# A Deployment System for Pervasive Computing

Jesper Andersson
Department of Computer and Information Science
Linköpings universitet
jesan@ida.liu.se

## Abstract

*Software has for a long time been used for controlling different systems. Today, there is a trend towards integrating more software in consumer electronics, home appliances, cars etc. Suddenly, software moves from traditional environments, such as the desktop computer into new and unknown territories. This will influence many aspects of the software engineering process, above all several new problems arise in the domain of software deployment. How can software be effectively deployed in these environments? Current deployment strategies are not directly applicable. This paper discusses problems with current deployment models and proposes a new, modified model. A deployment model for pervasive computing, must support component based development, different delivery models, and installation and activation strategies. Especially, support for dynamic installation and activation, i.e. modifications without de-activating the software, is important. We introduce a prototype implementation, the Java Distributed Run-time Updating Management System (*JDRUMS*), which we have used as a vehicle for eliciting requirements for our deployment model.*

## 1 Introduction

The last few years, mass-produced and cheap, single-chip computers have been included in different "devices", such as consumer electronics, home appliances, and vehicles. Implementing core functionality in software instead of hardware is a more flexible and cost-effective approach. New developments in communication technology, especially in the area of mobile and short-range digital communication will be available on the market within a year or two. This technology, integrated in different "devices", opens up for a new software market. Several companies claim that network connectivity and new types of application software will add significant value to their products, something that will be decisive on the market, where every competitor struggles with hardening competition and reduced margins.

Pervasive computing, the latest "buzz-technology", focus on the possibilities and problems with this new computing model."The intelligent household", with smart refrigerators, and "personal assistants" that combines mobile phone technology and palm-top computing, are some examples of applications.

This new computing model, will influence the software life cycle and several problems were current software engineering practice is not directly applicable, can be identified.

One of the most radical changes, compared to a traditional desktop systems, is how the devices' software is deployed, i.e. packaged, delivered, installed, configured, and activated. Devices have no dedicated administrative user, i.e. a user that is responsible for parts of a deployment activity. Most devices, will not even have a suitable interface were users can perform these tasks.

Our research focus on software deployment in "a world of devices". We see that without an administrative user, many of the existing deployment models are disqualified. Most end-users will not be interested in installing a new piece of software for their refrigerator, especially not if the refrigerator is only one out of hundreds of devices in the household. Then, it doesn't matter if the software is distributed via surface mail or downloaded from the Internet. The other actor in this play, the manufacturer, is on the other hand not very keen on having non-professionals fiddling with their appliances. Here, the alternative is to send qualified personnel to the end-user or call in the devices to a service workshop, when a change to the

software is necessary. The number of devices causes a logistic problem, i.e. transportation of people or transportation and storage of devices. So, neither of these approaches is really cost-effective.

Several of these devices will be mobile and move around in the world. Since, a device has certain resource limitations, such as the storage size, new strategies were functionality is downloaded on demand will emerge. This is similar to, but more automated than the "plug-in" approach used in many web-browsers. When the device need some specific functionality, it downloads, installs and activates it. In a mobile scenario, this type of updates will probably include some de-installation of some other functionality, due to the lack of resources. Current deployment strategies do not work well with this type of systems either.

To cope with this new situation we need another deployment model. A model that transfers the responsibility of doing administrative tasks from "end-users" to either the devices or device suppliers, and that utilize from network connectivity, the flexibility of independently deployed components, and dynamic updating systems.

In this paper we outline a deployment model that we believe is better dressed for this new computing model. Here the administrative responsibility can me transfered to either the devices or device manufacturers. We present a prototype implementation that supports our model, the Java Distributed Run-time Updating Management System (JDRUMS). This Java based system initially focused on dynamic updating of component based software, but additional subsystems that handles other aspects of our model have been introduced.

The remainder of this paper is organized as follows. Section 2 discusses existing delivery models and why these do not work well in the context of pervasive computing. Continuing in Section 3, we discuss the requirements for a pervasive computing deployment model. In Section 4, we present our prototype environment JDRUMS. Related work is surveyed in Section 5. Finally, in Section 6 we evaluate our model with respect to the requirements and other deployment systems, we conclude and discuss future work.

## 2 Software Deployment

In the last few years, software has become an integral part of many devices. This is an accelerating trend and today, it is difficult to imagine all future systems that will use software for adding value to the product. Increased use of software in different devices, gives raise to several questions; one is how all this software should be administrated so updated versions and new functionality that devices need is made available to the device. This will affect software deployment, i.e. how software is packaged, delivered, installed, and activated on devices.

Existing models for content delivery, discussed more in depth below, does not work well with this new type of systems.

1. A majority of the devices have no user-interface, from which a user can perform these administrative tasks.
2. Even if there is a suitable interface, it is impossible to require that the owner should take on the administrative responsibility for all these, potentially hundreds of devices.
3. Another side of the problem is that the application software in the devices will consist of several composed components. These components will be collected from different component manufacturers and integrated by the device manufacturer.
4. This new type of applications also requires a new system for managing upgrades, which is more flexible. It must have tha capability to work with single component level as well as i can handle traditional complex composites.

If we look at current practice for software deployment we see three widely used approaches by commercial systems. One approach used for private desktop machines and most computers in an office environment. Other semi-automated approaches, where servers install "images" on machines, often used in office environments. And third, approaches used for administration of distributed systems.

### 2.1 Current Deployment Models

The most widely used deployment model, is when the software is transported to the user, either over a network or on some media (e.g. CD-ROM). The user installs the software on a secondary storage and configures it, after this the software is ready for execution. In this model we can vary several variables, for instance, it could be a system administrator instead of an ordinary user, and the software could be installed on a server, but still, the deployment model is heavily dependent on the involvement of people in order to work.

In some companies, an automated approach to deployment is being used. Every machine in the company network has a "software profile", stored locally or on a central server. This profile is used to create machine specific software images that are downloaded and installed on a machine, for practical reasons most often in non office hours. This approach is an improvement compared to the approach presented above as the current software configuration is available and much of the work can be automated. But, the approach is to coarse-grained to be directly applicable in a pervasive computing environment and it also needs to be more flexible in terms of size.

In a distributed system environment, the remote administration approach is widely accepted. Here, a central command center supervises systems, and sees to that a correct configuration is running on all nodes. Tools are used for content delivery and installation of new packages. Some distributed infrastructures also provide for dynamic updating of components. This approach is also a step in the right direction, but it needs some polishing to better fit pervasive computing.

**Recent developments** As previously mentioned, several deployment systems are available. The increased usage of the Internet has influenced these products. For instance, "Installation wizards" the de-facto standard on Windows-based systems does not contain all the content anymore. Instead it works as a "content deliver proxy" that downloads the content needed from the "nearest" server on the Internet, based on the component requests specified by the user in the wizard..

In the Linux world, automated routines for download and installation of software are used in several distributions. These schemas are based on complete descriptions of the software that is currently installed on the system.

A survey of different approaches to software deployment and available systems for different deployment activities is presented by Hall [7].

## 3 Requirements

A deployment model for pervasive computing has to satisfy several requirements. In the discussion below we have chosen to divide the requirements into three groups.

- Component Based Systems. This implementation technique has grown in popularity the last few years, and we believe that components and component infrastructures will be a fundamental component in pervasive computing.
- Flexible Content Delivery. The variety of systems in a pervasive system, require that a deployment model supports different ways of delivering software. In pervasive computing several actors can take part in the delivery activity, users, devices, and manufacturers, which all require different support in terms of content delivery.
- Flexible Installation and Activation. Traditional installation where software and systems are de-activated before installation of new content, is inappropriate in many situations. Some devices have a high-availability requirement, which means that we must minimize downtimes, other are mobile and work autonomously. For this type of systems, dynamic installation and activation will be a fundamental requirement.

### 3.1 Component Based Systems

Component based system development is the most recent star on "the software engineering canopy of heaven". Components have been on the agenda since McIlroys paper from 1968 [10]. But owing to recent advances in component technology, such as CORBA, DCOM, and Java Beans, components have started to gain credence again.

Szyperski [12], foresees a market for software components that are self-contained, binary and independently deployable. A device manufacturer that uses a component strategy when developing their applications, can choose to develop all components in house, but more likely they will combine product specific components developed in house with procured components.

This new development strategy will have a great impact on how software extensions and upgrades should be managed. The problems related to component based applications differ depending upon if you take the perspective of a device manufacturer, a component developer or a device user.

From a device manufacturer perspective, software systems will be composed of components from multiple component manufacturer. New releases of these components will appear from time to time. Some of these releases are regarded as more critical, which means that a manufacturer probably would like to see these components deployed as soon as possible. Other components that are less critical can wait and be part of a major upgrade package, which is assembled later. So the optimum will be a system where individual components can be identified and deployed. Very few deployment systems (or run-time systems), support this level of granularity. Typically, a manufacturer needs to deploy superfluous content, in order

to deliver a single, newly released, component. For instance, if the bandwidth is limited big chunks of data that effects other communication is not desirable.

The current software configuration resident in the device must be accessible, so current versions can be compared to available versions when packaging releases. Some deployment systems supports this, others attempts to detect the configuration as a pre-installation activity.

How components are assembled and deployed should be transparent to developers. This will increase the number of candidate components to use. A deployment system should have minimal influence on the implementation of components.

The majority of device users have no interest in which software that is currently running in their devices and definitely not if it is delivered as components or some other type of software package. So, component wrapping should be kept transparent for users and developers. The wrapping of components, is not transparent to the user in many deployment system, for instance some systems require certain functionality that unpacks compressed files.

- Components packaged as “deployment-kits”, i.e. hiding how components are packaged and what it is that makes it "deployable". This will make the deployment activity transparent to both device users and component developers, i.e. deployment is principally a responsibility for device suppliers.
- Manufacturers need support for packaging content spanning the range from single components to complex composites containing components from different component producers.
- A first-class representation of the current software configuration available in a device. This is used in the decision process, when components are packaged.

## 3.2 Flexible Content Delivery

Aside the problems related to components, the number of devices that a deployment system will serve, is another problem dimension. Potentially hundreds of thousands of devices could have been sold on the market and the software running in these needs to be attended to. Different devices will require different delivery models. A household will have devices from several manufacturers, some devices will be mobile, some users maintains their own devices etc. A manufacturer require that the deployment system provide sufficient support for handling all these situations.

In order to fully support this diverse set, we formalize three delivery models that should be supported. Below, we describe two “pull” models were a user or a device initiates and controls the deployment. Then we describe a model based on a “push” strategy. Here the device supplier is responsible for the deployment.

Existing deployment systems, typically supports one or two of these models (or something similar). But we argue that all are equally important in an for pervasive computing environment.

### 3.2.1 Two “Pull” models

The “pull” strategies are based on a scheme where deployment of software content is initiated at the “user-side”. Packages are downloaded directly from application servers and installed on the device.

In the first “pull”-delivery model, a user initiates and control upgrades from a terminal, typically a personal computer. The model, depicted in Figure 1, supports the more advanced users whom wants to have control over the software that is downloaded and installed on their devices. In this model, the user needs some application that can be used to survey which devices that are on-line, query for deployment-kits available for these devices, and capable of delivering these to the devices. The application must be capable of handling devices from different suppliers, compare current software configuration with versions available on different application servers. This delivery model is capable of handling several devices and components from different manufacturers. Another advantage is that every device is treaded uniformly. The drawbacks include, detailed standardization of device interfaces and application servers, so applications can interact with devices from different manufacturers and different application servers.

- A possibility to locate and establishing communication with devices that are online in a specified domain.
- An administrative tool will interact with servers were deployment kits are stored.
- Automation of the deployment is important, i.e. hide details and minimize the risks of wrongdoing.

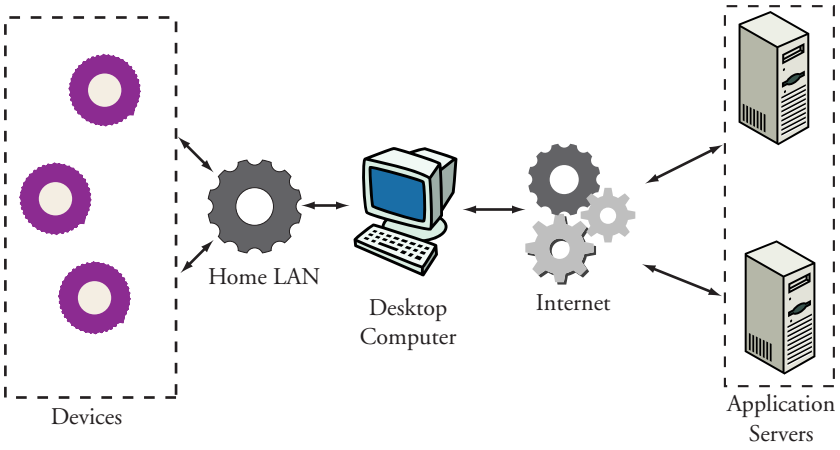


**Figure 1. Upgrades managed from a desktop computer**

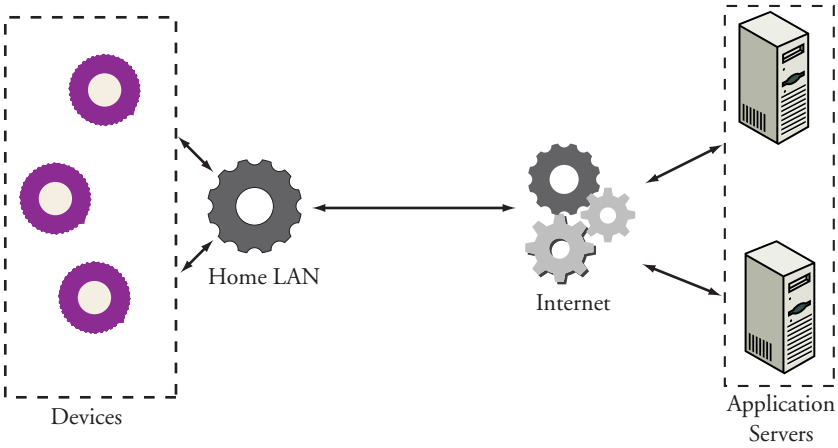


**Figure 2. Devices initiate and manage upgrades**

Our second model, based on a “pull” strategy, is outlined in Figure 2. Here, the functionality for delivery, installation, and activation is implemented in the devices, hence it is fully automated. Every device is responsible for checking at different application servers for available upgrades, download the adequate packages and install and activate these. In this model the devices will communicate directly with applications servers. This delivery model is really useful if the device need some form of “functionality on demand”. For instance, a mobile device with a positioning system. When the device move around in the environment it can add and remove functionality depending upon where it is. When the device enters the “automated bank” the device downloads the banking components, installs, and executes the application. Before that it must free resources, so it “de-installs” some components that is not directly needed in this environment, for instance the GPS-map software.

This model require, compared to the previous model, limited standardization, which means that device manufacturers can develop optimized implementations for their devices. The disadvantages include, “extra“ functionality in the devices, which can be really problematic since many devices have limited resources.

- Devices should be capable of carrying out the deployment.

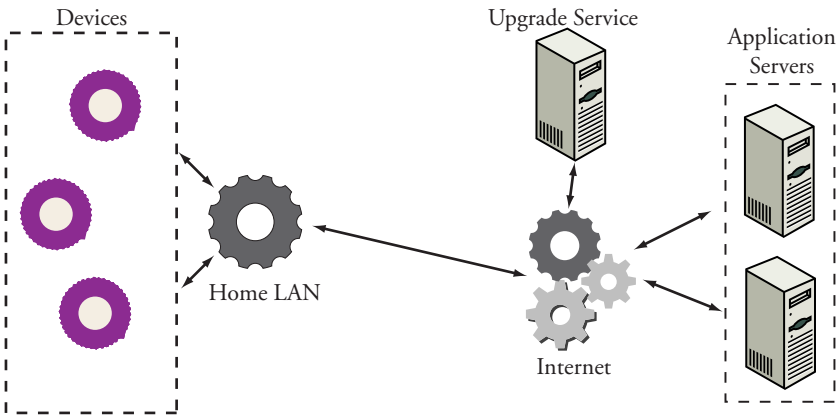


**Figure 3. Update service manage upgrades**

### 3.2.2 A "Push" model

The "push" strategy is based on a scheme where deployment is initiated at the "supplier-side". The manufacturer keep a record of devices and "push" deployment packages from application servers to devices. This strategy is similar to the approach often used for deploying software in a distributed system.

In Figure 3, we present a schematic view of a "push" systems. Typically, there is a dedicated "upgrade service" or deployment application, which automates the deployment. It is possible to move the administrative tool from the first pull model to the manufacturer site, but with many devices, a non-automated approach is not feasible. The deployment application keeps a record of the devices, with an address and the current software configuration. When an a new package is made available, it deploys this to all devices. This model is probably the most preferred, from a supplier perspective. Manufacturers will have full control. A deployment model supporting this delivery strategy must meet a certain set of requirements.

- The delivery system must keep track of devices and their current configuration.
- Device suppliers should have the possibility to communicate, directly or indirectly, with a large amount of devices and component servers, when pursuing deployment of new software.

## 3.3 Flexible Installation and Initiation

The delivery models discussed above, all depend on devices providing support for installation and activation. There are two strategies for this activity. The first one is similar to the one used on desktop computers, the software is de-activated, a new version installed, and finally the new software instance is re-activated. The second strategy, dynamic updating, is used in many mission critical applications, where high availability figures are required. For instance, many command and control applications and communication systems utilize from dynamic updating. Dynamic updating is discussed more in depth in Section 5. If we look for dynamic updating support in existing deployment systems it is difficult to find any system that offers this functionality. Some systems do, but most often these are domain specific systems that are difficult to use in other domains.

A deployment model for pervasive computing should support both these strategies. The "dynamic deployment strategy" will be useful in different applications, especially when deployment is controlled at the supplier-side. Then it is difficult (impossible) to decide whether some piece of software can be closed down or not, without interfering with the usage of the device. Another situation where dynamic updating is preferable is when software is deployed on a context dependent basis. When a device moves around in the world, changes to the software configuration should be fully transparent to a user. Finally, some devices, will be devices with a high availability requirement, e.g. fire- and intruder-alarms where dynamic updating is an absolute requirement. We present a more detailed list of requirements below.

- Activation, De-activation, and re-activation of applications and components.
- Installation and de-installation of components and applications.

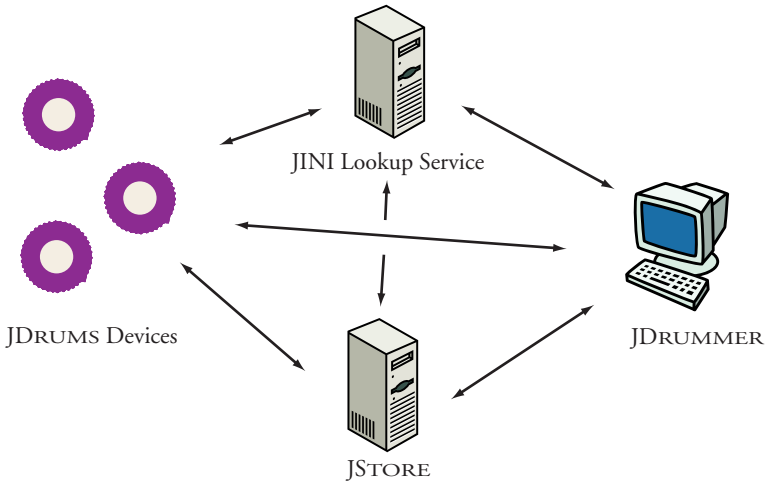


**Figure 4. The JDRUMS System Architecture**

- Replacement of running component instances with new versions, with the application state preserved.
- See to that dynamic deployments are made persistent. Dynamically deployed software should be used the next time an application is activated.

## 4 JDrums

The Java Distributed Run-time Updating Management System — JDRUMS [3], is a prototype implementation of a flexible framework for software deployment for pervasive computing. Previously, we discussed several requirements for such a deployment system. Initially, in this prototype, we have focused on basic support for component-based systems, different delivery models, and dynamic updating.

The prototype is based on and restricted to Java. One important factor for choosing Java as the implementation platform, is the JINI technology [4]. Directed towards small, embedded systems, JINI simplifies many tasks, for instance communication between different system components and registration of devices. Another reason for choosing Java was the strong component model Java Beans.

### 4.1 System Architecture

The JDRUMS system architecture consists of four components, depicted in Figure 4 and described below. First, there is the JDRUMS devices, were user applications are executed. In our current implementation a device is represented by a modified Java Virtual Machine (JVM). Modifications include support for dynamic updating and network awareness. Second, the JDRUMMER tool, that functions as the command post for deployment activities. Third, there are component servers (JSTORES) . These servers works as component warehouses and JDRUMS devices and JDRUMMER tools interacts with these in when carrying out deployment.

The communication between the components is JINI based. All components register as services in a JINI lookup Service. When a component require a service, it queries the lookup service, then it can connect to any of the services available using RMI [8]. JINI offers a clean Java solution and has built in security policies, which is advantageous in a network environment.

#### 4.1.1 The JDRUMS - JVM

In order to fully prototype our deployment model, we needed something that simulates the “devices”. In a Java environment, the choice of a JVM as the basis for the simulation of devices comes natural. The JDRUMS-JVM is a modified virtual machine with extra functionality, providing for dynamic updating of classes and registration of the JVM as a JINI service.

Dynamic upgrades works on a class[1] level [2]. When a "deployment-kit" is delivered to the JVM, it is unwrapped and installed. Later, when objects that are instances of an obsolete class version are referenced, an instance of the new class version is created, the object state is migrated from the "old" object to the "new" object, then the reference that "triggered" the migration is changed to reference the newly created object. The object migration is controlled by a conversion class, which is packaged together with new class release.

The contents of a deployment package varies. For small upgrades, the new version of a class and the conversion class sufficient. But, more complex upgrades, which includes more than one class, or when classes are extended, require more classes to be included in the package.

The JVM also keeps version information for all installed classes. This information is used when deciding the contents of the deployment packages that should be delivered.

#### 4.1.2 The JSTORE

The JSTORE component has one main responsibility, that is storing several releases of different components. It also keeps version and dependency information for all releases. The JSTORE can use information from the JVM or directives from the JDRUMMER, when creating deployment packages.

#### 4.1.3 The JDRUMMER tool

To assist system maintainers with deployment, the JDRUMS environment includes a tool, the JDRUMMER. It is a GUI based tool, in which you can easily combine one or more deployment packages from different JSTOREs and deliver these to a specified set of devices. The tool, also include lists of running JDRUMS-JVMs' and JSTOREs in the neighborhood. Using these lists, a administrator can query running JVMs for their current class configuration, find out if new releases are available at the component stores, and select JVMs as targets for deployment.

### 4.2 A deployment scenario

In the environment there is at least one JSTORE, component storage running. All registered in the JINI lookup service. When, a device is started it also registers in the JINI lookup service. The device can be given a startup directive to download the latest available software directly from a storage or use the software that is already available on the device. When a system administrator thinks it is time to administrate the running devices, he starts up the JDRUMMER tool. The JDRUMMER looks up running devices and JSTOREs and displays this information. Now, the maintainer begins with assembling deployment packages and deploy these to target devices, in the process he queries devices for the current software configuration and the component stores for available packages. As deployment packages arrive in the devices, these are automatically installed. Eventually, all objects in the running application are migrated to the new class representation. The work is done and the administrator exits the JDRUMMER tool.

## 5 Related work

### 5.1 Deployment systems

As we perviously mentioned there are several commercial deployment systems available on the market. Spanning the range of simple systems , delivering software from one site to another, to complex systems for enterprise wide installations and updates. Many of these systems work well when complete applications with very few inter-application dependencies are deployed. But suffers from severe problems in a pervasive computing environment.

The Software Dock [7] is research project, where a framework for software configuration and deployment being developed. The deployment framework uses a client-server architecture in combination with a event system. At the manufacturer- or supplier-site, there is a server (release dock) with release information. On every user-site, a client (field dock) works as an interface towards release dock. There is a registry containing all information on releases available at release docks and configurations in field docks. The Software Dock uses a event system for signaling changes in the registry and agents for controlling certain deployment tasks such as delivery, installation, and activation.

[1] The terms component and class are interchangeable in this description.

### 5.2 Dynamic updating systems

Dynamic deployment systems are by no means a new field of research, especially if we go into the domain of distributed systems and large mission-critical software packages. Several strategies for dynamic updating, both hardware and software based, have been proposed, although none have revolutionized dynamic updating by offering a solution cheap or practical enough for anyone to use. They all have some shortcomings. The special case with object-oriented systems, have no no support in existing approaches. More details and in depth descriptions of different types of can be found in the works of Segal and Frieder [11] and Gupta [6].

There are hardware-based solutions that allow dynamic reconfiguration of applications by replacing the running system with an updated one. Here, a secondary system is used where the new release is activated, eventually the system switches to the secondary system and de-activates the "old" release. Such a system is often operating in conjunction with a hardware-based system for fault-tolerance, but it is still difficult and expensive [11] to provide the functionality for migrating state between machines. The cost of duplication hardware in a device disqualifies this approach for pervasive computing.

The software-based approaches presupposes that programs consists of a number of independent building-blocks, or modules. To reconfigure the system, no modifications are necessary to the individual modules, since the connections between them are handled by the manager rather than hard-coded references. The main drawback is that special knowledge is demanded from both programmers and system administrators. This violates our transparency requirement. Other variants of a module based approach uses procedures and ADTs as modules, e.g. the dynamic type replacement system proposed by Fabry [5].

Other module-based approaches are based on the client-server architecture, often used in distributed systems. The client-server model works much in the same way as that for abstract data-types, only in a bigger scale. The granularity of components is increased from data-types to server-modules. The interfaces to the server never change, only the implementation of the different services it provides. A general drawback using systems, which follow this approach, is the size of the components being replaced. Even a small change will lead to an entire server being unavailable for some time, for the duration of the update. Darwin [9] is one example of a configuration language for distributed systems. Darwin provides support for dynamic reconfiguration at run-time in the form of a reconfiguration manager. A supervisor can send directives in a script language to the system that invokes dynamic reconfiguration. In a Darwin implementation, every element can be added, removed, or replaced dynamically.

## 6 Conclusions and Future work

We have proposed a flexible deployment model for pervasive computing environments. Furthermore, we have described the JDRUMS, a Java based deployment system that provides support for component based applications, flexible delivery, and dynamic updating. Below we evaluate our model and prototype with respect to the requirements listed in Section 2 and other approaches to software deployment.

With our model, component designers do not have to consider how a component is deployed. The deployment system handles everything. This transparency is favorable, since it will make components less product specific and widen the potential market. There is one exception though, when a component is targeted for dynamic updating, there is a need for additional functionality describing how an object should be migrated to the new class representation. Here, we see a clear violation of our transparency requirement. But, on the other hand, the component developer can probably develop a conversion class along with the new version without straining the project budget. For some components, the conversion class could be generated automatically using some reflection technique. Components are packaged into upgrade-kits that contain one or several components and other required parts. The JDRUMS-JVM keeps information on the current component configuration, the device or some other component in the system have access to this.

Our model supports three delivery models. But here more research and hands-on work is needed to fully implement all models in the prototype system. Our prototype relies on JINI, and devices and other components are registered in a lookup-service. Here, devices can lookup component storages, deployment tools can find devices etc.

The prototype supports dynamic updating of Java components, which has been our main objective. More work is needed here though, such as improved support for complex upgrades, performance optimizations, and introducing a true "first-class" representation of the software architecture configuration.

Compared to other deployment system, such as the Software Dock and other, commercial systems, our work has a different focus. Our system is intended for fine-grained deployment, at the component level. We emphasis dynamic updating, where new functionality and updates can be integrated into an active application transparently.

Future work in this area will focus on context and quality-criterion based configuration. We will look into problems with how to model, implement, and maintain applications were the architecture changes. These changes are due to certain events, a more detailed description is provided by Andersson [1]. We will also look into how to provide support for "large-scale" deployment. Here, our work will be influenced by others, for example the Software Dock project.

To summarize, we believe that our model and prototype meets most the requirements for deployment in a pervasive computing environment. It is flexible in the sense that we can easily adapt to new or modified requirements. A flexible deployment model that supports component based applications, different delivery models, and is flexible in terms of installation and activation is a key success factor for pervasive computing.

## References


[1] J. Andersson. Towards Reactive Software Architectures. Technical report, Linköpings universitet, May 1999. Licentiate Thesis. In Linköping Studies in Science and Technology, No. 769.

[2] J. Andersson, M. Comstedt, and T. Ritzau. Run-time Support for Dynamic Java Architectures. In *Proceedings of the ECOOP'98 workshop on Object-Oriented Software Architectures*, 1998. Technical report 13/98 University of Karlskrona/Ronneby.

[3] P. Danielsson and T. Hultén. Java Distributed Run-time Updating Management System. Master's thesis, Växjö universitet, 2000. Under preparation.

[4] W. Edwards. *Core JINI*. Prentice Hall PTR, 1999.

[5] R. Fabry. How to Design A System in Which Modules can be Changed on the Fly. In *Proceedings of International Conference on Software Engineering*, pages 470–476. IEEE-CS Press, 1976.

[6] D. Gupta. *On-line Software Version Change*. PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, November 1994.

[7] R. Hall. *Agent-based Software Configuration and Deployment*. PhD thesis, University of Colorado, 1999.

[8] *Java Remote Method Invocation - Distributed Computing For Java*, March 1998. Available at: `http://java.sun.com`.

[9] J. Kramer and J. Magee. Dynamic Structure in Software Architectures. In *Proceedings of the fourth ACM SIGSOFT symposium on Foundations of software engineering (FSE'96)*, pages 3–14. ACM, ACM-Press, October 1996.

[10] M. D. McIlroy. Mass produced software components. In *Software Engineering, Report on a conference sponsored by the NATO Science Committee*, 1969.

[11] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2):53–65, 1993.

[12] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1997.