dEVOLVe: Middleware Support for software maintenance in Distributed, EVOLVing Environments

Bart Elen
Sam Michiels
Wouter Joosen
Pierre Verbaeten
Report CW 511, March 2008



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

dEVOLVe: Middleware Support for software maintenance in Distributed, EVOLVing Environments

Bart Elen
Sam Michiels
Wouter Joosen
Pierre Verbaeten
Report CW 511, March 2008

Department of Computer Science, K.U.Leuven

Abstract

Distributed applications in evolving environments are typically difficult to maintain. Currently, human intervention is needed to maintain the application and adapt it to its changed environment. We need to: deploy application software on the added devices (buses), replace application software on the changed devices (e.g. to support the new advertisement display), and remove application software when no longer useful (e.g. because it requires an advertisement display which is removed). When the distributed environment becomes too large, or changes too often, manual software maintenance is no longer a feasible option. Automation is required.

This paper reports on our ongoing work on dEVOLVe, a new OSGi based middleware platform which will automate some software maintenance tasks in 'distributed evolving environments'. dEVOLVe will detect changes in the distributed environment and will adapt the application accordingly. This way, the user is free from software maintenance tasks caused by the environment evolution.

Keywords : Software maintenance, distributed evolving environment, middleware.
CR Subject Classification : C.2.4 Distributed Systems, D.2.11 Software Architectures

1 Introduction

A large amount of devices with computing power (cars, VCR's, cell phones, ...) have been introduced on the market during the last decades. A new trend is that those devices are being equipped with network connectivity. Those newly formed distributed environments provide a promising platform for new distributed applications. An example of such an application is the advertisement application of the IBBT-SPAMM project [3]. In this project, public transportation buses of 'De Lijn' [1] are equipped with an onboard computer, a wireless network connection and an advertisement display. The advertisement client application is executed on the company back-end application server. With this client, advertisements are sent to the advertisement server applications, running on the different buses, which will show them on the advertisement displays of the buses. Figure 1 gives an overview of the distributed environment of the advertisement application.

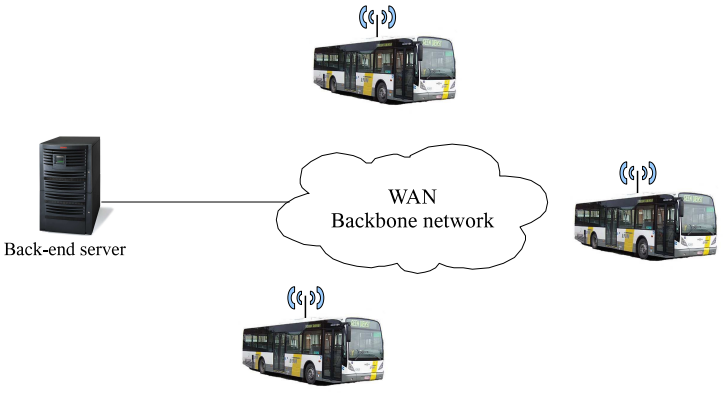

Figure 1: Distributed evolving environment of advertisement application

We can distinguish 3 types of evolution in this distributed environment:

1. Devices (buses) will be added to and removed from the distributed application. The added devices (buses) can differ from the previously used ones in system software (e.g. new Java Virtual Machine (JVM)) and/or hardware (e.g. new type of advertisement displays), introducing heterogeneity in the distributed environment.

2. The employed devices will change over time when peripherals are added to or removed from the devices. For instance, busses which are upgraded after some years of service with new types of advertisement displays.

3. The application software itself will evolve: new features will be added, bugs will be fixed, ...

We focus in this paper on middleware support for the first two types of evolution. Software evolution and application life cycle management in distributed environments are currently not fully covered in our research.

Changes in the distributed environment may require an adaptation of the application software: 1) When a device (bus) is added to the distributed environment, the appropriate application software must be deployed on it. 2) When

a device changes (e.g. a peripheral device is added), it may be required to replace the application software (e.g. to support the new advertisement display). 3) Application software can become useless after a change in the distributed environment (e.g. because it requires an advertisement display which is removed). This application software should be stopped or removed to free resources.

The adaptation of the application software is made harder by the heterogeneity in the distributed environments. Application software imposes requirements to its execution environment. It typically requires a certain JVM, the presence of certain devices or peripherals (e.g. an advertisement display), ... Hence, in a heterogeneous environment, different application software is needed on different devices for the same task.

Human intervention is currently needed to maintain applications in distributed evolving environments; to detect the changes in the distributed environment, to determine the best fitted application software for this environment, and to deploy the selected software on the correct devices. In large distributed environments that change a lot over time, software maintenance is complex, time consuming and error-prone. Therefore, middleware support is needed to automate the software maintenance tasks caused by the evolution of the distributed environment.

The main contribution of this paper is an OSGi [4] based middleware platform, called dEVOLVe, which automates software maintenance tasks in 'distributed evolving environments'. The middleware will adapt distributed applications towards their changing environment by fulfilling the following tasks: 1) Determining the properties of the distributed environment. 2) Detecting the addition and removal of devices in the distributed environment. 3) Detecting when the peripherals of the devices change. 4) Selecting the application software, best fitted for the execution environment. 5) Automatically deploying the selected software on the correct devices. 6) Detecting the presence of software that is no longer needed and removing it from the application.

The remainder of this paper is structured as follows. In section 2, we present the requirements for software maintenance middleware for distributed evolving environments. Section 3 demonstrates how an important part of our middleware requirements can be realized with existing OSGi technology. Section 4 enumerates the major functionalities with which dEVOLVe extends the OSGi service platform. In section 5, we compare our middleware platform with some state-of-the-art middleware platforms for software maintenance in distributed evolving environments. Section 6 concludes this paper and describes future directions.

2 Middleware requirements

In this section, we identify the main requirements for software maintenance middleware for distributed evolving environments. First, we explain why we see a component based application architecture with service dependencies as a pre-requisite. Second, we describe a use case of the advertisement application to identify the middleware requirements.

2.1 Pre-requisite

As a pre-requisite, we require for all applications a component based architecture and the usage of service dependencies [6]. The component based architecture limits the amount of application software that has to be replaced when the execution environment changes. An example is given in Figure 2. When a large advertisement display is added to the execution environment, the monolithic application architecture requires the replacement of the complete application. In case of a component based architecture however, only the adaptation of a part of the application is required. The usage of service bindings between the components allows the application to be rewired at-runtime.

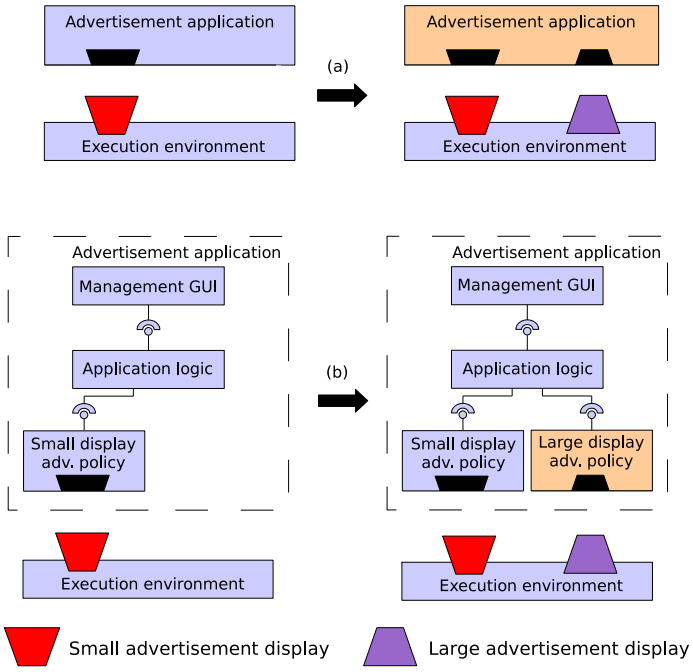

Figure 2: A monolithic (a) and a component based (b) application architecture in an evolving environment

2.2 Advertisement application use case

We identify the middleware requirements with a use case of the 'De Lijn' advertisement application. Figure 3 gives a simplified representation of the component based application composition at two moments in time. The left part of Figure 3 shows the advertisement application running on two identical busses and on the back-end server. The right part of the figure shows how the execution environment may become over time. The advertisement display of one bus (a) is replaced by a large one, and a new model bus (b) is added with future system software (Java Standard Edition 8) and two large advertisement displays. To support this evolution, the middleware platform must be able to:

- Detect the replacement of the advertisement display on the second bus

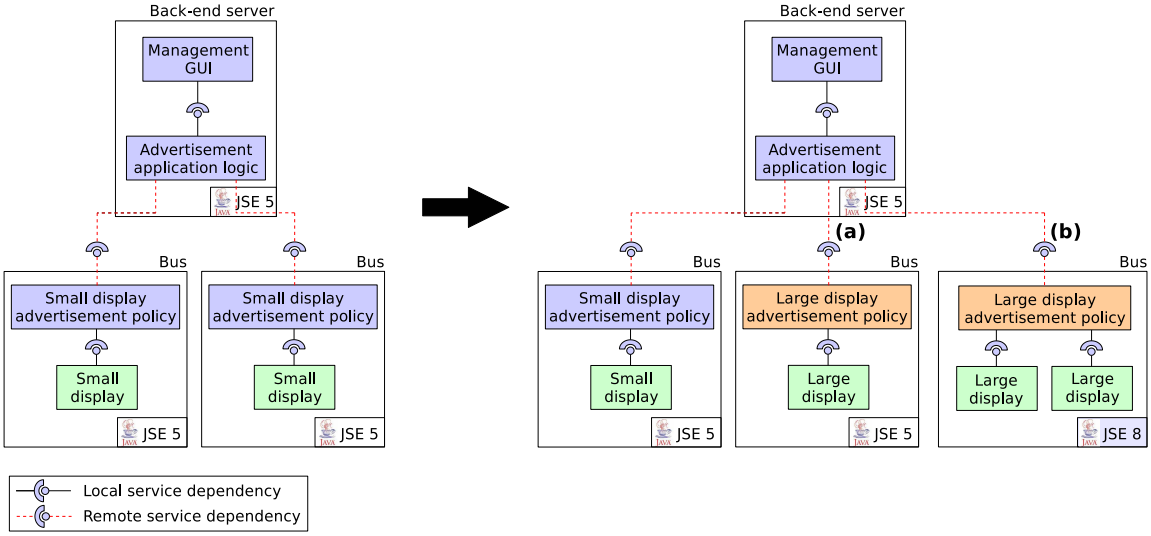

Figure 3: Advertisement application in distributed evolving environment

by a new model with different possibilities: **Environment evolution, determining local environment properties**

- Select the application software best fitted for the large advertisement display. Automatically deploy the selected application software on the second bus. Rewire the application software to replace the old 'Small display advertisement policy'-component by the new 'Large display advertisement policy'-component: **Local automatic application composition, Automatic software deployment**
- Detect the adding of a new bus to the execution environment: **Environment evolution, group discovery**
- Detect the difference in system software (JVM) and peripheral devices between the new bus and the other busses: **Determining local environment properties**
- Select the best fitted application software for this bus, install it on the bus and wire it in the distributed application: **Local automatic application composition, automatic software deployment**

Further, there are two additional requirements:

- The application software on the back-end server requires a 'busAdvertisementService' from the buses (not represented in figure). The middleware platform must be able to resolve remote dependencies. **Remote dependencies**
- The remote dependencies (between software components on back-end server and bus) must also be resolved during the automatic wiring of distributed applications: **Distributed automatic application composition**

3 OSGi service platform

The OSGi [4] service platform has support for component based architectures, service dependencies and application life cycle management. In this section, we describe how the OSGi service platform can be used to fulfill a subset of the middleware requirements identified in section 2. First, we demonstrate how local environment properties can be determined with OSGi. Second, we show how OSGi can be used to compose an application composition, adapted to the local execution environment.

3.1 Determining local environment properties

The OSGi service platform is able to determine different kinds of environment properties. It identifies the underlying JVM and checks the JVM requirements of each application component before deployment. Further, OSGi can be used to detect the presence of peripheral devices (displays, sensors, actuators, ...) and to identify their offered functionality by using peripheral devices which are able to register themselves with the OSGi platform as services [9, 13].

3.2 Local automatic application composition

The middleware platform must be able to automatically compose new application compositions, adapted to the local environment. To make this possible, we equip each application with an **application-root component**. This component describes which critical functionality must be offered by the application during its complete lifespan. It distinguishes itself from the other components in two ways. First, this is the only component of the application for which we don't allow the middleware to replace it automatically. We made this choice to guarantee that the critical functionality of the application is maintained. The second difference is that this is the only application component which does not have to offer any services to other application components. This is because we start the application composition from this component. The middleware platform automatically composes the application by resolving the service requirements of the application-root component with application components best fitted for the local environment. The root component in the 'De Lijn' advertisement application (Figure 3), is the 'Management GUI'-component.

When the local environment properties are determined, the middleware has to adapt the application composition accordingly. First, the middleware has to determine the best fitted application composition for this environment by resolving the service requirements of the application-root component. If not locally available yet, the middleware has to download and install the application components of the selected application composition. Once all needed application components are installed on the local platform, the middleware needs to wire them in the distributed application.

We are currently working on an implementation of our middleware platform. In this implementation, we are using the **OSGi Bundle Repository Service** or OBR [5] for the selection of the best fitted application composition, the downloading and the installation of the needed application components. For the automatic application composition on each device, we use the OSGi **Declarative Service** (DS) service [2].

4 dEVOLVe

Figure 4 presents an overview of the middleware requirements identified in section 2. For each requirement, the middleware components used to meet the requirement are marked with a rectangle. As mentioned in the previous section, a large part of the required functionality is covered by existing OSGi technology. However, middleware support is missing for: remote dependencies, group discovery, distributed automatic application composition, and environment evolution.

	OSGI	DS	OBR	dEVOLVe
Determining local environment properties	■			
Local automatic application composition	■	■	■	
Automatic software deployment			■	
Remote dependencies				■
Group discovery				■
Distributed automatic application composition				■
Environment evolution				■

Figure 4: overview of the functionality offered by the middleware platform components

We are currently working on a OSGi component, called dEVOLVe, which contains the missing middleware support. In this section, we describe how dEVOLVe realizes the needed functionality.

Figure 5 gives an overview of our middleware platform for software maintenance in distributed evolving environments. Both the dEVOLVe, OBR, DS and application components run on top of the OSGi base framework. OSGi allows direct access to both the JVM and the OS.

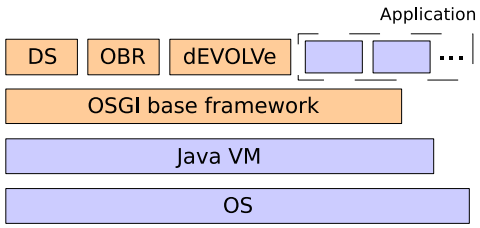

Figure 5: Middleware platform for software maintenance in distributed evolving environments

4.1 Remote dependencies

The OSGi service platform has support for the description and wiring of service, package and bundle[1] dependencies within a single VM. However, dependencies with remote devices are not supported. Therefore, we added a remote service, package and bundle dependency header (Figure 6) to the component manifest

[1]Components are called bundles in the OSGi community.

file as remote variants on the existing OSGi dependency headers. The 'group' argument of those manifest headers will be discussed in the next subsection.

```
Remote-Service:ServiceName; group="roleName(s)"
Remote-Package:PackageName; group="roleName(s)"
Remote-Bundle:BundleName; group="roleName(s)"
```

Figure 6: Remote dependency manifest headers

The dEVOLVe component parses the manifest files of the used application components and tries to resolve their remote dependencies. OBR [5] is used to deploy the needed application components on the remote devices.

4.2 Group discovery

The devices in an evolving environment will change over time. Because of this, application developers can not describe the devices themselves in the remote dependencies. To deal with this problem, we introduce weak couplings with remote devices by adopting the **role** concept from the sensor network community [11]. The developers can describe dependencies with groups of remote devices fulfilling certain roles. They do not need to know which devices will fulfill these roles, and devices can be added and removed at runtime to each role. The assignment of roles to the devices is considered a task of the local network manager who may choose to automate this task with a role assignment algorithm [7].

4.3 Distributed automatic application composition

Automatic wiring is not possible with remote package and bundle dependencies. Remote service dependencies can be wired automatically with DS when the remote services are registered as local services. To realize this, dEVOLVe generates a proxy object for each remote service and registers it as a local service. This is similar to the approach of R-OSGi [12] to make remote services accessible. A simple remote method invocation mechanism is used to handle the remote service calls.

4.4 Environment evolution

The middleware platform must be able to deal with changes in the execution environment. A very robust, but not necessary the best, technique to deal with changes in the execution environment is a periodical recomposition of the complete distributed application. We realize this with an **application heartbeat**: a periodic signal which starts from the application-root component and travels through the complete distributed application by following the service bindings between the application components. On each device where this heartbeat passes by, we react on this signal by checking if the current application composition is still the optimal one to satisfy the service requirements. When a better fitted application composition is discovered, the application will be adapted.

Some installed application components may no longer be used after an application recomposition. For instance, because a new component has been installed

to deliver the required service, or because a broken network connection is making it impossible to deliver the remote service. In case of local service dependencies, DS will be able to react by stopping the currently unused components. However, this is not possible for the remote service dependencies. The heartbeat signal, which travels periodically through the complete distributed application, allows us to identify the components which are no longer part of the current application composition. Those components will no longer receive the heartbeat signal and can be removed or deactivated since they are no longer useful in the application.

5 Related work

Existing, state-of-the-art middlewares are able to adapt software towards its changing distributed environment. However, they all have limitations, making them less fitted for software maintenance in distributed evolving environments.

5.1 Impala

Impala [10] is a middleware platform for distributed sensor networks, able to adapt the distributed application on-the-fly towards the evolving execution environment. Impala determines on each device some properties of the local execution environment such as the battery level and number of direct neighbors. The user is able to describe switching rules which determine how the application has to be adapted when the environment properties change. Impala is also able to check the peripheral requirements of the application software. This avoids the execution of application software with unmet hardware dependencies.

However, Impala only supports monolithic applications. This makes it impossible to adapt parts of the application. Further, Impala does not support the assignment of roles to devices. Therefore, only distributed applications which require the execution of the same task on each device are supported. Impala also is not able to check remote dependencies. Because of this, all application software on the network must offer and require the same remote services.

5.2 TinyCubus

The TinyCubus [11] middleware supports component based application architectures with service dependencies between the application components. It is able to detect changes in the environment, to selects the best fitted application components and to wire them together. The selection of the application components is done based on 1) the device environment (e.g. network density), 2) the application requirements (e.g. reliability requirements), and 3) the optimization parameters (e.g. minimal resource usage).

However, TinyCubus does not allow to describe remote dependencies. This makes it difficult to guarantee cooperation between the application software running on the different devices. Further, TinyCubus is only able to select components installed on the device. The automatic deployment of components on the devices is currently an un-solved problem. TinyCubus also is not able to rewire service dependencies at-runtime.

5.3 Gridkit

Gridkit [8] is a middleware framework able to adapt itself to changes in the environment. A component based architecture with service bindings is used to allow dynamic adaptation. In contrast to the previously discussed middlewares, Gridkit uses a centralized approach. A configurator collects information about the composition of the complete distributed framework and receives events about changing environmental conditions. With this information, the configurator determines when and how to change the framework by inserting, deleting, disconnection, connecting and replacing middleware components on the different devices. However, although Gridkit is able to adapt the middleware support towards the applications, it is not able to adapt the distributed applications themselves.

6 Conclusion and future work

In this paper, we have identified the middleware requirements for software maintenance in distributed evolving environments. We illustrated this with a use case of an advertisement application for public transportation. We have demonstrated how existing OSGi technology can be used to realize an important part of the middleware requirements. We have presented an OSGi based middleware platform, called dEVOLVe, which adds the missing middleware support for: **Remote dependencies:** Remote service, package and bundle dependencies can be described and are automatically resolved.
Group discovery: Application developers can describe remote dependencies with an unknown group of devices fulfilling a certain role.
Distributed automatic application composition: Remote services are made available as local services to allow automatic wiring of service dependencies.
Environment evolution: Changes in the distributed execution environment are detected by the middleware and the application is adapted accordingly.

Application developers who are familiar with OSGi and DS can start writing applications for dEVOLVe without much additional effort. They only have to learn to use three new manifest headers for the description of remote dependencies. dEVOLVe does not require the implementation of additional methods, or the inheritance of dEVOLVe classes. Even existing OSGi components can be used without any adaptation.

dEVOLVe is work in progress. In the near future, we want to add some optimizations to allow dEVOLVe to react faster to changes in the environment. We also are going to evaluate the proposed solutions and enhance them where possible. Further, we want to investigate how dEVOLVe can be extended to support life cycle management (including software evolution) in distributed evolving environments. Our current middleware platform already contains some life cycle management support; the OSGi service platform has support for software life cycle management on a single device, and dEVOLVe extends this life cycle management support to distributed applications. However, we believe that the current life cycle management support for distributed applications is too difficult to use. Research of additional middleware support is required.

Acknowledgements

The authors like to thank Sam Michiels for valuable comments and stimulating discussions. This work is part of the SPAMM project, funded by the IBBT (Interdisciplinary institute for BroadBand Technology).

References

[1] De lijn.

[2] Declarative services, osgi service platform service compendium, release 4, osgi alliance.

[3] The ibbt-spamm project: Solutions platform for advanced mobile mesh.

[4] Open services gateway initiative.

[5] Osgi alliance rfc-0112, osgi bundle repository, http://www2.osgi.org/div/rfc-0112_bundlerepository.pdf.

[6] Guy Bieber and Jeff Carpenter. Introduction to service-oriented programming, 2002.

[7] Christian Frank and Kay Römer. Algorithms for generic role assignment in wireless sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 230–242, New York, NY, USA, 2005. ACM Press.

[8] Paul Grace, Geoff Coulson, Gordon Blair, Barry Porter, and Danny Hughes. Dynamic reconfiguration in sensor middleware. In *MidSens '06: Proceedings of the international workshop on Middleware for sensor networks*, pages 1–6, New York, NY, USA, 2006. ACM Press.

[9] Jeffrey King, Raja Bose, Hen-I Yang, Steven Pickles, and Abdelsalam Helal. Atlas: A service-oriented sensor platform. In *SenseApp '06: Proceedings of the first IEEE International Workshop on Practical Issues in Building Sensor Network Applications*, November 2006.

[10] Ting Liu and Margaret Martonosi. Impala: a middleware system for managing autonomic, parallel sensor systems. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 107–118, New York, NY, USA, 2003. ACM Press.

[11] Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Jörg Hähner, Robert Sauter, and Kurt Rothermel. TinyCubus: A flexible and adaptive framework for sensor networks. In *EWSN '05: Proceedings of the Second European Workshop on Wireless Sensor Networks*, pages 278–289, January 2005.

[12] Jan S. Rellermeyer and Gustavo Alonso. Services everywhere: Osgi in distributed environments. In *EclipseCon '07*, 2007.

[13] J. Russo, A. Helal, J. King, and R. Bose. Self-describing sensor networks using a surrogate architecture. Technical report, University of Florida, June 2005.