Abstract

With the development of mobile computing, applications have to be accessible at any time, from any device and in any place in a wide variety of execution contexts. Consequently, applications have to adapt to different terminal capabilities both for efficiency and ergonomics.

In this paper, we present a deployment infrastructure, called Smart Deployment Infrastructure (SDI), which aims to ease the installation of large distributed applications for any kind of user terminal. SDI is designed to take into account the execution context (including available resources and the user’s terminal capabilities) in order to bring an application to the user and adapt it to the execution context.

By considering an application as an assembly of distributed software components, SDI provides the opportunity to decide at installation time which components, among packaged components, will be instantiated on the terminal and which ones will be installed on other hosts.

SDI implementation demonstrates that the deployment infrastructure offers acceptable application deployment times and, at the same time, lowers both the application execution times and terminal resource consumption.

1 Introduction

The growing interest for using worldwide services from desktop computers has naturally led to the necessity to increase the accessibility of services. Mainly because of new mobile device capabilities, extending service accessibility has become possible. Users can access services at any time, from any place and from any device (e.g. from desktop computers to smart phones).

Mobile devices raise many new issues in application development and service infrastructure. One of the main cause is that mobile devices capabilities differ from desktop computers, not only in their user interface peripherals, but also in their memory and processing capacity. As a consequence, the software for a given service cannot be the same for every device. Because of peripheral differences, we can easily imagine that user interfaces differ from a mobile phone to a desktop workstation. Furthermore, the code on the mobile device has to be economic in terms of memory, battery and CPU consumption, as they are scarce resources. These factors create a growing interest in allowing the service to adapt itself to each device capabilities.

With pervasive computing, the number of access devices for each user increases, so that manual installation for each service on each device is no longer acceptable. Subsequently, service installation or deployment needs to be automatic and include adaptation to each device.

We aim at distributed applications built with software components running at different locations in large scale networks. Typically, such applications involve at least three components: the user interface component, the processing component and the data access component. All these components may be offered and deployed by several tiers. This type of application is currently a huge industrial trend.

Such applications are commonly built with middleware technologies such as CORBA [4] and SOAP [1] which ease the development of large scale distributed applications. New middleware, such as CORBA Components [11] and EJB [9] include component technologies which allow an application to be described as an assembly of components. This gives the opportunity to design new kinds of tools to deploy and connect the components of an application.

In this paper, we present a software infrastructure to support the deployment of large scale distributed applications that we call Smart Deployment Infrastructure (SDI). SDI offers automatic deployment of multi-component applications. SDI provides a deployment solution to customize the installation of applications for mobile users and to adapt to the device’s capabilities, to the user’s preferences and geographical location. With the proliferation of new mobile devices with variable capabilities, “smart” deployment solutions are becoming essential.

SDI is intended for Application Service Providers (ASPs). ASPs use software technologies to host and maintain applications for their customers. Then, customers can “rent” the use of the application rather than licensing the application from a software vendor [12]. For the customer, the benefit is to get up-to-date versions of applications without the need for a new installation. For the company
2 Software deployment and SDI

In this section, we first give a general definition of software deployment, then we specify the kind of deployment handled by SDI: a just in time deployment of multi-component applications which offers adaptation to each deployment context.

2.1 Deployment definitions

Today, a solution to the increasing complexity of software development is to split the development process into different tasks managed by different teams. The consequence is the creation of new roles in the development process: in addition to software designers and software implementers, we find software packagers who have to prepare coherent units of distribution, software assemblers who create an application with the composition of several packages and software deployers who are responsible for installing software on the computers.

In [5] deployment is defined as a set of inter-related activities: installation, activation, update, adapt, de-installation, de-activation. Installation represents the first deployment of the software on a computer. Activation is to start the execution of the application. Update is the modification of the software installation with a new version. Adapt is a modification in the installation triggered by the consumer computer. The deployment is a collection of tasks which occur at different times.

Deployment can be either done on a single computer: the software is monolithic and installed on a single host, or distributed: different pieces of software are installed on different computers (two machines for client servers applications or even more computers for multi-tier applications). In the latter case, the deployment is done either in a single operation or in several operations distributed in time.

Generally, the deployment is distributed in time: the installation process and the activation process are not performed at the same moment. But it can also be just-in-time: the installation and the activation are both performed at the latest possible time, when the user accesses to the service.

2.2 SDI deployment

With SDI, we focus on the process triggered when a user accesses a service. We target specific kinds of applications that are services: the application is “rented” by the mobile user on request. We don’t need pre-installation of the service on the access computer (also called the terminal) as the deployment is made just-in-time. De-activation and de-installation are done when the user leaves the service.

With this model, there are no updates necessary on the terminal. As the deployment is made just-in-time, we take the opportunity to stress the adaptation of the service to the execution context: terminal, user preferences, geographical location and system load.

SDI has been designed with the following assumptions:

- the terminal has connectivity to the network, as there is no pre-installed software on the terminal, SDI do not need connectivity;
- the user interface is the only component always required on the terminal, other components may be distributed on several hosts;
- the ASP provides computers that store the application packages and computers (called SDI hosts) that can host component instances. Thanks to the deployment process and SDI hosts, the terminal resources may be saved by locating heavily resource-consuming components outside the terminal.

The scope of software deployment is broad and some important but orthogonal aspects of deployment are not covered in this paper. Namely the security and licensing problems are outside the scope of this paper. The development of user interfaces for a wide range of devices is also another topic not covered by this paper.

2.3 Multi-component applications and deployment

Services sold to end-users are usually built with a graphical interface and several other processing units. In SDI, we consider that the whole application is built from software components, so each processing unit, such as the graphical interface, is a software component itself. A component type is defined through its interfaces (provided and required). The same component type may lead to several component implementations. For instance, one User Interface (UI) component type leads to several UI implementations, each one suited to a specific terminal type.

Furthermore, the division of the application into software components allows - if necessary - the execution of each component at a different site: a software component is a good unit of deployment [15]. There are several reasons which can lead to the distribution of the components: for instance taking advantage of different execution environments, handling security and reducing the use of terminal scarce resources.
In SDI, the architecture of an application, is described with a component assembly descriptor which lists component instances with their component type and connections between the components instances (connections from Required interfaces to Provided interfaces). It is important to note that SDI assembly descriptors describe only the logical layout of applications in terms of software components, but not the physical place for running the components. This enables the deployment process to take into consideration, not only the value of static properties, but also the value of dynamic properties at deployment time. Just-in-time deployment can thus take the appropriate decisions in order to make the best usage of available resources.

3.1 Deployment resources

We explain in this subsection the different resources involved in the SDI deployment process of multi-component applications.

SDI applications are described by an assembly of components. At the deployment time, these components can be in different states. Some components are already instantiated on an SDI host: they are ready to be connected to other components. Some others are only stored on a computer and are in a package state, waiting to be downloaded onto an SDI host for instantiation. The deployment process will be different for these two kinds of components.

We illustrate this concept with the sample application presented above (Figure 1). The three sample application components have different behavior during the deployment process. The bank account is linked to data, its location is fixed, it is instantiated before the user accesses the service and it will remain afterwards. We call it a pre-instantiated component. On the other hand, the two other components, the bank agent and the user interface, are not linked to any resource. They will be instantiated just-in-time, i.e. when the user accesses the service. The user interface will be instantiated on the terminal and the bank agent on any SDI host.

In order to support package components, SDI offers two kinds of servers: the package servers and the instance managers. A package server is a storage server which contains a collection of component implementation packages. When a component package needs to be instantiated, it is downloaded from a package server and instantiated in an instance manager. An instance manager role is to download and execute software component implementations.

The components of an application have different lifetimes. For example in our sample application, after the user leaves the service, the user interface component is de-instantiated from the terminal, the bank agent will be destroyed only when it has triggered the transfer, and the bank account is never destroyed.

The information on component state and component lifetime needs to be known for the deployment and undeployment processes. It is included in a deployment descriptor.

3.2 SDI Deployment Descriptor

An SDI application assembly descriptor (see 2.3) describes the component types involved in an application and the connections between these components. This description is useful for the deployment process, but it is not sufficient. In order to deploy applications automatically, it is necessary to add information such as the state, life-cycle and the constraints on the location of the components. For this purpose we introduce SDI Application Deployment Descriptors.

Figure 2 presents the SDI deployment descriptor for our sample application in XML. We present below, the different elements of this descriptor.
The context elements (lines 5, 15, 29 and 38) give information to build requests to the trader: a trader-location, a constraint and a preference. If the trader element is out of a component element (e.g. line 5) it gives information for all the deployment requests to the trader. In the example, the location of the trader is global and used for all the discovery requests.

In 3.1, we presented the two possible states of a component at the deployment time: package component or pre-instantiated component. The deployment process is indeed different for these two component states. For pre-instantiated components it comes down to finding their reference somewhere in the network. For package components, the deployment process is more challenging: it consists in finding a package server which stores the package component and a compatible instance manager to instantiate it. Then an instantiation of the package in the instance manager has to be done.

In order to select a deployment process, the state of each component is defined in the deployment descriptor. The lifecycle attribute of the component element gives this information. The entity value means a pre-instantiated component (e.g. the BankServer line 36). The session value means a package component whose lifetime is bound to the access time to the service (e.g. the BankGUI line 9). The process value means a package component whose lifetime is bound to an action (e.g. the BankAgent stays active until the bank transfer is triggered, line 27).

Depending on the lifecycle attribute, we find different elements in a component node. A pre-instantiated component can be discovered using the instance element (line 37) containing information to build the trader request. For a package component we need more information: the package element (line 14) to discover the package, the imlocation element (line 11) to determine the instance manager, and the instantiation element (line 22) to give an initial parameter to the instance manager for the instantiation of the component.

For the package components, the location of the instance manager in which it is instantiated is chosen at deployment time. Constraints or preferences on the location of the instance manager can be fixed in the deployment descriptor. The location of the user interface component is always fixed on the terminal (onhost element of imlocation, line 12). Some components may have co-location constraints and thus some components may require to be instantiated in the same instance manager as others (withcomponentinstance element of an imlocation node).

### 3.3 Discovery service for context awareness

SDI manages adaptation by finding the best match between an application deployment context and the resources available at the deployment time. The adaptation is supported by a discovery service which allows resources to be selected from a map of available resources: i.e. component
packages, instance managers and pre-instantiated components. The **deployment context** is constructed from the SDI deployment descriptor, the terminal properties and the user preferences. The deployment context is used to select the most suitable resources.

The SDI discovery service is built upon the CORBA trading service. The **CORBA trader** presents the following characteristics. It stores a repository of service types. Each type of service (i.e. type of resource) has been previously defined in the trader with a list of typed properties, properties may be mandatory or optional. There is inheritance of type of service and so an inherited resource gets all its parent resource properties. At registration time, a resource is stored with values given for all the mandatory properties. At discovery time, the **CORBA Trader** allows constraints with a constraint language (OCL: OMG Constraint Language) to be expressed. The **CORBA trader** allows results of a discovery to be listed in an order given by an OCL preference expression. Each CORBA trader may belong to a federation of **CORBA traders**: this feature allows the scope of a discovery to be extended and eases the scalability of the discovery service to large networks.

### 3.4 Deployment resources registration

Each deployment resource type (i.e. instance manager, pre-instantiated component and package component) is defined in our infrastructure by a **service type** which associates a resource with its properties. As they enable adaptation, the properties of each type of service have been chosen with care.

Figure 3 presents the different service types defined and their relationship. We have defined several “abstract” service types and we take advantage of their inheritance: as the package and the instance manager service types both inherit the properties of the abstract runtime service type, it is easy to find a package suitable to a given runtime, or a runtime suitable to a given package.

![Diagram of deployment resource service types](image)

**Figure 3: Deployment resource service types**

A **component identification** defines the component type. For the same component type, there are several implementations which are described by their **runtime**. The runtime gives information on all the requirements necessary to instantiate a component (in the case of a package description), or all the characteristics of the runtime provided (in the case of an instance manager description). The runtime includes information on the operating system, the execution environment, the list of software used/available. The **logical location** gives information on the network address of an instance manager, it may be useful for example for co-locating two components in the same local area network (e.g. finding two instance managers in the same LAN). The **geographical location** is useful either for giving a physical address (in latitude/longitude) of a resource or for giving a zone in which a pre-instantiated component serves. Finally the **load** service type is useful for selecting the least loaded pre-instantiated component or instance manager among several resources available.

A **package** is described by the component identification and the runtime required and also by the URL from which the package may be downloaded. A **pre-instantiated component** consists of its component identification, and also the identification of the instance manager on which it is instantiated (this information is necessary to select the least loaded replica), and some information relevant to the type of component such as the geographical zone about which this component may furnish information. An **instance manager** is described by the runtime provided, its logical and physical location, and its load.

All the resources available for a deployment are registered with their respective characteristics. In the preparation of the infrastructure stage, pre-instantiated components have to be made available. To enable scaling, some of them may be replicated. Package servers and instance managers have to be installed, they also have to be replicated and distributed in the network. We take advantage of the replication by updating the load characteristic when necessary with a monitoring facility (see [14]). An expression of constraints and preferences among this properties will allow the selection of the most suitable resource for each deployment.

### 3.5 Deployment Manager

The deployment process (figure 4) is controlled by a **deployment manager**. A deployment manager receives deployment requests sent by deployment clients running on users’ terminals.

The deployment manager handles the deployment with the help of deployment services: instance managers running on each host (SDI hosts and the terminal), package servers and discovery servers.

The deployment request is built by the deployment client. The deployment client collects different sources of information to build this deployment request: application information and context information. The application information includes the SDI Application Assembly Descriptor and the SDI Deployment Descriptor. The context information consists of: (i) a user profile (the preferences of the user e.g. language, service settings), (ii) a terminal description that details the terminal’s capacities (e.g. execution environment, peripherals) and (iii) user’s geographical location.

All this information leads to a deployment supervised by the deployment manager.
The terminal must only execute an instance manager and a deployment client. The deployment client is in charge of choosing an application to deploy, collecting terminal contextual information and sending the deployment request to the deployment manager. The deployment descriptor specifies the part of contextual information that must be filled by the deployment client (see line 19 and 23 of Figure 2).

The deployment manager can be executed on any computer. This gives the opportunity to execute it on an efficient computer in order to reduce the overall deployment time. The deployment manager is in charge to find packages suited to the terminal, to decide which components are installed on the terminal and finally to find less loaded instance managers suited to packages.

3.6 Experimentation and performances

The SDI infrastructure has been implemented with Java and CORBA middleware. As proof of concept, we build a simple banking application which has been described in Figure 1. We considered two kinds of end-user hosts: workstations and PDAs. For workstations, the banking agent is deployed on the terminal, the user interface is built upon Java Swing and provides a rich set of features. For PDA (iPAQ 3600), the agent is deployed on a SDI host to minimize resource consumption on the terminal, another implementation of the interface (using IBM MicroView libraries) suited for the PDA is dynamically chosen. The GUI implementation, as well as the host for instantiating the banking agent are transparently chosen by SDI, thanks to the deployment manager adaptation policies. We compared the execution with adaptation policies to an execution without any adaptation nor decisions (the agent is always instantiated on the user terminal). Concerning deployment time, the overhead of deployment policies is low for an end-user: 11.8% on Linux PC as terminal and of 8.42% on iPAQ as terminal. The overhead is more important on the Linux PC due to its faster processing and communication time vs. processing time. It offers serious gains on resource consumption on the PDA: the memory use is lowered by 20% and over 20% of the battery capacity is saved (in a 30 minutes experiment).

4 Related work

In this section, we compare SDI with related works. This section is divided into three parts: firstly, we make a short classification of deployment solutions, secondly we present the descriptors used by electronic deployment and finally we present different adaptation solutions proposed for mobile devices.

4.1 Deployment solutions

We classify deployment solutions by using the following criteria: (i) when they make the deployment (pre-installation or just-in-time deployment), (ii) what software is needed on the terminal and (iii) whether the target of the deployment is distributed or not (the deployment involves one or several hosts).

The most current deployment solutions consider software as monolithic: they pre-install (i.e. installation and use are separated in time) an application to a user’s desktop. InstallShield [6] and Microsoft Installer [10] are examples of such deployment tools for the Microsoft Windows operating systems. Both solutions are script-based installations that are responsible for copying files, registering libraries and components, creating windows registry keys and values for the application (for default settings) and handling licensing. These ad-hoc systems work perfectly when applications are not distributed and when the target platform is specific. All the software is pre-installed on the terminal. A specific installation has to be made on each host.

At the opposite extreme we find services accessed through web browsers. For these services nothing (except the browser) is pre-installed on the terminal. All the installation is on the server-side (e.g. Web servers and Java Server Pages). However, web access through html pages has limitations. Their interfaces are poor and the adaptation to small screens is not an easy task. The adaptation (web page appearance) is only possible through the web browser identification and has a limited scope (page layout). Furthermore, web pages don’t allow any off-line processing and they can’t benefit from object middleware technologies, such as events and methods calling.

The Dock research project [5] focuses on enterprise deployment of applications. It provides an architecture for the automatic installation and update of software for thousands of users. It is based on an event server which informs interested agents of software releases. The Dock architecture allows large scale software pre-installation and could
be used in SDI as a complementary tool for updating package servers.

Other deployment tools, such as JavaWebStart [7], allow just-in-time deployment. JavaWebStart automatically downloads all files necessary for the execution of an application. It is supported by Java Network Launching Protocol (JNLP) and JNLP files. JNLP files are XML descriptors which contain the application jar files URL as well as the link to pre-required packages and the activation method. This deployment is monolithic and does not support any adaptation.

SDI provides just-in-time deployment of multi-components applications with adaptation. This kind of deployment is made possible because of new components middleware such as CORBA CCM [11]. Just-in-time deployment has an overhead cost on the access time to the service because it adds the time needed for the adaptation and the downloading of files. But we believe that this overhead is well balanced by the advantages offered: no software installation on the terminal for users and a large latitude for adaptation.

4.2 Deployment descriptor

The evolution of software distribution from physical media to network distribution has already led to the definition of languages used to describe software distributions. Examples are OSD (Open Software Description) [16], an XML based language proposed by the W3 Consortium and the software MIF (Management Information Format) from DMTF (Desktop Management Taskforce) [2]. The software is described with its identification, its different implementations (with the location of the implementations for downloading and requirements on the runtime), dependencies (software dependencies) and installation procedures. Their major limitation is that they are limited to monolithic software.

CORBA CCM specifications [11] provide several level of descriptors. Software package descriptors describe implementation requirements (descriptors based on OSD), CORBA Component descriptors describe component design (interfaces required and provided) as well as information necessary for containers (persistence and transaction services) and the Component Assembly Descriptors describe an application as an assembly of components. The Component Assembly Descriptor includes information on components co-location (groups of components which must be on the same host). The deployment tool is free to choose hosts to instantiate groups of co-located components.

SDI is based on the own deployment descriptors. For adaptation purpose, SDI registers in the trader repository: packages, SDI hosts and pre-instantiated components descriptors. Furthermore, SDI defines separate assembly descriptors and deployment descriptors to separate assembly and deployment in the development process. All these descriptors are essential in the deployment and adaptation processes.

4.3 Mobile device capability adaptation

One key role of SDI is providing adaptation. Adaptation of applications is possible at different levels: at the development level by re-writing the applications and by compilations of applications; at the execution level by reflexivity; at deployment time; and finally by reconfiguration of running applications. In this subsection, we consider some research projects which are concerned both with mobile devices and the adaptation problem.

ICrafter [13] is focused on User Interface adaptation. It offers a tool to automate user interface software generation for each dedicated terminal. In SDI, we consider that several implementations of the UI have already been developed and stored in package servers and those implementations can profit from UI generation. The descriptions of both the terminal capabilities and packages requirements are used to select suitable user interfaces via the trader. This approach is similar to ServiceUI [17] which defines the properties of a user interface for registration in the Jini Lookup Service.

Adaptation to mobile devices capabilities may be done by reconfiguration of the application during the execution of the application. One example of this approach is AIDE [8] which offloads portions of an application from a resource constrained device to a nearby server. The portion of the application offloaded is a part of the graph of java classes calculated during the execution. Another work concerned with reconfiguration is Spectra [3]. It monitors, during the execution of the application, the resource usage of the different parts of the applications and takes the decision to offload part of the application. SDI also offloads part of the application from constrained devices to SDI hosts, but this decision is taken at deployment time. Taking this decision at deployment time saves the cost of execution before offloading and the cost of monitoring which is important for devices with low resources.

We think that mobile devices raise two new important problems: adaptation of user interface, and adaptation to low level capabilities. In SDI we solve these two problems at the deployment time: selection of user interfaces by finding adequacy between the description of both the terminal capabilities and component implementation requirements. The low level capabilities are resolved by installing, when possible, components outside the terminal and in this latter case into the least loaded host among a set of generic SDI hosts.

5 Conclusion

In this paper we have described a deployment infrastructure for distributed component-based applications, which is especially suited for added-value services accessed by users from mobile terminals. In order to save memory space on such terminals, the client-side code of the many applications that the user can execute does not permanently reside on the terminal, but is loaded on demand at use time.
for the selected application. In order to take user mobility into account, the geographical location of the terminal can be considered to select the appropriate components for a given execution of an application. The main features of this smart deployment infrastructure are the following:

1. The current distributed application deployment processes involve the end-user action for downloading and installing the client-side application code on the terminal and, in the case of mobile terminals, usually a desktop computer with a synchronization tool is necessary. On the opposite the SDI deployment process is fully transparent to the user. The logical layout of the application and the user preferences are stored on the terminal, and contain enough information to take appropriate deployment decisions without any user action. The deployment process ships the appropriate components directly to the mobile terminal, so that no partner desktop computer is needed.

2. The deployment infrastructure can handle dynamic information related to the user (terminal location) and to the computing environment (server load). The choice of the appropriate hosts which will instantiate the application components is made according to the current state of the environment, leading to a better resource usage. To this end, we rely on a scalable and portable dynamic load balancing service. The deployment is made “just-in-time” (when the user starts an application) to take advantage of this dynamic information.

3. Central to the deployment infrastructure is a discovery service, as exemplified by the CORBA trading service, which stores both static and dynamic data in a uniform way. For each host, static data are its type (mobile terminal, desktop or server), operating system, runtime, and installed utility software, as well as, for servers, pre-instantiated components or packages available for downloading. Dynamic data describe the current state of the environment. Lookup criteria for various properties can thus be processed at a single location, making the resolution process efficient.

4. The deployment process is driven by the Deployment Manager, which is a small program which interprets the logical description of the application to be deployed, issues the corresponding lookup requests (enriched with terminal static data and possibly terminal location) to the global repository, and sets up the connections between the selected components. This Deployment Manager is small enough to run on a mobile terminal.

5. The deployment infrastructure lowers development efforts for multi-platform applications by reducing platform specific code to a few components, like the user interface. Each component type can lead to several implementations that suit some specific properties in the discovery service and the choice is made just-in-time, when the user launches the application. In this way, the whole application architecture and generic components stay the same while current solutions require a full development process or re-engineering for each platform.

We have implemented a prototype of the deployment infrastructure using standard CORBA tools. A sample application has been used to demonstrate the flexibility of the deployment process, which can adapt both to terminal capabilities and to the current state of the environment by selecting the least loaded machine in a set of replicated server hosts. We have made some deployment measurements which show that deployment times are acceptable and that just-in-time deployment may be used to reduce low devices’ resource consumption.

This deployment infrastructure applies only to distributed applications which are designed according to a software component model. However, because of the many benefits of this programming model, we believe that this is not a serious limitation - mobile applications are a very new area in software development, so that there are few legacy applications. Although the prototype we used to validate the deployment infrastructure design was based on CORBA tools, the underlying concepts are generic enough to be implemented on other distributed component-based architectures.

References


