domains may be heterogeneous too, and also change mainly due to failures of devices and communication links. This instability necessitates dynamic redeployment. Consequently, deployment of service-oriented systems is complex and time-consuming, and it is difficult to guarantee non-functional properties.

Solution. van der Burg and Dolstra propose Disnix, a tool for automatic and reliable deployment of service-oriented systems consisting of various types of components in heterogeneous domains (van der Burg and Dolstra, 2010), and an extension called DisnixOS which supports deployment of infrastructure components (van der Burg and Dolstra, 2014). Disnix relies on Nix, a package manager and tool for local deployment.

Automatic distributed deployment is based on declarative models. The *service model* defines the available distributable components and their properties and inter-dependencies. The *infrastructure model* defines the domain. Lastly, the *distribution model* specifies the mapping between the services and the domain, that is to say the deployment plan. Using models and transformation tools hides the complexity of deployment from the deployment managers.

Disnix deployment is launched from a coordinator machine: it consists in building and installing the services, then activating them. In practice, source codes are compiled using the adequate compiler, connectors (such as Java DataBase Connectivity drivers) may be generated, components are installed in such a way that their dependencies (to the host machine or to other components) are satisfied, then services are composed. Update is optimized by limiting it to the deployment of the new components, deactivation of the obsolete services and activation of the new ones.

van der Burg and Dolstra (2011) propose a self-adaptive deployment framework built on top of Disnix. In conjunction with a runtime discovery service (for the machines entering the domain) and an infrastructure generator, the framework generates a mapping of components to machines using a *quality of service model* which supports the expression of a distribution policy based on quality attributes. Fig. 8 illustrates the new architecture.

Main points. Disnix manages the deployment of heterogeneous component-based systems on networks of heterogeneous devices. The dynamics of the application are limited to update, while the dynamics of the domain (appearing devices and failures of devices or network links) are taken into account in the extended version of Disnix which supports automatic redeployment.

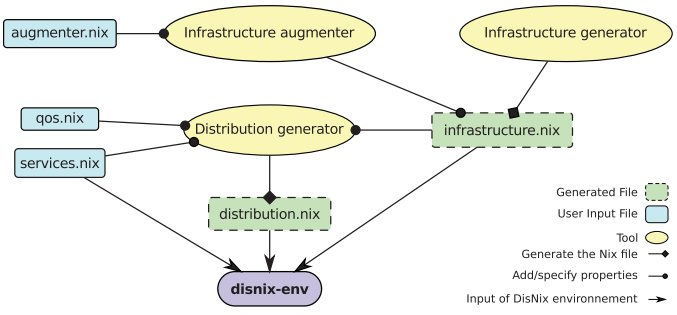

Fig. 8. Extended Disnix deployment architecture (van der Burg and Dolstra, 2011).

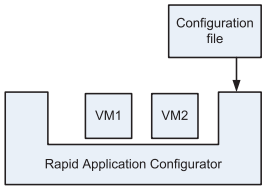

Fig. 9. The RAC container (Liu, 2011).

The deployment activities handled are installation and deinstallation, activation and deactivation, update, reorganization and redistribution in a self-adaptive manner. Each target machine must run the DisnixService (the bootstrap) which enables the coordinator machine to control remote deployment operations.

Disnix users can be administrators of service-oriented distributed systems, but more broadly, various stakeholders may be involved in the expression of deployment (for example, software developers may set the *service model*). The different concerns are clearly separated in different models. In return, any Disnix user must be an expert in its domain. Note that the nature of the specification and domain transparency have evolved with the versions of Disnix: in the initial version, the composition of the domain and the deployment plan had to be explicited, while the self-adaptive deployment framework supports dynamic discovery of devices and plan generation. In that case, as generation of models is continuous, the applicability of the solution appears to be restricted to domains which instability is quite limited.

4.2.4. RAC

Problem. In the context of cloud computing, the use of Virtual Appliances (VM images with pre-packaged and pre-installed software components, written VAs) alleviates software installation and configuration. However, several problems remain which result from the embedding of configurations in VAs: many VM images should be generated in order to cover the multiple combinations of software components corresponding to the multiple deployment scenarios. Besides, the interdependencies among VAs, in multi-tier applications for example, increase the complexity of system configuration and limit customization possibilities.

Solution. Liu proposes Rapid Application Configurator (RAC), an approach for software installation and configuration based on separation of concerns and inversion of control[2] (Liu, 2011). The idea is to separate the configuration data from the application logic, and define configurable properties as variables in the VA header file. Variables are to be set (and the VM configured) at deployment time using configuration metadata. In practice, when deploying a configurable VA, the RAC container (Fig. 9) parses the configuration metadata, performs initial validation (checks basic errors), then instantiates, configures and launches the VM. The whole process is illustrated in Fig. 10 (AMI abbreviates "Amazon Machine Image", where the configurable VA is stored).

In order to be able to exchange values, the RAC container and the VMs expose Web interfaces. The RAC container is implemented as a Web service running on a dedicated server and can read values from

[2] Haller and Odersky define inversion of control" (IoC) as follows (Haller and Odersky, 2006): *"Instead of calling blocking operations (e.g. for obtaining user input), a program merely registers its interest to be resumed on certain events (e.g. an event signaling a pressed button, or changed contents of a text field). In the process, event handlers are installed in the execution environment which are called when certain events occur. The program never calls these event handlers itself. Instead, the execution environment dispatches events to the installed handlers. Thus, control over the execution of program logic is 'inverted'."*

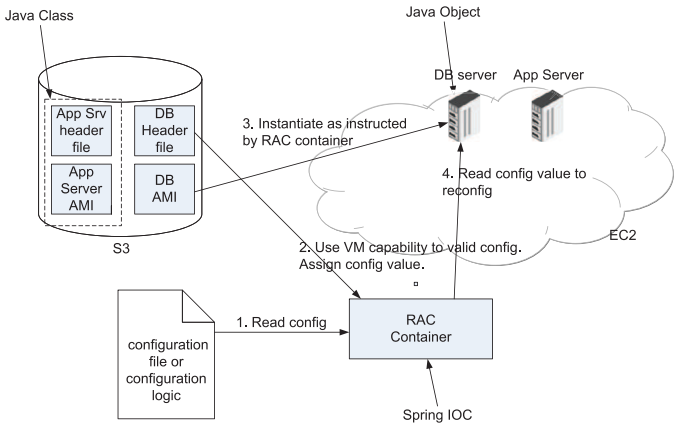

Fig. 10. RAC-based deployment process in Amazon EC2 cloud (Liu, 2011).

the VM. In each VM, a residing agent carries out configuration and reconfiguration: it queries the configuration values once at startup, then periodically polls to see if any value has changed.

Main points. RAC proposes a cloud-specific solution for the full automation of configuration and reconfiguration (installation, activation and reorganization activities), which removes humans from the loop.

The configuration task is shared between the VA producer, an expert in configuration and the end-user, each party having the required knowledge and skills. In order to relieve humans of configuration tasks, metadata replace installation manuals and the RAC container replaces end-users and acts as a centralized service that provides configuration data on demand. This proposal results in reduced cost and time for configuration handling.

Additionally, component distribution (the construction and the realization of the deployment plan) is supported by the cloud infrastructure and so the domain transparency. Here, the only necessary bootstrap is the virtualization platform.

4.3. Dynamic resource allocation

Grids and clouds provide runtime environments for high performance computing over heterogeneous resources and multiple administrative domains. Associated management tools provide services to deal with resources such as provisioning and scheduling. Nevertheless, as the availability and the quality of resources are not permanently predictable or can vary at runtime, dynamic adaptation of the deployment plan is required in order to supply components with the needed resources. Automatic and dynamic deployment is thus desirable. However, finding a location for programs is a complex task. CoRDAGe addresses autonomic deployment of long-span applications on large-scale grids (Cudennec et al., 2008), and Wrangler targets automatic deployment of distributed applications on clouds (Juve and Deelman, 2011).

4.3.1. CoRDAGe

Problem. Nowadays large-scale grid applications may run for days or even weeks over hundreds or thousands of nodes. The exact need for physical resources is difficult to predict before or when initiating deployment. Thus, the deployment manager (operator) has to monitor the applications and permanently satisfy their resource requirements: in practice, elasticity of the deployment domain is required in order to expand or retract the application. Unfortunately, existing deployment tools are "one-shot": they do not provide support for continuous (re)deployment. Another problem concerns the management of "co-deployment", that is to say deployment of several coupled applications.

Solution. Cudennec et al. propose CoRDAGe, a co-deployment and redeployment tool (Cudennec et al., 2008). It is based on ADAGE (ADAGE, 2007), a tool for centralized and automatic deployment on computational grids. ADAGE operates as follows. The input information is: a description of the application, a description of the expected quality of service at runtime, and a description of the resources or of their location. In a scheduling step, a deployment plan is built from these descriptions: the components of the application are mapped onto a subset of selected resources, which satisfy constraints concerning the operating system, processors, memory, etc. Then, the deployment plan is performed: files (executables and data files) are transferred according to the plan, processes are created, configured and launched, and finally a deployment report is generated.

CoRDAGe supports the deployment of applications before, during and after their execution. Fig. 11b illustrates the advantages of CoRDAGe: deployment is transparent, i.e. the user is not in charge of requesting resources nor of deploying applications (as illustrated in Fig. 11a); the only information needed from the user is the initial application description. In Fig. 11, numbers refer to the order of the operations.

High-level generic models are used to represent both the application and the physical resources. An application is described as a set of *types of entities*, each of them being a program to be executed on a single physical resource (a computing node). Configuration consists in defining entities by instantiating types; entities are the deployment units managed by CoRDAGe. CoRDAGe generates tree-based representations of both the application from the application description

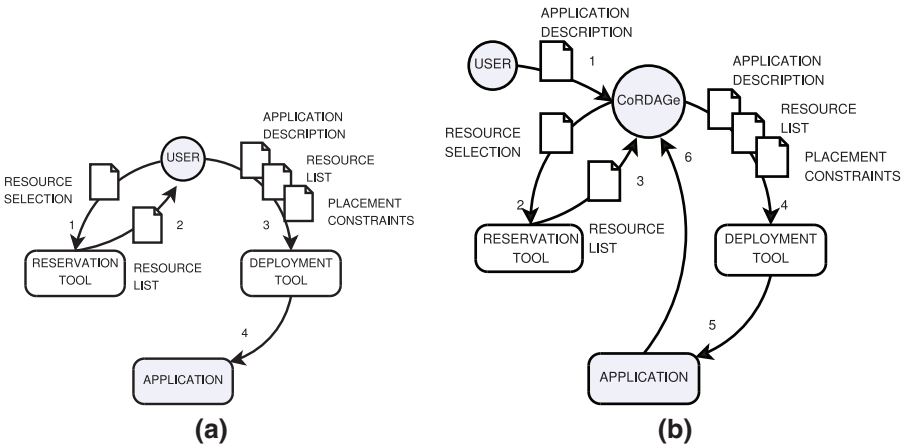

Fig. 11. Deploying an application by hand (a), and using the CoRDAGe tool (b) (Cudennec et al., 2008).

and the physical resources (the deployment domain) from a selection of the bookable ones. Then, a mapping of the application tree onto the physical tree decides on the placement of the entities; this placement (distribution of the system) is performed by the deployment tool provided by the grid considering a set of constraints to be satisfied.

At runtime, relying on CoRDAGe services, applications can manage deployment operations autonomously (that is to say, without user interaction) by requesting expansion and retraction. Dynamic expansion and retraction are also supported by tree-based operations.

Additionally, CoRDAGe can manage several applications together, considering cross-application spatial and temporal constraints.

Main points. On top of ADAGE and various reservation and deployment middleware tools, CoRDAGe manages deployment of long-span distributed applications whose structure and resource requirements may change at runtime. It focuses on the initial placement of programs, their reorganization and their redistribution, on large-scale grids (thousands of nodes) where resource availability and quality vary dynamically.

The bootstrap on devices is the ADAGE platform (and the grid middleware) which performs deployment locally. However, deployment is controlled in a centralized manner.

At design time, the deployment manager must specify the initial application and constraints using high-level models, but not the domain. Thus, little expertise in deployment is demanded.

4.3.2. Wrangler

Problem. Compared to clusters and grids, clouds are highly dynamic especially as several providers are involved. Given these dynamics, deploying services in the "Infrastructure as a Service" (IaaS) cloud is a challenge. Even if cloud infrastructures allow resource provisioning, they lack services for deployment, configuration and application-specific customization of runtime environments (i.e. for building virtual clusters which host applications). These tasks can be handled manually, but they are complex, time-consuming and error-prone.

As for traditional high-performance computing, tools are needed to automatically install, configure, and run distributed services. In Juve and Deelman (2011), the authors list a set of requirements: deploy automatically distributed applications, support complex dependencies, allow dynamic provisioning in order to adapt deployment to the application runtime requirements, support multiple cloud providers, and continuously monitor the state of the deployment.

Solution. Juve and Deelman propose a system called Wrangler which allows users to specify their application declaratively (*via* an XML description), and automatically provision, configure, and monitor its deployment on IaaS clouds (Juve and Deelman, 2011). Wrangler interfaces with several different cloud providers in order to provision virtual machines, coordinates the configuration and initiation of services to support distributed applications, and monitors applications over time. It handles possible failures and dynamic addition and removal of nodes.

Wrangler distributed architecture is illustrated in Fig. 12. It relies on four kinds of components: *client*, *coordinator*, *agent*, *plugin*. Clients run on user's machines and send requests to the (unique) coordinator for new deployment, incremental deployment, or termination; requests for deployment include XML descriptions of the nodes to be launched and of the VM images and plugins to use. Plugins are user scripts that define in a modular way different aspects of the application-specific behavior for a node (the configuration). The coordinator is a Web service: it serves as an intermediary between the clients and the agents. It also interacts with the resource providers in order to provision the adequate virtual machines. Any node of the deployment domain hosts an agent: the code of the agent is pre-installed

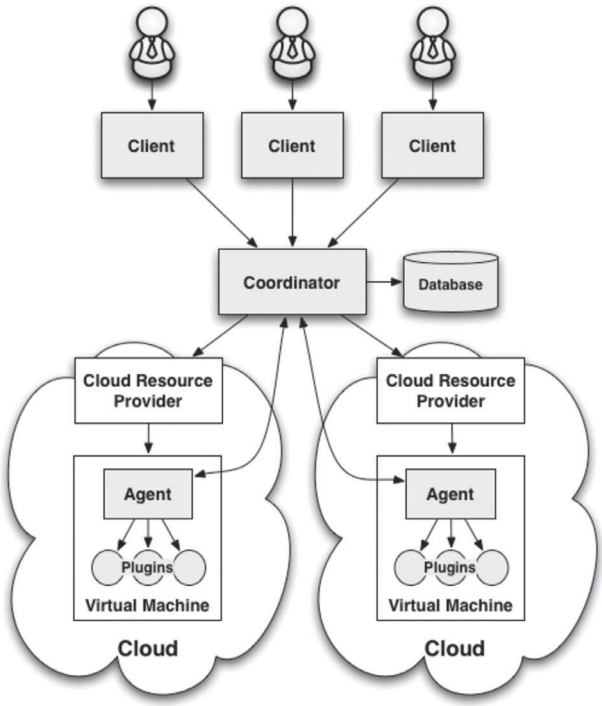

Fig. 12. Wrangler system architecture (Juve and Deelman, 2011).

in the VM image, and when the VM boots up, it launches the agent which registers with the coordinator. Agents receive commands for configuration: in response, they retrieve the list of plugins of the node from the coordinator and download and start them. At the same time, agents monitor their hosting node by invoking the "status" method of the plugins, and send the collected data to the coordinator. They also terminate plugins in response to a termination command. Therefore, agents are in charge of the distributed and decentralized deployment tasks.

Main points. Wrangler is a partially decentralized solution for deployment on multi-provider dynamic clouds of distributed applications whose resource requirements may vary at runtime. Application components are VM images.

Relying on the cloud infrastructure (which is the only bootstrap), Wrangler supports distribution of virtual machines, redistribution, configuration and reconfiguration of nodes (reorganization), and termination of applications. However, the centralized coordinator limits the scalability (related to the domain).

In practice, the deployment designer has to specify the components with their parameters and dependencies, the cloud providers, and the plugins. He should have a certain level of expertise, particularly as he may be involved in the building of VM images or correction of problems at runtime.

4.4. Management of resource-limited and mobile devices

Mobility of devices with limited capacity poses problems related to resource consumption, disconnections and quality of service, and as a consequence demands just-in-time deployment. DVM (Balani et al., 2006) and QARI (Horré et al., 2011) targets WSNs, where resource management (especially energy) is critical. Kalimucho addresses the adaptation of the deployment plan to the quality of service (Louberry et al., 2011). StarCCM supports context-aware deployment (Zheng et al., 2006) and Codewan supports opportunistic deployment of component-based software on disconnected Mobile Ad hoc NETworks (Guidec et al., 2010). Cloudlet is a VM-based solution which

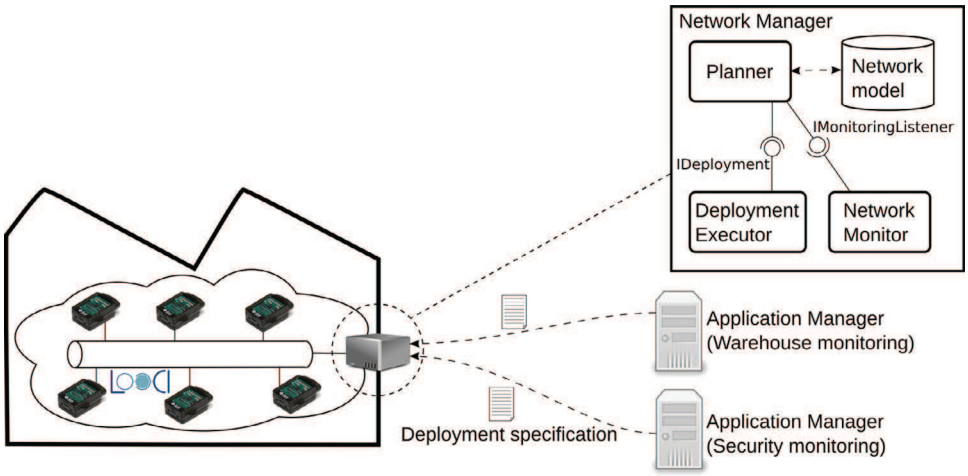

Fig. 13. Overview of QARI (Horré et al., 2011).

allows the workload to be moved from mobile devices to a proximity cloud (Satyanarayanan et al., 2009).

4.4.1. DVM

Problem. In WSNs, usage scenarios are frequently known without enough precision before or at initial deployment time. This point motivates dynamic adaptation of the network by reconfiguration and reprogramming of initially deployed software. However, realizing adaptation is strongly constrained by the limited capacities of sensor nodes (such as energy or memory).

Solution. Balani et al. state that upgrade costs and resource consumption in WSNs increase with flexibility. For a better trade-off between them, they propose DVM (Balani et al., 2006), a dynamically extensible virtual machine in charge of local deployment in an homogeneous network of sensors. DVM runs on top of an operating system consisting of an adaptable tiny kernel and binary modules that can be added to the kernel, updated, reconfigured or removed dynamically. It executes the deployment operations within the operating system in reaction to specified events. DVM basic library may itself be extended dynamically by high-level scripts. However, flexibility of the deployment (what is possible) and efficiency (how much does it cost) directly depend on the defined set of operations in the basic library: dynamic loading increases deployment costs, and the price to pay for optimization is a loss of abstraction in programming.

Main points. DVM focuses on the realization of deployment operations in WSNs. More precisely, it deals with dynamic upgrade of the operating system *via* installation, update, and reorganization of binary modules. It acts locally on sensor nodes in reaction to the events defined by the designer. Thus, deployment is fully decentralized but its scalability (in number of nodes) may be limited by the network traffic. On the other hand, DVM (the system in charge of deployment) is dynamically extensible. In a general way, limited capacities of sensors and efficiency requirements constrain the extensibility and upgrading abilities.

Here, designers write reconfiguration scripts in a concise and flexible way, and can customize the event-based system. However, low-level platform-specific programming is necessary to meet efficiency requirements. Therefore, designers should be experts both in deployment on WSNs and in the device operating system.

4.4.2. QARI

Problem. When deploying in dynamic and unreliable networked systems, the single failure of one host can compromise the whole deployment. Nevertheless, from the developer perspective, the deployment can be considered as valid as long as some requirements are fulfilled. The problem is to give to the developer tools to express high-level quality goals, then to meet the quality requirements in the different deployment phases.

Solution. QARI (Quality Aware Reconfiguration Infrastructure) (Horré et al., 2011) addresses the challenge of software management in WSNs by offering to application developers a way to specify quality goals in XML files, and then enabling quality-aware software deployment. An example is to set a minimal sampling rate for temperature sensing, and then deploy temperature sensing components accordingly. The realization of such a specification is delegated to an Application Manager and a Network Manager, both located on the WSN gateway, as illustrated in Fig. 13. A Network Monitor, a Deployment Planner, and a Deployment executor are the buildings blocks of the Network Manager. These entities use contextual data about the sensors, that have been collected nearby (and so valuable), to achieve the deployment with the required level of quality and to maintain it throughout the lifetime of the application by redistributing components. The Planner uses a heuristic algorithm to calculate the initial assignment of the components to the nodes.

QARI is built on top of LooCI (Hughes et al., 2012), a component model for networked embedded systems such as WSNs, which supports runtime introspection and reconfiguration with a low overhead. The LooCI middleware is designed for Java devices such as Sun SPOT and Sentilla Perk (supporting Java ME) but can also run on more powerful Java devices.

Main points. Relying on quality goals, QARI deploys LooCI component-based applications on dynamic WSNs (with failures or mobility).

QARI mainly targets installation and redistribution (precisely, component placement). It relies on the QARI/LooCI platform that serves as a bootstrap. The scalability of the solution may be limited to a certain extent by the centralized processing of data by the managers located on the WSN gateway.

With QARI, application developers specify quality goals in a simple way (XML rules), and neither the domain nor the deployment plan, which is computed and updated by the planner. Apart from the quality features, the designer may have skills in QARI and, to a lesser extent, in LooCI.

4.4.3. Kalimucho

Problem. Resource-limited mobile devices bring new challenges that deployment platforms must cope with. Actually, the permanent need to manage resources demands specific ways to deploy applications

on the devices: in order to guarantee an adequate quality of service to end-users, distribution (the deployment plan) has to be adapted reactively and dynamically.

Solution. Kalimucho is a distributed platform for dynamic deployment on heterogeneous devices such as desktops, laptops and mobile devices (Cassagnes et al., 2009). Applications are made of business components linked by connectors, in accordance with the Osagaia component model and the Korrontea connector model. According to the principle of separation of concerns, Osagaia business components run inside configurable containers which support connection to other containers using Korrontea connectors. One of the Kalimucho services is devoted to supervision and is distributed over the domain. It has a global vision of the network and the devices. Other basic services allow creation, stopping or removal of components or connectors, migration of components, their connection and disconnection, etc.

Louberry et al. introduce a contextual deployment heuristic, in order to select a configuration which satisfies the quality of service requirements, then to place the components on the devices (Louberry et al., 2011). Parameters of the algorithm are the kind of devices, and their amount of energy, CPU workload and available memory. Therefore, when there is a new functional requirement (for example, resulting from an action of a user), if a device appears or disappears from the network or becomes low in resources, or if the priority of the requirements changes, the Kalimucho platform revises the deployment plan, then performs it. Reorganization and redistribution are done while the application is running without having to stop it, thereby ensuring continuity of service and durability of applications.

Main points. Deployed applications are systems of Java components, which comply with the Osagaia and Korrontea component models, and may dynamically evolve. The domain is a set of networked devices that must host the Kalimucho platform (possibly, a limited version depending on device capacity) as a bootstrap. Devices may dynamically enter or leave the domain.

Kalimucho supports consumer-side deployment activities: installation and deinstallation, update, reorganization and redistribution. It deals with decentralized and context-aware adaptation of the deployment plan.

Users can interact with the Kalimucho platform by issuing a deployment plan or explicitly activating or moving a component. They may also express expected properties of quality of service, and parameter the deployment heuristic by describing the components (constraints and requirements), configurations, devices, etc. As a result, they must be experts in the Kalimucho technology.

4.4.4. StarCCM-based deployment

Problem. On one hand, ubiquitous applications should be aware of the available resources and the constantly changing runtime context in order to be able to adapt, and automation of their deployment is demanded. On the other hand, component-based middleware such as Corba Component Model (CCM) provides deployment facilities but not context-sensitivity. Here, the problem concerns just-in-time adaptation of the deployment plan to the runtime context.

Solution. Zheng et al. propose a middleware-based approach for the deployment of context-aware component-based distributed applications (Zheng et al., 2007). Applications and middleware implementations are based on CCM.

The overall middleware architecture (see Fig. 14) is based on three core services: context management which provides information about context, adaptation management which decides on a deployment plan, and configuration management which realizes the plan, that is to say the redistribution of the components.

Fig. 15 explains how information about context is collected, filtered and processed, and then presented to the *component deployer* (which is an abstraction for both adaptor and configurator components). The data processing chain is based on the publish-subscribe pattern, and on different components: sensor agents, collectors, interpreters and analyzers.

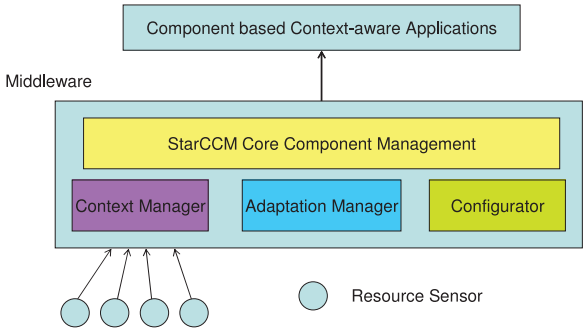

Fig. 14. Architecture of the context-aware middleware (Zheng et al., 2007).

The core of the solution is the *Adaptation Manager*. It computes a new deployment plan at runtime, relying on a set of rules about the component units (selection of version, individual adaptation by configuration) or the application (assembly-level adaptation, optional components, placement): an A* algorithm selects the best version and hosting device for every instance of component.

Main points. Zheng et al. focus on context management as a key for adaptive deployment. StarCCM supports context-aware just-in-time (re)deployment of distributed component-based applications over networks of devices whose resources are limited and changing. The solution targets dynamic adaptation of the deployment plan, and thus the deployment middleware supports the redistribution activity. It relies on a centralized entity: the Adaptation Manager. The architecture is reasonably abstract, but its implementation is tied to CCM.

In this context, the deployment designer should specify adaptation rules at the component and the system levels. He must be an expert in deployment, and also have some knowledge about the properties of the components to deploy.

4.4.5. Codewan

Problem. In infrastructure networks, deployment of software components (precisely their delivery) is based on server hosts which store components in repositories and deliver them on demand. However, in Mobile Ad hoc NETworks (MANETs) with their volatility, disconnections and fragmentation, components cannot be retrieved from a central repository. Additionally, due to continuous and unpredictable

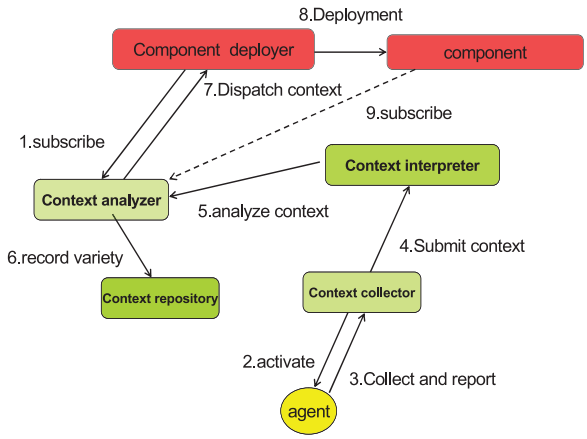

Fig. 15. Dynamics of the context-aware middleware (Zheng et al., 2006).

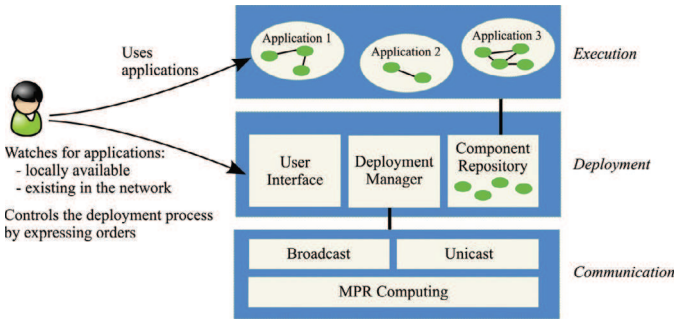

Fig. 16. Architecture of the Codewan platform (Guidec et al., 2010).

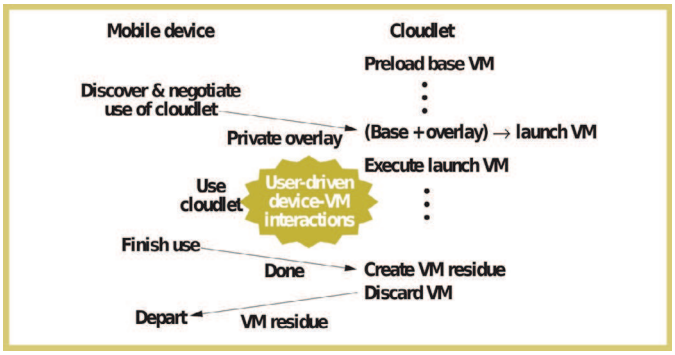

Fig. 17. Dynamic VM synthesis timeline (Satyanarayanan et al., 2009).

changes of the network structure, there is no guarantee that a demand can be satisfied because the component may be (temporarily or not) inaccessible in the neighborhood of the enquirer.

Solution. In the context of the SARAH project (SARAH, 2005), Guidec et al. have proposed a cooperative decentralized model for provision and delivery of software components on disconnected MANETs, and a middleware implementation called Codewan (Guidec et al., 2010). In their neighborhood, devices interact opportunistically in order to discover and exchange software components.

The architecture of the Codewan platform is composed of three layers as illustrated in Fig. 16. The opportunistic communication layer supports the dissemination of components: their announcements are broadcast while unicast transmissions support demands in reply (MPR means multi-point relays). On a device, the deployment layer contains a component local repository, a deployment manager and a GUI. The user interface allows the user both to observe the status of the components and to demand (or cancel) deployment of components or of component-based applications. The deployment manager takes orders from the user. Peer-to-peer cooperation between deployment managers allows devices to obtain copies of the software components required by the user and available in their neighborhood, while the neighbors can benefit from the same service symmetrically. A deployment manager is responsible for updating the local repository. It can fully manage the (local) deployment of a component-based application: it can examine the dependencies between components and run a complete recovery process. In order to do that efficiently, it learns from the interactions with its neighbors.

Main points. Codewan targets deployment of component-based applications (without being tied to a particular component model) on disconnected MANETs. Deployment is local and on demand, and fully supports the dynamics of MANETs.

Codewan focuses on one part of the installation activity, namely delivery. In addition, the solution relies on a model for component packaging, and thus concerns the release activity. Even if scalability is not an objective, the solution could scale as the domain grows due to the full decentralization and the peer-to-peer approach. Codewan platform is implemented in Java and must be pre-installed in all involved devices (bootstrap).

MANET technology supports automatic domain discovery. The deployment is directed by the user of the device using a GUI which requires few advanced skills. However, for the packaging, the software producer must be an expert.

4.4.6. Cloudlet

Problem. Devices taking part in mobile computing are intrinsically resource-limited. This can be a major technical limitation for many advanced applications which require resources like processing power and energy. A solution could be found in clouds, but WAN interactions raise issues of latency, longer delay and jerking. Satyanarayanan et al. introduce the concept of local cloud (Satyanarayanan et al., 2009): instead of delegating computation tasks to a distant cloud, authors propose to delegate them to a nearby (LAN accessible) and resource-rich device called "cloudlet", thus avoiding some of the problems. The challenge is passed on to software management, ideally self-management.

Solution. Satyanarayanan et al. present a solution based on "transient customization" of the cloudlet infrastructure using virtual machine (VM) technology and dynamic VM synthesis (Fig. 17): a customized VM is dynamically synthesized from both a base VM preloaded on the cloudlet infrastructure and a small complementary overlay VM which encapsulates the application, and which is transferred from the mobile device to the cloudlet. The base VM is extended by means of scripts for installation and resumption. After having remotely run the application in the customized VM, the latter is discarded and the cloudlet returns to its initial state. Accordingly, performance depends only on local resources (bandwidth between the cloudlet and the mobile device, and the cloudlet computing power), and WAN failures do not affect synthesis and execution.

An implementation called Kimberley relies on a virtual machine manager for Linux. It lets open several issues such as cloudlet sizing and security.

Remark that the principle of overlay VM is close to the one of RAC (see Section 4.2.4) and to the idea of overlay component used to customize a grid middleware such as it is proposed in GridKit (Coulson et al., 2006).

Main points. The Cloudlet solution deploys (moves and brings back) full applications at runtime. Applications are technology-independent. The deployment target is a remote nearby device (the cloudlet) which appears spontaneously in a context of mobility.

The focus is placed on the transfer of computations. The handled activities are installation and deinstallation (including transfer), activation and deactivation. In order to function, a Kimberley Control Manager (KCM) should run both on the cloudlet and on the mobile device (bootstrap), and the cloudlet should also host the base VM.

The transfers are initiated and explicited by the user whose expertise level may be low.

4.5. High-level formalisms and expressiveness

This section reviews several works which main objective is to provide abstractions and facilitate the expression of the deployment. DeployWare is a complete framework for large-scale deployment based on a modeling language dedicated to deployment (Flissi et al., 2008). ADME is another framework, which targets autonomic deployment and relies on the resolution of constraint satisfaction problems (Dearle et al., 2004). TUNe provides high-level formalisms for autonomous management of decentralized large-scale grids (Broto

et al., 2008; Toure et al., 2010). SmartFrog has been designed with the express purpose of making the design, deployment and management of distributed component-based systems simpler and more robust (Goldsack et al., 2009; Sabharwal, 2006). At last, ORYA focuses purely on deployment strategies and proposes a framework for their expression using properties and rules (Cunin et al., 2005).

As deployment design is a particular operation with specific requirements, some Domain-Specific Languages (DSL) have been proposed to meet the needs for expression of deployment properties. DSLs allow the definition of the deployment properties using well-adapted and optimized idioms and abstractions, so they can be used efficiently by experts of the domain (Strembeck and Zdun, 2009). Among DSLs for deployment, we can cite Deladas (see Section 4.5.2), J-ASD (Matougui and Leriche, 2012), and Pim4Cloud (Brandtzæg et al., 2012). In Sledziewski et al. (2010), authors advocate the use of DSL in order to enhance application development and deployment on clouds. Nevertheless, reviewing DSLs for deployment is outside the scope of this article.

4.5.1. DeployWare

Problem. Flissi et al. analyze the main challenges of large-scale deployment on grids. Deployment frameworks should address the complexity resulting from the huge number of nodes and software dependencies, the heterogeneity related both to the software and the deployment domain, reliability, parallelization, scalability, and monitoring and management issues (Flissi et al., 2008).

Solution. Flissi et al. propose DeployWare, a generic framework for the deployment of distributed and heterogeneous software systems on grids (Flissi et al., 2008). DeployWare provides a Domain-Specific Modeling Language (DSML) based on a metamodel which captures the abstract concepts of deployment, independently of the software paradigm and technology. DeployWare models describe configurations to deploy. They are written using an architecture description language (ADL), and validated before execution to ensure reliability.

The DeployWare runtime is made of Fractal software components, and distributed on selected server nodes. It executes DeployWare descriptions: a virtual machine interprets descriptions and automatically orchestrates complex deployment processes, dealing with software dependencies and hardware heterogeneity (see Fig. 18). Additionally, a graphical console allows administrators to monitor and manage the deployed system at runtime.

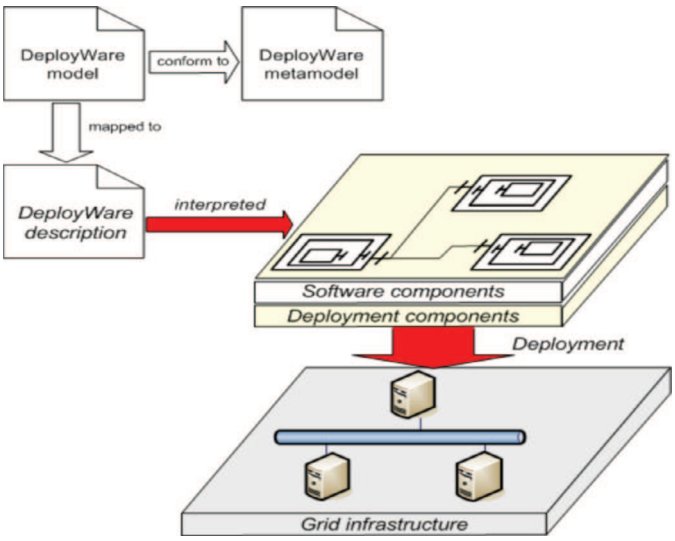

Fig. 18. DeployWare architecture (Flissi et al., 2008).

Main points. DeployWare is a complete solution with integrated tools, which supports reliable and scalable deployment of distributed software systems on large-scale networks. However, it does not address unstable or open environments.

The deployment system is distributed and the DeployWare runtime must be present on the target machines (bootstrap) in order to locally manage the deployment in a decentralized way.

DeployWare brings deployment to a high level of abstraction, using a metamodel and an ADL, hiding to the developer the complexity of the deployment orchestration. Among the models, the "hosts" part explicits the domain and the "software" part explicits the software to deploy and on which host. Several stakeholders may contribute separately to the expression of the deployment: system administrators, software experts, and end-users (end-users define the deployment plan). Due to the separation of concerns, each of them expresses deployment properties by relying on their business skills.

4.5.2. ADME

Problem. Deployment of component-based distributed applications poses two main problems: design of the initial deployment and evolution at runtime when faced with host failures and other disturbances. As these problems are too complex to be handled by humans, Dearle et al. aim at automatizing deployment and applying the autonomic loop defined by Kephart and Chess (2003).

Solution. Dearle et al. propose ADME (Autonomic Deployment and Management Engine), a framework for deployment and autonomic management of component-based distributed applications (Dearle et al., 2004). ADME relies on Deladas (Declaratory Language for Describing Autonomic Systems), a DSL that supports the expression of deployment properties using constraints. Constraints are processed by a constraint solver that generates a configuration (a deployment plan) encoded in XML. An autonomic management engine both handles the initial deployment and adapts the application to changing circumstances at runtime. To feed the autonomic loop, applications are instrumented with probes that locally monitor the execution and generate events. Events are collected and processed by a centralized component. In case of host failure or other disturbance, this component tries to find a new configuration and orchestrates the overall adaptation process. Except for some trivial solutions, it has to launch once again the constraint solver to find another deployment plan. However, in order to minimize redistribution, the problem is first more constrained than the original one: the part of the current mapping that is not involved in the situation to be corrected is also given as a constraint. Afterward, these constraints are progressively removed until an appropriate plan is found.

Main points. ADME is a solution for the deployment of software systems composed of any-type components on networks in a context of evolutivity and volatility of resources.

From an operational point of view, ADME focuses on the autonomy of the system in charge of deployment: it supports initial deployment (installation and activation), and reorganization and redistribution at runtime, without requiring human intervention. Monitoring is decentralized but the computation of new configurations is centralized and the centralized solver is required in most cases. The ADME runtime system relies on a particular kind of middleware, called Cingal, which must be installed on every machine of the domain (bootstrap).

Concerning design, Deladas language provides a high degree of expressiveness to designers: they are not forced to express the plan, but only constraints and properties about the components and the hosts, from which the plan is automatically computed. Therefore, they should have skills in software management, but benefit from a DSL.

4.5.3. TUNe

Problem. Grid computing environments are large-scale, dynamic and heterogeneous, and consequently complex. Centralized deployment and management of distributed applications is no more feasible: decentralization and autonomy of runtime management are required. Besides, expressiveness is also demanded in the description of deployment properties.

Solution. Toure et al. propose TUNe, an autonomic management system dedicated to decentralized deployment of large-scale component-based systems on large-scale grid environments (Broto et al., 2008; Toure et al., 2010). Software components are encapsulated within Fractal components, using a wrapping description language (called WDL) which allows to specify the interface that components provide to the TUNe runtime management platform. A TUNe instance is deployed on every machine and the management system is organized hierarchically to efficiently deploy the Fractal components. Deployment is handled in a decentralized manner. On the machines, specific probes generate events which trigger deployment operations. Events are treated locally, except if they concern entities managed by another machines.

Moreover, UML profiles support the description of the application, a tree-based representation of the domain (i.e. the grid environment), the description of the decentralized administration policy, and statecharts which define the workflow to start or redeploy the application. Deployment relies on mappings between the domain diagram, the administration diagram, and the application diagram.

Main points. TUNe manages component-based applications distributed over heterogeneous and dynamic grids. It targets scalability both at the software and the domain levels.

The deployment activities handled are installation (including delivery and configuration), activation, and redistribution. At runtime, specific probes monitor the domain and the events they generate are treated locally as much as possible. Then, redistribution is performed autonomously by the TUNe instances in a decentralized way, without requiring human participation. In order to function, TUNe and the DIET grid middleware must be available on every machine (bootstap).

The deployment plan is not expressed by the designer, but results from a mapping between models (among them, a model must describe the grid as a hierarchy of nodes). These models should be given by experts.

4.5.4. SmartFrog

Problem. Initial deployment and dynamic management of large-scale distributed systems of components on grid infrastructures (composed of multiple heterogeneous machines) lack simplicity and robustness. Manually deploying grid applications do not scale when the grid grows. Additionally, as different frameworks may be involved for component configuration and lifecycle management or failure handling, the deployment data from different stakeholders (component programmers, integrators, deployment managers, etc.) may be scattered and repeated, leading to inconsistency and misunderstandings.

Solution. Sabharwal and Goldsack et al. propose the SmartFrog framework for configuration-driven deployment on grid infrastructures (Goldsack et al., 2009). SmartFrog allows to describe and configure distributed software systems as collections of cooperating components, using a specific component model (Sabharwal, 2006). Configuring components means setting the links with other components and other parameters (organization) and the hosting device (distribution). SmartFrog supports automatic installation, activation and management of components in Java virtual machines (JVM). The SmartFrog deployment infrastructure is also component-based and distributed, and uses Java RMI for remote interactions. Its components provide services for the delivery and maintenance of the running software systems (management of component lifecycle and failures).

Basically, a SmartFrog daemon runs on every machine of the deployment domain. Initially, daemons are deployed in a centralized way from a master machine using protocols such as ftp and ssh. Then, the deployment infrastructure is deployed using the daemons, and finally the software system.

SmartFrog offers a framework to express configuration data in a consistent way using a hierarchical data model and templates. It allows discovery of configuration parameters at runtime and late binding (therefore, adaptation to context facilities). Additionally, SmartFrog allows the definition of software configuration lifecycle managers. Lifecycle managers properly configure components or groups of components according to the configuration data. Each of them implements a set of methods in order to take care of, for example, installation, activation, termination, failure notification and status checking.

SmartFrog framework provides some other advanced features. Deployment designers may express the deployment plan by giving location exact values or only properties (such as proximity to another component or sub-system). Deployment may dynamically adapt to changing circumstances. Concerning dependability, validation structures may be included in the configurations in order to check particular assumptions. Additionally, in order to prevent attacks, SmartFrog proposes a security model based on security domains and a certification authority.

Main points. SmartFrog is an advanced deployment framework, dedicated to grid infrastructures. It manages the deployment of distributed systems of components, based on a specific component model. As systems change over time, it is possible to add or remove components at runtime.

SmartFrog supports several lifecycle activities: installation and deinstallation, activation and deactivation, reorganization and redistribution. Deployment is distributed, decentralized and scalable. No particular bootstrap needs to be present initially on the machines.

SmartFrog users have a language for describing component collections and component configuration parameters, with mechanisms for composition and extension. When expressing configurations, component locations may be defined in an abstract way by some property, avoiding so a complete description of the plan. In practice, designers should be experts in configuration and lifecycle management.

4.5.5. ORYA

Problem. Application deployment on large sets of machines is complex and cannot be performed by hand. Advanced but ad hoc strategies are commonly used in companies by system administrators. Difficulties arise in their expression and implementation.

Solution. Cunin et al. propose a framework called ORYA (Open enviRonment to deploY Applications) for designing and driving property-based specialized deployment strategies (Cunin et al., 2005).

A strategy is attached to a machine or a group of machines and is applied to a set of deployable units. It may concern the selection of deployment units and their versions, the ordering of deployment operations, or the handling of exceptions, and it takes into account the properties and constraints of the domain and of the application. It is defined in a declarative way by a 3-tuple *<Activity, LogicalExpression, Choice>*. *Activity* specifies a deployment activity as defined in Section 2 (but it is limited to installation and update). When carrying out the specified activity, the *LogicalExpression* is evaluated for all the current deployable units, dividing them into two subsets: the "true set" and the "false set". *Choice* defines rules for the execution of the activity for each subset. In addition to the basic behavior, a strategy is defined by some features such as scope (related to the

domain), visibility and precedence in order to avoid ambiguity in case of concurrent strategies, etc.

Main points. ORYA focus on the expression of deployment strategies, in particular those concerning the choice and the ordering of the operations over the domain. Basically, the application is monolithic and the domain is a local network of a company, which may be divided into sub-domains.

The concerned activities are mainly installation and update, and the process is centralized.

Strategies are explicitly dedicated to machines. The deployment domain and the deployment units must be explicited, with dependencies, constraints, and deployment rules. Thus, ORYA users must be experts in deployment.

5. Synthesis and conclusion

The aim of this article is to review different research works on automatic deployment. The contribution is triple. First, we set an up-to-date terminology related to software deployment. Then, we propose a framework in order to analyze works on automatic deployment highlighting the following points: nature of the software, software changeability, nature of the domain, number and scalability, dynamics of the domain, activities, control, bootstrap, nature of the specification, domain transparency, designer skills. Finally, we review the state of the art of automatic deployment using the analytical framework. Section 5.1 summarizes the review. It shows that existing solutions are incomplete, and possibly inefficient or unusable, when distribution, heterogeneity, scalability, dynamics and openness are primary concerns. Then, Section 5.2 concludes and presents some open issues that should be addressed in the next future.

5.1. Synthesis

The synthesis of the survey is organized in four parts corresponding to the four basic questions concerning deployment, according to the analytical framework (see Section 3). In the four parts, the highlighted points are considered separately and, for each of them, a table summarizes the analysis and shows the coverage of the nineteen reviewed solutions.

5.1.1. Synthesis in relation to the software deployed

Many solutions target component-based distributed systems as Table 1a shows. Half of them are dedicated to a particular type of software component, while the other half do not make any hypothesis regarding this point (see Table 1b). However, among the latter, some solutions require that any-type components are wrapped into ones of a specific type, such as TUNe does by wrapping components into Fractal ones before deploying them. Note that Disnix and DeployWare take into account software heterogeneity by using models and model transformations. Besides, we can observe that only one solution (SmartFrog) really address the scalability issue (in number of components).

Table 2 shows that few solutions consider the changeability of the software system (that is to say, components that change, enter or leave the system at runtime), or in a limited way.

5.1.2. Synthesis in relation to the deployment domain

Most works address deployment issues in the context of networked machines whereas some focus on grid or cloud infrastructures, as Table 3 shows. They generally suppose a particular type of network and target one kind of device (personal computers in most cases, smartphones, etc.). The heterogeneity issue is especially addressed by Disnix, DeployWare, and TUNe, while Kalimucho and Cloudlet focus on the limitation of the resources of the devices (and support transfer of computations from devices with limited capacity to more powerful computers) as well as the solutions for WSNs such as QARI and DVM.

In Table 4, we can observe that few solutions address large-scale deployment (number of devices and scalability), or in a very limited way *via* partial decentralization of deployment operations.

Table 1
Nature of the software.

	(a) Structure	
	Monolithic application	Component-based application
FROGi	√	
R-OSGi		√
Soft. Dock	√	
QUIET	√	
Disnix		√
RAC		√
CoRDAGe		√
Wrangler		√
DVM	√	
QARI		√
Kalimucho		√
StarCCM		√
Codewan		√
Cloudlet	√	
DeployWare		√
ADME		√
TUNe		√
SmartFrog		√
ORYA	√	

	(b)Type of the component(s)
	Technology
FROGi	Fractal-OSGi
R-OSGi	OSGi
Soft. Dock	*Any*
QUIET	SUIPM, MS Windows apps
Disnix	*Any*
RAC	VM images
CoRDAGe	*Any*
Wrangler	VM images, configuration scripts
DVM	Binary code
QARI	LooCI
Kalimucho	Osagaia-Korrontea
StarCCM	CCM
Codewan	*Any*
Cloudlet	*Any*
DeployWare	*Any*
ADME	*Any*
TUNe	*Any*
SmartFrog	SmartFrog component model
ORYA	*Any*

Table 2
Handled changeability (software system).

	Limited	Advanced
FROGi	OSGi features	
R-OSGi	OSGi features	
Soft. Dock	Update	
QUIET		
Disnix		Atomic upgrade and rollback
RAC	(Re)configuration	
CoRDAGe		Resource requirements
Wrangler		Provisioning of resources
DVM		Update, reconfiguration
QARI	LooCI features	
Kalimucho		Creation, removal, update, composition
StarCCM		
Codewan		
Cloudlet		
DeployWare		
ADME		
TUNe		
SmartFrog		Creation, removal, failure
ORYA	Update	

Table 3
Nature of the domain.

	Local (remote)	Network	WSN	Spontaneous network	Grid	Cloud
FROGi	✓					
R-OSGi		✓				
Soft. Dock		✓				
QUIET		✓				
Disnix		✓				
RAC						✓
CoRDAGe					✓	
Wrangler						✓
DVM			✓			
QARI			✓			
Kalimucho		✓				
StarCCM		✓				
Codewan				✓		
Cloudlet				✓		
DeployWare					✓	
ADME		✓				
TUNe					✓	
SmartFrog					✓	
ORYA		✓				

Table 4
Number of devices and domain-related scalability.

	Limited	Advanced
FROGi		
R-OSGi	✓	
Soft. Dock	✓	
QUIET		✓
Disnix		
RAC		
CoRDAGe		✓
Wrangler	✓	
DVM	✓	
QARI	✓	
Kalimucho		
StarCCM		
Codewan		
Cloudlet		
DeployWare		✓
ADME	✓	
TUNe		✓
SmartFrog		✓
ORYA		

Table 5
Handled dynamics (deployment domain).

	Limited	Advanced
FROGi		
R-OSGi	Failures	
Soft. Dock	Network connectivity	
QUIET	Resource availability	
Disnix		Failures, connections, disconnections
RAC		
CoRDAGe		Resource availability and quality
Wrangler		Cloud dynamics
DVM		Designer-defined events
QARI		QoS, resilience to node failure and mobility
Kalimucho		Connections, disconnections, QoS
StarCCM		Context
Codewan		Disconnections, network fragmentation
Cloudlet	Mobility *(manually)*	
DeployWare		
ADME		Host failures and disturbances
TUNe		Grid dynamics
SmartFrog		Failures, resource availability
ORYA		

Table 6
Covered activities.

	Rel.	Inst.	Act.	Upd.	Reorg.	Redist.	Deact.	Deinst.	Ret.
FROGi		✓	✓	✓	✓		✓	✓	
R-OSGi		✓	✓	✓	✓		✓	✓	
Soft. Dock	✓	✓	✓	✓	✓		✓	✓	✓
QUIET	✓	✓						✓	
Disnix	✓	✓	✓	✓	✓	✓	✓	✓	
RAC	✓	✓	✓		✓				
CoRDAGe		✓	✓		✓	✓	✓	✓	
Wrangler		✓			✓	✓			
DVM		✓		✓	✓				
QARI		✓				✓			
Kalimucho		✓		✓	✓	✓		✓	
StarCCM						✓			
Codewan	✓	✓							
Cloudlet		✓	✓				✓	✓	
DeployWare		✓	✓				✓	✓	
ADME		✓	✓		✓	✓			
TUNe		✓	✓			✓			
SmartFrog		✓	✓		✓	✓	✓	✓	
ORYA		✓		✓					

Dynamics of the domain mainly concerns connections and disconnections (especially for spontaneous networks), failures of machines and communication links, and variations of the quality of service. Table 5 points out if and how far the solutions handle the dynamics of the deployment domain.

Note that some solutions focus on heterogeneity, while others focus on scalability or dynamics, but except for TUNe, none of the propositions considers heterogeneity, scalability and dynamics at the same time. In a general way, the solutions do not support dynamic reaction to unforeseeable events, and dynamic modification of the plan is limited or centralized (but Software Dock, QUIET or Codewan for example propose decentralized and local autonomous adaptation).

5.1.3. Synthesis in relation to the realization of deployment

This section summerizes the impacts of the solutions in terms of deployment activities, architecture of the system in charge of the deployment and decentralization of the control, and deployment bootstrap.

Table 6 shows what activities are handled by the different solutions. The abbreviations Rel., Inst., Act., Upd., Adapt., Reorg., Redist., Deact., Deinst., and Ret. refer respectively to release, installation, activation, update, reorganization, redistribution, deactivation, deinstallation, and retire. Installation and activation (and, to a certain extent, deactivation and deinstallation) are the most commonly handled.

The issues related to the dynamics are addressed in different ways or not at all. Several solutions target the dynamic modification of the deployment plan (Disnix, Kalimucho, StarCCM, ADME), in some cases by means of resource allocation techniques (CoRDAGe, Wrangler). Other solutions target reorganization or transfer of computations (RAC, Cloudlet, TUNe, SmartFrog), or component delivery (Codewan).

Several solutions focus on the distributed architecture of the system that supports the deployment. They target decentralization (Software Dock, QUIET, Wrangler, Kalimucho, Codewan), context-awareness (StarCCM), self-adaptation and autonomy (Disnix, ADME).

Table 7 indicates how the deployment process is controlled and shows that some solutions are fully decentralized (Software Dock, QUIET, DVM, Kalimucho, Codewan, DeployWare, TUNe, SmartFrog). Note that ORYA is dedicated to the expression of deployment strategies in order to support the specification of the deployment process.

In practice, the systems in charge of the deployment rely on bootstraps. Most of the solutions assume the availability of the bootstrap on the machines without considering this question precisely. Table 8 shows the nature of the bootstrap. Many solutions rely on a specific one (that is to say, a bootstrap developed on purpose). Some rely

Table 7
Control.

	Centralized	Decentralized
FROGi	*n/a*	*n/a*
R-OSGi	*n/a*	*n/a*
Soft. Dock		√
QUIET		√
Disnix	√	
RAC		√ (partially)
CoRDAGe	√	
Wrangler		√ (partially)
DVM		√
QARI	√	
Kalimucho		√
StarCCM	√	
Codewan		√
Cloudlet	*n/a*	*n/a*
DeployWare		√
ADME		√ (partially)
TUNe		√
SmartFrog		√
ORYA	√	

Table 8
Nature of the bootstrap.

	Specific	Non specific	Standard
FROGi	FrogiBundleActivator	OSGi platform	
R-OSGi		OSGi platform	
Soft. Dock	Field dock	Voyager platform	
QUIET		JADE platform	
Disnix	DisNixService		
RAC			Virtualization platform
CoRDAGe		Grid middleware, ADAGE	
Wrangler			Virtualization platform
DVM	DVM basic library		
QARI	QARI/LooCI platform		WSN deployment tools
Kalimucho	Kalimucho platform		
StarCCM	StarCCM middleware		
Codewan	Codewan platform		
Cloudlet	baseVM and KCM		
DeployWare	DeployWare runtime		
ADME	Cingal infrastructure		
TUNe	TUNe runtime, DIET		
SmartFrog			Java, ftp, ssh
ORYA	*n/a*	*n/a*	*n/a*

Table 9
Nature of the specification.

	Property-based
FROGi	
R-OSGi	
Soft. Dock	
QUIET	
Disnix	√
RAC	
CoRDAGe	√
Wrangler	√
DVM	
QARI	√
Kalimucho	√
StarCCM	√
Codewan	
Cloudlet	
DeployWare	√
ADME	√
TUNe	√
SmartFrog	√
ORYA	

Table 10
Domain transparency.

	Domain transparency
FROGi	
R-OSGi	
Soft. Dock	√
QUIET	
Disnix	√
RAC	√ (cloud)
CoRDAGe	
Wrangler	√ (partial)
DVM	
QARI	√
Kalimucho	√
StarCCM	
Codewan	√
Cloudlet	
DeployWare	
ADME	
TUNe	
SmartFrog	√
ORYA	

on non-specific tools or platforms, but few of them (RAC, Wrangler, SmartFrog) are really technology-independent.

5.1.4. Synthesis in relation to design and expressiveness

Section 2.1 has presented different roles played by various stakeholders. Here, the focus is placed on expressiveness and on the level of abstraction that designers can benefit from, when they are expressing the deployment properties. Indeed, providing abstraction and expressiveness to deployment design is a major challenge, especially concerning the plan and the domain.

Table 9 indicates if the designer has the possibility to only express a set of properties (constraints, choices, or configuration properties from which the plan can be computed or dynamically adapted) without being forced to make the plan explicit. In practice, the expression of the deployment properties relies on XML (QARI, Wrangler) or on domain-specific languages (ADME), and in some cases on the rule-based programming style (StarCCM). Several solutions (Disnix, CoRDAGe, DeployWare, TUNe) propose the use of models and transformation of models. Models bring a high level of abstraction to designers, and facilitate role-oriented separation of concerns.

Table 10 shows that the impact of the solutions is very limited regarding domain transparency, that is to say that few solutions allow deployment specification without explicit designation of the devices. If so, the solution supports dynamic domain discovery and/or late binding between components and devices.

Table 11 points out the required skills of the deployment managers. Few technologies are usable by inexperienced people, and most of them are used by specialists, which play the roles of deployment designer, deployment operator, and system administrator (assuming that they control the resources). Thus, specifying deployment usually remains out of reach for ordinary users.

5.2. Conclusion

Modern software systems are increasingly large-scale, distributed, ubiquitous, mobile, and heterogeneous. Dynamics and instability become usual due to mobility of devices and users, to openness of both the deployed software system and the network of hosts, and more generally to dynamic variations of the availability and the quality of resources and services. Besides, these systems may involve a mass of devices, users, or software versions, with their heterogeneity. In such a context, deploying software systems is a complex and challenging task. While traditional deployment most often requires human intervention, deployment of modern distributed software systems demands an increasing level of automation and autonomy.

Table 11
Deployment designer skills.

	Required expertise
FROGi	Fractal ADL, OSGi
R-OSGi	OSGi
Soft. Dock	Deployment, DSD language
QUIET	Deployment
Disnix	System administration, modeling
RAC	VA production and configuration
CoRDAGe	*Low*
Wrangler	Deployment, virtualization, administration
DVM	Deployment, low-level features
QARI	QARI, LooCI *(low)*
Kalimucho	Kalimucho system
StarCCM	Deployment, adaptation
Codewan	Release
Cloudlet	*Low*
DeployWare	*In accordance with the designer's skills*
ADME	Application management and deployment
TUNe	Deployment, design of deployment models
SmartFrog	Configuration, component lifecycle management
ORYA	Deployment activities and strategies

Besides, as complexity grows, deployment demands appropriate methods, processes and tools which provide both expressiveness and abstraction in design, and control and automation in the realization. The main requirements are high-level expression of deployment properties, plan generation from the specification, automatic achievement of the deployment plan, plan adaptation at runtime, continuous and incremental deployment, decentralized and context-aware management. In our opinion, answering all of the requirements and enabling automatic and autonomic deployment is becoming a major issue in the domain of software engineering and management. The state of the art provides various solutions but the problem remains widely open.

It is worth to notice that several relevant questions have little been addressed from now. One of them concerns security and privacy. Security and privacy might provide additional deployment requirements that actual solution do not take into account. The exception is SmartFrog, which proposes a security model based on security domains. Besides, the solutions generally disregard the problem of multiple administration domains and most often assume that the system in charge of the deployment has the permission to operate on the machines.

Designation and identification of devices or sub-domains is another challenging issue. For example, in wide-area domains, it could be useful to consider proximity in order to specify deployment on the devices of a particular sub-network, of a given geographical area, or owned by the members of a social network. This question is left open even if Kalimucho, Codewan or Cloudlet ultimately take opportunistically advantage of spatial proximity between devices.

Concerning the achievement of deployment, a main point is reliability, and more generally deployment based on quality attributes (non-functional properties). These issues are addressed by Disnix, QARI, DeployWare and SmartFrog, and by ADME (optimization of the plan). Efficiency is addressed by few works: Software Dock, QUIET, Wrangler, DVM, QARI, and TUNe. Quality of the deployment also relies on context-awareness; this question is addressed by StarCCM, ADME and TUNe.

At last, another set of questions concerns the impact of deployment on the running application (Kalimucho supports the continuity of services), flexibility of the deployment process and scheduling of deployment operations. On the last point, the impact of the state of the art is almost non-existent: only ORYA allows the definition of deployment strategies and operation ordering.

In our opinion, these questions are among the main ones that should be addressed in the next future.

Acknowledgments

This work is part of the French National Research Agency (ANR) project INCOME (INCOME, 2012) (ANR-11-INFR-009, 2012–2015). The authors thank all the members of the project that contributed directly or indirectly to this paper.

References

ADAGE, 2007. Adage: an automatic deployment tool. http://adage.gforge.inria.fr (last accessed 11.14).

Ansible, Inc., 2014. Ansible is the simplest way to automate IT. http://www.ansible.com/home (last accessed 11.14).

Ashton, K., 2009. That 'Internet of Things' thing. http://www.rfidjournal.com/articles/view?4986 (RFID J., September 2009, last accessed 11.14).

Bailey, E.C., 2000. Maximum RPM. SAMS Publishing.

Balani, R., Han, C.C., Rengaswamy, R.K., Tsigkogiannis, I., Srivastava, M., 2006. Multi-level software reconfiguration for sensor networks. In: Proceedings of the 6th ACM & IEEE International Conference on Embedded Software (EMSOFT). ACM, New York, NY, USA, pp. 112–121. doi:10.1145/1176887.1176904.

Brandtzæg, E., Parastoo, M., Mosser, S., 2012. Towards a domain-specific language to deploy applications in the clouds. In: 3rd International Conference on Cloud Computing, GRIDs, and Virtualization (Cloud Computing 2012). IARIA, pp. 213–218.

Broto, L., Hagimont, D., Stolf, P., Palma, N.D., Temate, S., 2008. Autonomic management policy specification in tune. In: Wainwright, R.L., Haddad, H. (Eds.), 23rd Annual Symposium on Applied Computing (SAC 2008). ACM, pp. 1658–1663. doi:10.1145/1363686.1364080.

Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B., 2006. The FRACTAL component model and its support in Java. Software: Pract. Exp. 36 (11–12), 1257–1284. doi:10.1002/spe.767.

Carzaniga, A., Fuggetta, A., Hall, R.S., Heimbigner, D., van der Hoek, A., Wolf, A.L., 1998. A characterization framework for software deployment technologies. (Technical report). Defense Technical Information Center (DTIC) Document.

Cassagnes, C., Roose, P., Dalmau, M., 2009. Kalimucho: software architecture for limited mobile devices. ACM SIGBED Rev. 6(3), 12.

Chef Software, Inc., 2014. Automation for web-scale IT. https://www.chef.io/ (last accessed 11.14).

Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., Sivaharan, T., 2008. A generic component model for building systems software. ACM Trans. Comput. Syst. 26(1), 1–42. doi:10.1145/1328671.1328672.

Coulson, G., Grace, P., Blair, G., Cai, W., Cooper, C., Duce, D., Mathy, L., Yeung, W.K., Porter, B., Sagar, M., Li, W., 2006. A component-based middleware framework for configurable and reconfigurable grid computing. Concurr. Comput.: Pract. Exp. 18(8), 865–874. doi:10.1002/cpe.v18:8.

Crnkovic, I., Sentilles, S., Vulgarakis, A., Chaudron, M.R.V., 2011. A classification framework for software component models. IEEE Trans. Softw. Eng. 37(5), 593–615. doi:10.1109/TSE.2010.83.

Cudennec, L., Antoniu, G., Bougé, L., 2008. CoRDAGe: towards transparent management of interactions between applications and resources. In: International Workshop on Scalable Tools for High-End Computing (STHEC 2008), pp. 13–24.

Cunin, P.Y., Lestideau, V., Merle, N., 2005. Orya: a strategy oriented deployment framework. In: Dearle, A., Eisenbach, S. (Eds.), Component Deployment, Lecture Notes in Computer Science, vol. 3798. Springer, Berlin, Heidelberg, pp. 177–180. doi:10.1007/11590712_14.

Dearle, A., 2007. Software deployment, past, present and future. In: Briand, L.C., Wolf, A.L. (Eds.), Workshop on the Future of Software Engineering (FOSE 2007), pp. 269–284. doi:10.1145/1253532.1254724.

Dearle, A., Kirby, G.N.C., McCarthy, A.J., 2004. A framework for constraint-based deployment and autonomic management of distributed applications. In: International Conference on Autonomic Computing (ICAC'04). IEEE Computer Society, pp. 300–301.

Desertot, M., Cervantes, H., Donsez, D., 2006. FROGi: fractal components deployment over OSGi. In: Löwe, W., Südholt, M. (Eds.), Software Composition. Springer, pp. 275–290. doi:10.1007/11821946_18.

Flinn, J., 2012. Cyber Foraging: Bridging Mobile and Cloud Computing. Synthesis Lectures on Mobile and Pervasive Computing, Morgan & Claypool Publishers. doi:10.2200/S00447ED1V01Y201209MPC010.

Flissi, A., Dubus, J., Dolet, N., Merle, P., 2008. Deploying on the grid with deployware. In: CCGRID. IEEE Computer Society, pp. 177–184. doi:10.1109/CCGRID.2008.59.

Fok, C.L., Roman, G.C., Lu, C., 2005. Rapid development and flexible deployment of adaptive wireless sensor network applications. In: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS). IEEE Computer Society, pp. 653–662.

Goldsack, P., Guijarro, J., Loughran, S., Coles, A.N., Farrell, A., Lain, A., Murray, P., Toft, P., 2009. The SmartFrog configuration management framework. Oper. Syst. Rev. 43(1), 16–25. doi:10.1145/1496909.1496915.

Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M., 2013. Internet of Things (IoT): a vision, architectural elements, and future directions. Future Gener. Comput. Syst. 29(7), 1645–1660. doi:http://dx.doi.org/10.1016/j.future.2013.01.010.

Guidec, F., Sommer, N.L., Mahéo, Y., 2010. Opportunistic software deployment in disconnected mobile ad hoc networks. Int. J. Handheld Comput. Res. 1(1), 24–42. doi:10.4018/jhcr.2010090902.
Hall, R.S., Heimbigner, D., Wolf, A.L., 1999. A cooperative approach to support software deployment using the software dock. In: Boehm, B.W., Garlan, D., Kramer, J. (Eds.), International Conference on Software Engineering. ACM, pp. 174–183.
Haller, P., Odersky, M., 2006. Event-based programming without inversion of control. In: Lightfoot, D., Szyperski, C. (Eds.), Modular Programming Languages, Lecture Notes in Computer Science, vol. 4228. Springer, Berlin, Heidelberg, pp. 4–22.
Heydarnoori, A., 2008. Deploying component-based applications: tools and techniques. In: Lee, R. (Ed.), Software Engineering Research, Management and Applications, Studies in Computational Intelligence, vol. 253. Springer-Verlag Prague, Czech Republic, pp. 29–42. doi:10.1007/978-3-540-70561-1_3.
Horré, W., Michiels, S., Joosen, W., Hughes, D., 2011. Advanced sensor network software deployment using application-level quality goals. J. Softw. 6(4), 528–535.
Hosek, P., Cadar, C., 2013. Safe software updates via multi-version execution. In: Proceedings of the 2013 International Conference on Software Engineering (ICSE). IEEE Computer Society, Piscataway, NJ, USA, pp. 612–621.
Hughes, D., Thoelen, K., Maerien, J., Matthys, N., Horré, W., del Cid, J., Huygens, C., Michiels, S., Joosen, W., 2012. Looci: the loosely-coupled component infrastructure. In: 11th IEEE International Symposium on Network Computing and Applications (NCA 2012). IEEE Computer Society, pp. 236–243.
INCOME, 2012. The INCOME project. www.anr-income.fr (last accessed 11.14).
Juve, G., Deelman, E., 2011. Automating application deployment in infrastructure clouds. In: IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom 2011), pp. 658–665. doi:10.1109/CloudCom.2011.102.
Kephart, J.O., Chess, D.M., 2003. The vision of autonomic computing. Computer 36(1), 41–50. doi:10.1109/MC.2003.1160055.
Kessis, M., Roncancio, C., Lefebvre, A., 2009. DASIMA: a flexible management middleware in multi-scale contexts. In: 6th International Conference on Information Technology: New Generations (ITNG'09), pp. 1390–1396. doi:10.1109/ITNG.2009.338.
Levis, P., Patel, N., Culler, D., Shenker, S., 2004. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In: Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI), pp. 15–28.
Liu, H., 2011. Rapid application configuration in amazon cloud using configurable virtual appliances. In: Chu, W.C., Wong, W.E., Palakal, M.J., Hung, C.C. (Eds.), 26th Symposium on Applied computing (SAC 2011). ACM, pp. 147–154. doi:10.1145/1982185.1982221.
Louberry, C., Roose, P., Dalmau, M., 2011. Kalimucho: contextual deployment for QoS management. In: Felber, P., Rouvoy, R. (Eds.), Distributed Applications and Interoperable Systems (DAIS 2011), pp. 43–56. doi:10.1007/978-3-642-21387-8_4.
Manzoor, U., Nefti, S., 2008. Silent unattended installation package manager—SUIPM. In: Mohammadian, M. (Ed.), International Conference on CIMCA/IAWTIC/ISE. IEEE Computer Society, pp. 55–60. doi:10.1109/CIMCA.2008.202.
Manzoor, U., Nefti, S., 2010. QUIET: a methodology for autonomous software deployment using mobile agents. J. Netw. Comput. Appl. 33(6), 696–706. doi:10.1016/j.jnca.2010.03.015.
Marrón, P.J., Gauger, M., Lachenmann, A., Minder, D., Saukh, O., Rothermel, K., 2006. Flexcup: a flexible and efficient code update mechanism for sensor networks. In: Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN). Springer, pp. 212–227.
Matougui, M.E.A., Leriche, S., 2012. A middleware architecture for autonomic software deployment. In: The 7th International Conference on Systems and Networks Communications (ICSNC'12). XPS, Lisbon, Portugal, pp. 13–20. 12619.
Miorandi, D., Sicari, S., Pellegrini, F.D., Chlamtac, I., 2012. Internet of things: vision, applications and research challenges. Ad Hoc Netw. 10(7), 1497–1516. doi:10.1016/j.adhoc.2012.02.016.
Object Management Group, 2006a. Corba Component Model (CCM). http://www.omg.org/spec/CCM (last accessed 11.14).
Object Management Group, 2006b. Deployment and configuration of component-based distributed applications specification, Version 4.0. http://www.omg.org/spec/DEPL (last accessed 11.14).
Octopus Deploy Pty. Ltd., 2014. Automated deployment for .NET. http://octopusdeploy.com (last accessed 11.14).
Oracle, 2013a. Enterprise JavaBeans (EJB). http://www.oracle.com/technetwork/java/javaee/ejb/index.html (last accessed 11.14).
Oracle, 2013b. JavaBeans tutorial. http://docs.oracle.com/javase/tutorial/javabeans/index.html (last accessed 11.14).
OSGi Alliance, 2009. OSGi service platform core specification, Release 3. Version 4.2 (Technical report).
OW2 Consortium, 2009. The FRACTAL project. http://fractal.ow2.org/documentation.html (last accessed 11.14).
Puppet Labs, 2014. Automate IT. http://puppetlabs.com/ (last accessed 11.14).
Rellermeyer, J.S., Alonso, G., Roscoe, T., 2007. R-OSGi: distributed applications through software modularization. In: Cerqueira, R., Campbell, R.H. (Eds.), 8th International Middleware Conference. Springer, pp. 1–20. doi:10.1007/978-3-540-76778-7_1.
Sabharwal, R., 2006. Grid infrastructure deployment using SmartFrog technology. In: International Conference on Networking and Services (ICNS 2006). IEEE Computer Society, p. 73. doi:10.1109/ICNS.2006.54.
SARAH, 2005. SARAH – delay-tolerant distributed services for mobile ad hoc networks. http://www-valoria.univ-ubs.fr/SARAH (last accessed 2014).
Satyanarayanan, M., Bahl, P., Cáceres, R., Davies, N., 2009. The case for VM-based cloudlets in mobile computing. IEEE Pervasive Comput. 8(4), 14–23. doi:10.1109/MPRV.2009.82.
SELFWARE, 2005. The SELFWARE project. http://sardes.inrialpes.fr/~boyer/selfware (last accessed 11.14).
Sledziewski, K., Bordbar, B., Anane, R., 2010. A DSL-based approach to software development and deployment on cloud. In: 24th IEEE International Conference on Advanced Information Networking and Applications (AINA 2010). IEEE Computer Society, pp. 414–421. doi:10.1109/AINA.2010.81.
Strembeck, M., Zdun, U., 2009. An approach for the systematic development of domain-specific languages. Softw.: Pract. Exp. 39(15), 1253–1292. doi:10.1002/spe.936.
Szyperski, C., 2002. Component Software: Beyond Object-Oriented Programming, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
Toure, M., Stolf, P., Hagimont, D., Broto, L., 2010. Large scale deployment. In: 6th International Conference on Autonomic and Autonomous Systems (ICAS). IEEE Computer Society, pp. 78–83. doi:10.1109/ICAS.2010.20.
USENET, 2007. UseNet – ubiquitous M2M service networks. https://itea3.org/project/usenet.html (last accessed 11.14).
van der Burg, S., Dolstra, E., 2010. Automated deployment of a heterogeneous service-oriented system. In: 36th EUROMICRO Conference on Software Engineering and Advanced Applications. IEEE, pp. 183–190. doi:10.1109/SEAA.2010.10.
van der Burg, S., Dolstra, E., 2011. A self-adaptive deployment framework for service-oriented systems. In: Giese, H., Cheng, B.H.C. (Eds.), 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'11). ACM, pp. 208–217. doi:10.1145/1988008.1988039.
van der Burg, S., Dolstra, E., 2014. Disnix: a toolset for distributed deployment. Sci. Comput. Program. 79, 52–69. doi:10.1016/j.scico.2012.03.006.
van Steen, M., Pierre, G., Voulgaris, S., 2012. Challenges in very large distributed systems. J. Internet Serv. Appl. 3(1), 59–66. doi:10.1007/s13174-011-0043-x.
VMware Inc., 2008. Building the virtualized enterprise with VMware infrastructure. http://www.vmware.com/pdf/vmware_infrastructure_wp.pdf (White paper).
Weiser, M., 1996. Nomadic issues in ubiquitous computing. http://www.ubiq.com/hypertext/weiser/NomadicInteractive/ (slides presented at the NOMADIC'96 conference, last accessed 11.14).
Weiser, M., 1999. The computer for the 21st century. Mobile Comput. Commun. Rev. 3(3), 3–11. doi:10.1145/329124.329126.
Zheng, D., Wang, J., Han, W., Jia, Y., Zou, P., 2006. Towards a context-aware middleware for deploying component-based applications in pervasive computing. In: 5th International Conference on Grid and Cooperative Computing (GCC 2006). IEEE Computer Society, pp. 454–457. doi:10.1109/GCC.2006.92.
Zheng, D., Wang, J., Jia, Y., Han, W., Zou, P., 2007. Deployment of context-aware component-based applications based on middleware. In: Indulska, J., Ma, J., Yang, L.T., Ungerer, T., Cao, J. (Eds.), Ubiquitous Intelligence and Computing (UIC 2007). Springer, pp. 908–918. doi:10.1007/978-3-540-73549-6_89.

Jean-Paul Arcangeli is an Associate Professor (MC, HDR) in Computer Science at the University of Toulouse (France), and he is a director of the master program "Software Development". He is a member of the "Cooperative Multi-Agent Systems" research team at the Institute of Research in Computer Science of Toulouse (IRIT), and a leader of the INCOME project (Multiscale Context Management for the Internet of Things) funded by the French National Research Agency (ANR). His research work mainly concerns distributed software architectures and deployment, component-based and agent-based software engineering, mobile agents, software adaptation and self-adaptation.

Raja Boujbel graduated with a PhD in Computer Science in January, 2015. Her PhD thesis took place in the "Cooperative Multi-Agent Systems" research team at the IRIT Lab., and was directed by Jean-Paul Arcangeli and Sébastien Leriche. Her research work is about autonomic deployment of massively distributed component-based software in a context of dynamic and open systems. It is part of the ANR INCOME project.

Sébastien Leriche is an Associate Professor (MC) in Computer Science at ENAC (the French Civil Aviation University) since 2013. He graduated in 2006 with a PhD in Computer Science from the University of Toulouse and conducted his research as an associate professor in Télécom SudParis (France) for 6 years where he studied distributed systems in ubiquitous environments and middleware for smart objects. His current research interests include distribution and deployment of services for the IoT, domain-specific languages (DSL) and multiscale issues in open distributed systems.