# Over-the-air Deployment of Applications in Multi-Platform Environments

Tore Fjellheim
*Queensland University of Technology,*
*126 Margaret St. Brisbane, Queensland, 4000,*
*t.fjellheim@qut.edu.au*

## Abstract

*Over-the-air (OTA) delivery of applications is important to support as it enables easy deployment and upgrades to applications, thereby reducing the disrupting effect which installations may have on mobile users. The mobile environment is highly heterogeneous, hence OTA servers must be able to deliver customised applications and also adapt their delivery mechanism to various protocols. This paper outlines our experience in designing an adaptive platform to enable heterogeneous OTA delivery. We have utilised the 3DMA architecture which includes features such as changing interactions, disconnection support and dynamic delivery of applications. We have extended previous work on this architecture by using it for implementing an adaptable webserver to support OTA over HTTP. A simple case study found that by allowing JIT packaging of data and behaviour, delivery of both content and behaviour can be tailored to the current context. This eliminates the need for pre-packaged deployment solutions that are difficult to employ in environments with dynamic variations in resources and context.*

## 1. Introduction

It is becoming increasingly important to support over-the-air (OTA) delivery of applications . This is especially true for mobile devices, where delivering new applications to support changing context, or upgrading existing applications should be done as easily as possible. Users should not have to surrender their devices to computer specialists, or have to connect to a powerful desktop computer, in order to receive new application updates. OTA delivery allows users to receive new applications and updates anytime and anywhere, enabling users to easily continue with their work. The heterogeneity of devices is, however, is a major problem for application providers. There are an increasing number of different types of devices being used, with each supporting different protocols and platforms. Mobile device support for Java, may for example vary from very limited support in case of the CLDC to very powerful J2EE devices. Even for Java, the write-once run-anywhere paradigm is therefore no longer true. Because of this heterogeneity, the protocols for delivery to various devices vary, and application servers must be able to support a wide variety of middleware, protocols, and platforms. Nevertheless. changing between protocols and adding new protocols should not require rewriting existing protocols or applications.

OTA delivery protocols can provide applications which are tailored according to the context of the device and the user. Limited memory, processing power, and bandwidth may restrict what functionality can be delivered, and the user's situation may determine what parts are selected for delivery. In order to support this type of delivery, applications must be component based and have appropriate metadata to enable the system to locate the correct components for deployment.

The first contribution of this paper is an outline of important design decisions in designing mobile applications for OTA deployment. We also outline what type of meta-data is required of components to enable easy selection of correct components for deployment. Our second contribution is, extending existing work on the 3DMA architecture [7], by desinging and implementing an OTA server capable of multi-platform delivery. This OTA server allows us to change the delivery protocol according to device and user context. As a more general principle, we argue that the same techniques can be used to facilitate many types of context aware service execution. This paper discusses deployment of applications, however the 3DMA architecture aims at supporting the full lifecycle of mobile applications including, offloading, disconnection support and composition. These issues are however outside the scope of this paper.

This paper will firstly outline related work in the area of mobile programming and delivery before we out-

Proceedings of the 2006 Australian Software Engineering Conference (ASWEC'06)
1530-0803/06 $20.00 © 2006 IEEE

line how to develop mobile applications, and describe them using appropriate metadata. Then we show examples of protocols for delivery of applications and how our architecture adapts to different user and device context. We then give a case example, and evaluate the system. Finally our our future work will be summarised before concluding.

## 2. Related Work

Several standards for how applications can be delivered to the mobile devices currently exist. The MIDP 2.0[1] specification describes a HTTP protocol for delivery, as does the WAP OTA and OSGi[2] standard. The problem with current implementations of these systems is that they only allow full application deployment. An application is delivered as a stand-alone Java JAR file. Upgrading or incremental delivery is difficult, as these are not component based. Programs are always delivered in their entirety. In addition, programs are typically legacy type Java programs which use synchronous communication for interactions, that complicates swapping components.

Research projects described by Gu et.al [10] and the Spectre project [8] only consider legacy applications which are typically very static. These projects do not consider the delivery of applications to the device, but rather the offloading of applications already on the device.

Taconet et.al [13] outlined how context aware delivery could be done to devices, and the Sparkle project [1] also outlined a mechanism for composing such applications by selecting the most appropriate components. Both of these projects considered delivery only within one platform, and assumed only one protocol for delivery. There is little decoupling between components, again making the process of changing an application after assembly very difficult. These projects mainly addresses more the actual selection algorithms, which compliment our work in that we are able to plug in any selection algorithm to our protocols.

Several research projects exist which consider decoupled communication in mobile devices, such as Lime [12] and Limbo [5]. These systems also only support one protocol, and therefore fail to interact with outside systems. Assumptions are made that all devices and users run the same system, and therefore the same protocols.

Although previous work has looked at how to deploy functionality depending on data, no previous project, to our knowledge, has looked at varying the delivery protocol according to the context. Thus we will show how we can build programs which use decoupled communication, thereby making it easier to deploy components (including upgrades), rather than entire applications. We will then show how we can assemble such programs and change the delivery protocol according to the context of the device.

[1] java.sun.com/products/midp/

[2] www.osgi.org

## 3. Programming for Dynamic Deployment

In this section we consider three important aspects when programming applications for dynamic deployment on multiple platforms. Firstly, they should be component based, secondly we require an appropriate metadata description, and finally, interactions should be decoupled. Each of these issues are detailed in the following sub-sections.

### 3.1. Components

To enable incremental delivery of applications and upgrades to occur, applications must be built from components. Components in our system are an aggregation of objects, similar to FarGo [14]. As the first step in creating the components we encourage the separation of data and behaviour. Keeping data and state separated from behaviour makes replacement easier, as we need not move state between components. Behaviour can be deleted and replaced easily, if necessary, without affecting the current state of the application.

One main problem in component design is determining the granularity. Our approach is based around the notion of activities. Mobile applications are oriented around the user and his activities, and aim to support the user in performing a set of activities. Components should therefore be separated according to the activity they perform on behalf of the user.

The object oriented approach defines objects around the notion of real world entities, encapsulating the data within an object, which also contains all the required behaviour to modify that data. We call this a horizontal partitioning strategy. A single user activity may use several objects, but only a little of the available behaviour or data in the objects. This has the disadvantage of users having to load in code which may remain unused. Loading in unnecessary code increases the load on the network, drains unnecessary battery and wastes memory space. We, therefore suggest a vertical partitioning strategy, based around the notion of a program slice [11]. A program slice is a subset of an application which contains only the code which affects a certain variable. A component is therefore a slice containing only what is

relevant for a given activity. The basic strategy in designing components can be divided into four steps:

- Define user activities
- Define application activities
- Create initial components
- Aggregate components

Firstly, we define the user activities. These are often very high level descriptions of what a user wants to do with the application, such as "Write a Document" or "Draw a Picture". We take these activities and decompose them into application activities. These are either, (1) activities which the application performs when the user executes the activity, or (2) activities that the application must perform before the user performs the activity. Data distribution is a typical example of the latter, where the user must have available data before any activities can be performed. The next step is to determine which of the following categories each application activity belongs to:

- User Interface (GUI)
- Processing (Proc)
- Communication (Coms)

If an activity fits into several of these categories, it should be separated into finer grained activities. After this process each activity is mapped to exactly one component, and each component contains only one activity. When loading the components we can now load exactly the application activities required for a given user activity. However, these components may be very fine grained, which may lead to unnecessary communications overhead. As components can not be decomposed during runtime, the next part consists of aggregating the components. Before detailing this process, we discuss the component metadata required.

### 3.2. Metadata

To enable the system to locate the correct components for deployment, a metadata description must be attached. When performing intelligent delivery of mobile applications, the component descriptions must be matched with user and device context. To enable this, we define metadata based around the notion of *Context Validity*, where each component is described in terms of the context it can operate under, or in other words, which context it is valid in [3]. As an example, we consider the following context elements:

- Activity
- Java Platform
- Code-Size

The activity element specifies the functionality of this component. If one component implements two very different activities, the programmer may consider creating two components instead. We have currently adopted a simplified view of the "activity" element, and describe it using a tuple consisting of $\{$activity,type$\}$. For example a user interface component may be described using "View GIF", or "Edit Text". More advanced descriptions can be plugged into the architecture, and a potential future direction is to look at more elaborate descriptions of activity, if required.

To cater for various devices, sometimes it is required to create several components for various different platforms. The Java Platform element, specifies which platform(s) a component runs under. A component which implements an Activity "View GIF" and Java Platform "MIDP CDC", could be replicated so that other devices which has other platforms can run the component as well.

The Code-Size element, specifies an approximate size of a component. Because of runtime variations, the size may sometimes be difficult to determine accurately. This element is used to estimate the memory requirements of the device, and thereby to deploy only the components suitable for the current constraints.

Alternatively other context elements can be used to determine the validity of components. For example data components may not require the activity field, but could for example be described in terms of which location it is valid in, or which user is allowed to access it. Behavior components can be described using context elements such as screen size, or bandwidth requirements. All these elements aid the selection algorithm in finding the most appropriate component for a given context and assist the programmer in determining the granularity of components.

The final decision of the granularity of components is up to the developer, however context validity can be a useful way of outlining how components should be decomposed. For example. sometimes increasing the validity of components will increase their resource consumption. To use the context validity to determine granularity, the general rule-of-thumb is that components should have a large context validity (useful in many contexts), but should take up little resources on the device.

As mentioned, a component can not be decomposed during runtime, yet components can be assembled from a set of component parts before deployment. Each of

these component parts can implement a certain part of the component. Thus several versions of a component part may exist. When a component is assembled for delivery, the part which implements the correct version of the required functionality is selected. After this, the component is assembled, and can be instantiated. An example of a component part could be an object which communicates with native code on the desired platform. This part would be selected based on which context validity is required.

For most current deployment systems, it is typically up to the user to find the correct application to perform the required task. It is often impossible to find the correct components based on the data the user wishes to, for example, process or display. To enable this in our system, data components are described to fit with the activity descriptions of behaviour components. Part of the data is described as a tuple $\{\text{type, name}\}$. If a behaviour component requests certain processing to take place on a data component, and the local system can not handle this request, another request may be sent remotely for deployment (or potentially remote execution) of a component which can handle the specified activity and data type.

### 3.3. Aggregating Components

This section describes three simple rules for aggregating components. This aggregation is performed manually by the developer. Firstly, the developer need to look for activities which are identical. This could occur if two user activities performed the same processing as part of their application activities. Identical application activities can be reused by multiple user activities. If there are two components with the exact same validity for all elements, one can be removed and need not be considered during component selection.

The second step in determining granularity is to use the context validity of components. If two components have very different validity, they are best kept separately. If the components are similar, they may be aggregated into a component which performs two application activities provided this aggregation does not significantly reduce the context validity for each of the supported activities. The definition of significant will depend on the type of context element which changes. For example, if one component with activity a1 is platform dependent, and another with activity a2 is not platform dependent, combining them into one component would cause a2 to be platform dependent as well, and would result in a significant change. If neither were platform dependent, and combining them would only cause a very slight memory consumption increase then the aggregation could be justified. In some cases the creation of several components with the same activity may be required. Each component should then have a distinct context validity. This means that different devices could perform the activity by using a different component suitable for their context.

The final method we use to determine granularity, is the way the activity is accessed, and its relation to other activities. For this, we outline three structures which aid in allowing incremental deployment. (Figure 1)

The first structure can be used where each component adds extra value to a user activity, however, only the first component is required. By delivering more components, we enhance the activity but consume more resources. If remote processing is available, then the local components could access the remote components to achieve the same result as having all components locally. A disconnection would result in a lower quality, but the user would not completely lose the desired functionality. An example of this is a word processor which has three activities, text editing, spell checking and grammar checking. The text editor component calls the spell checker component, which in turn calls the grammar component. Depending on the resources on the device, the device can support all activities, or only a subset of these activities.

The second structure is similar to the first in that the first component is required. However in this case there is no priority between the components which the initial component calls. A subset of the components can be chosen according to the preferences of the user or the device context. For example, if each of the subsets perform a different aspect to rendering of a page to be displayed on the device, and the device has little resource, the device can select to only render a few aspects. The picture would still be shown, but not in full detail.

The final structure is used where the data that the components modify is very tightly integrated with the behaviour. For example a graphics editor user interface component may have several activities such as “draw line”, “draw square” or “draw circle”. These activities can not be separated into different components as this would make it difficult for the user to change between the activities, since the user would also have to change between user interfaces. To allow for variable delivery we can therefore create several versions, ranging from components with few activities and little resource consumption, to many activities and high resource consumption. The appropriate version can be delivered to the device upon request, and can later be upgraded if desired. If the activities do not have to run locally, then a component with the full functionality can be accessed remotely to allow the user to use all activities provided a connection is available.

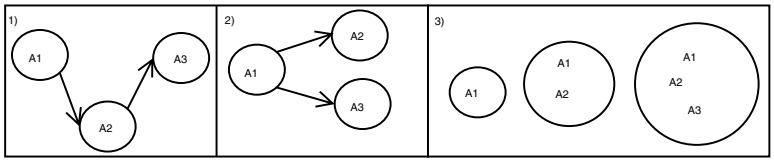


**Figure 1. Structures for incremental functionality deployment**

### 3.4. Interactions

To enable incremental delivery of components, communication between components should be separated from the actual processing performed. Components should be able to change what other components they interact with, so that we can easily select various components depending on context. This would allow delivery of applications where, depending on context, certain components would run locally and others would run remotely. For this reason, components should use event-based, decoupled communication.

Decoupled interactions can be supported in various ways depending on the capabilities of the platform. To support such execution on the server, and on certain capable devices we use a space based approach, this is further discussed in Section 4.1. In order to support decoupled interactions on limited mobile devices, the developer must implement several component parts, as discussed in Section 3.2. This includes component proxies. These proxies implement the communication between the components. When a component call is performed, the proxy object starts a separate thread which calls the component, thereby allowing control to return to the caller immediately.

Each component which can potentially run on the server, requires that a remote-access caller is specified. A remote-access caller is another proxy component part which runs on the device and sends messages to the server space. The component running remotely (in the AOS) can then read this message and execute. Alternatively the remote-access caller may instead of executing the component remotely, send a request for it to be deployed as a new component/application. The proxy object mentioned in the previous paragraph, links to the standard implementation of the component if it is running locally, or links to the remote-access caller if the component is installed remotely. This requires the generation of an extra connector in the space. This connector receives calls from the remote-access callers and translates them into objects which are put in the space. If the remote components require the ability to send data to a local component, another local "incoming message handler" component part can be created. This component part accepts all incoming requests and notifies the local component.

## 4. Dynamic Protocols for Deployment

After programming the application, a series of steps will have to be taken to select the appropriate components and to package them correctly for delivery. The packaging or selection algorithms are based on matching of the capabilities of the device with the context validity of the components. In this section we focus on how we can change between different algorithms dynamically during runtime. As all services used in the protocols are decoupled, and have no knowledge of each other, the algorithms can be easily changed during runtime if more advanced searching is required. The 3DMA architecture aims to be able to change interactions dynamically, and thereby allow more general context aware and personalised execution of services. This article uses dynamic delivery protocols as a case example of how changing interaction can allow context aware service execution.

This article will firstly present the architecture. Then we will give examples of how we extend the architecture by implementing two OTA delivery protocols, before finally showing how the system can change between these protocols depending on the user context.

### 4.1. The 3DMA Architecture

The 3DMA architecture is based around an object space based system called Active Object Spaces (AOS). Each entity which communicates with the space uses a variation of the standard tuple space operations defined by Gelernter [9]. These are read, write, take and notify. The 3DMA architecture extends the AOS by defining another set of entities called workers. Workers function as a rule base, and can react to the existence or entry of new objects in the space by writing new objects to the AOS, by creating new workers, or by transforming the objects. Workers can be enabled or disabled during runtime. Each worker has its own thread of control and therefore runs autonomously and can be changed independently of each other.

In the 3DMA architecture, each entity which communicates with the space has a set of post-conditions

and a set of pre-conditions. The post-conditions edit the output of the entity, according to a specified rule. This rule can be created by users according to user preferences, or the system to enforce a system policy. The pre-conditions will notify the entity of when it can execute. The notification can be triggered by a certain request, a change in context, or a combination of these. The decision to execute is not taken by the service itself, but by the worker. The separation of the actual processing from the decision of when to start processing is a key to enabling context aware service selection. The workers can furthermore be enabled or disabled depending on the client context or capabilities. Through disabling a pre-condition a certain functionality of an entity can be inhibited. This is similar to the notion of inhibitors taken by Braione [2], which can be used to enable or disable component behaviour. We apply this to context aware service execution, previously proposed by Cole et.al [4]. We extend their work by allowing service execution to vary depending on any type of context, not just location.

Another important entity in the 3DMA system are the connectors. Connectors allow systems which do not have the correct capabilities to connect to the space. The connector translates incoming messages to an AOS message which can be stored in the space. A connector may for example implement a connection to a MIDP device which does not have the capabilities to use the libraries required for a direct connection with the AOS, or it can be used to implement a Bluetooth connection to a remote system with a different protocol.

Internally in the AOS, objects are packaged with their class files in *Augmented Objects*. A connector implementing a connection to a device that is running the mobile AOS, will read the Augmented Object and translate it into a Mobile Augmented Object (MAO). The MAO contains both the serialised object and the class files. This object can then be received by the mobile device, which installs the application.

### 4.2. Example Protocols

Two alternative delivery protocols are described in this section. It is assumed that there is a way to retrieve the context (capabilities) of the local device through a well known query address. Although currently not all devices supports this ability, several industry standards (WAP, CC/PP) have been proposed to enable this. In addition, the space, loaded on the device, supports capability queries. This assumption can therefore be made without loss of generality. The protocols illustrate how our architecture can, through changing interactions, swap between protocols according to client capabilities. The protocols will both analyse the clients capabilities,

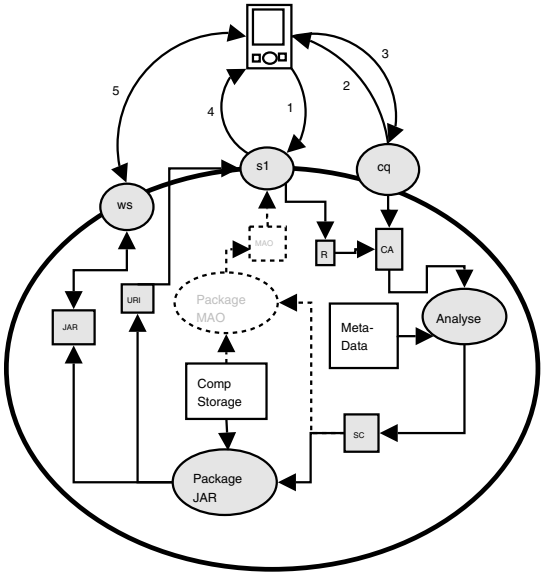


**Figure 2. A protocol for OTA delivery to a MIDP CLDC device**

and package the application accordingly. The two protocols outlined below are for:

- The MIDP CDC platform OTA delivery protocol with a simple communication space. Delivery and installation must occur via the MIDP OTA protocol.
- Standard 3DMA delivery protocol with a more elaborate class-loading space. Delivery and installation occurs directly in the space.

**Protocol 1** The first protocol provides context aware delivery to a MIDP CLDC device running a simple space with only basic communication facilities. This is illustrated in Figure 2.

The device will start by calling a GET on the adaptive web-server connector (S1) with the URI of the application it wants (R). The URI is given to the capability query connector (CQ), which queries the device's capabilities and stores them in the space (CA). This causes a request description to be created, and a session id is attached to this description. This session id is not changed by the various services in the request, and so the session remains throughout processing to distinguish it from other concurrent requests by other devices.

After the request has been created, the server stores the capabilities for future lookup. The capabilities and the requested URI are picked up by a service which analyses the context and the application (Analyse). The service then selects the correct subset of components for

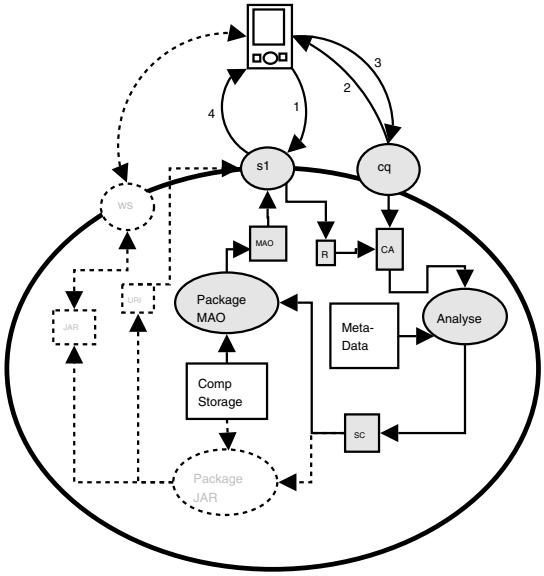


**Figure 3. A protocol for OTA delivery to a Personal Java device**

delivery. The output is a list of selected components (SC). The packaging service (Package JAR) reads the class files and assembles them into a JAR file. The selection and packaging process is done without human intervention. This JAR is stored as a temporary object in the space. The URI for this JAR is returned to the device. The device can use this URI to retrieve the assembled application through the Web Server connector (WS). If the application requires state transfer, then the remote space will assign the data to a particular URI, which the application, upon startup, can use for retreival.

**Protocol 2** The second protocol is outlined in Figure 3. It is similar to the first protocol until after the Analyse process. When the list of selected components has been produced the components are taken from the space and serialised. They are packaged with their class files, in a AOS format, and sent to the mobile device through the initial connector (S1). The mobile device can then simply unserialise and define the class files, before restarting the application.

### 4.3. Changing between the protocols

The difference between the abovementioned protocols is in the way the program is packaged and delivered. Also, the application in protocol 1 will itself have to retrieve its state after deployment, whereas in protocol 2, the state is transported with the application. If protocols require different transport mechanisms (e.g. bluetooth), we can easily change between connectors.

In these protocols, workers are used as a type of session information specific to a user, or potentially a group of users with the same device types. One worker will only listen to requests regarding a particular user. When the capabilities of a device are sent to the space, the workers for the user associated with these capabilities will react accordingly. This may cause certain workers to be enabled, or disabled based on the description of the capabilities.

When the capability description and current context is delivered in the space, the workers will be notified and react accordingly. If the platform capabilities of the device are specified to be of type MIDP, the pre-condition of the Package MAO service will be disabled, because it is not valid in the current context of the device. This results in the final delivery being a URI. The second protocol, is activated when the platform on the device is specified to be of type AOS. In this case the Package JAR will be disabled, and a Mobile Augmented Object will be delivered. Again, since workers in this case are specific to one user, the service will only be disabled for this user. Other users may have other services activated.

The workers in this example is created by the programmer, deployed with the services, and remain unchanged throughout the system execution. However, it is possible to deploy workers during runtime and even change existing workers. This can be used to allow users themselves to choose how to orchestrate services depending on changes in context. We are currently investigating interfaces to the workers, to allow insertion, removal and editing.

The outlined protocols are examples of pull by the user, however push protocols are possible as well. In a push protocol, a data item, or a behaviour component is selected to be sent to the device by a service outside the mobile device. In case of a data item, a query is sent to the device to find out if the device has the capabilities to process the data. If no such behaviour is found, then the appropriate behaviour is selected remotely and deployed according to the capabilities of the device, as discussed above. If the remote system simply seeks to push behaviour to the device, it is packaged and sent according to the mentioned protocols.

In the outlined case we changed the delivery mechanism depending on the device platform. These context elements change fairly infrequently. It is possible that workers can be used similarly to adapt interaction and protocols to changes in location or bandwidth. For example, we can introduce a new service which compresses state and data before sending it to the device. This service could be defined valid only when the bandwidth drops below a certain threshold. When the band-

width increases, no compression is done. This service could also be valid only if the remote device had a remote space with the capabilities to uncompress the data.

## 5. Case Study: Delivery of Surveys

The outlined system is advantageous in case characterised by a high level of heterogenity, where device characteristics can not be predicted, or where the frequency of deployment is almost as high as the frequency of use. Such as where each application is only used once or twice before being discarded. One example of such a case include surveys. Surveys are a popular way of acquiring useful information to achieve improvements in software design or enterprise processes as well as to get user feedback. Typically surveys are sent to many independent users. To start the survey, the system will push the surveys to potentially mobile users. The users can instantly fill in the survey and return the results. This allows the persons conducting the survey to retrieve data in real-time and analyse the data quicker than in traditional paper surveys.

Survey users may be employees in a corporation or students at a university. Such users are usually not equipped with mobile devices by the corporation they are involved with, but instead carry their own personal devices. This causes the environment to become very heterogeneous. When delivering the surveys to many users, it is therefore important that a heterogeneous network is supported through multiple protocols.

A survey application contains forms to be filled out, tasks to be performed, and the behaviour to support the forms and tasks. As an example, two types of users are considered. One user has a mobile phone with J2ME, and the other user has a PDA with direct connection to the AOS. Each of these users require different deployment strategies, but the same application.

### 5.1. Architecture of Application

There is only one user activity for this application, which is "fill in survey". This user activity requires four application activities. These are:

- display survey
- submit survey
- validate survey
- store survey

The "store survey" and "validate survey" activities are in separate components. The activities of display and submit are done by a single GUI viewer component. There is only one data component, and that is the survey itself. One viewer exists for each type of device which needs to be supported. The viewer components are annotated with metadata to show which platform it supports. There is only one generic Java validation component, thus the validation is kept separate from the viewer component to maximise the context validity of the validation component. If the mobile device can not run the validation component, this activity is performed remotely instead, and is therefore not packaged in the JAR or Augmented Object delivered to the device. Instead, a remote-access caller is delivered, which performs the required call to the validation component. Also an "incoming message handler" is delivered in case the validation component sends a failure message back to the viewer. The decision of whether to do validation locally or remotely would then also depend on factors such as the size of the remote-caller and the message handler component parts.

The survey application uses the first structure described in Figure 1 between the viewer and the validation, and the third structure between the display and submit activities. The data-flow of the application is shown in Figure 4. The application is activated upon deployment, and the survey viewer component reads the survey form and displays it. When the survey is completed, the viewer component sends it further for validation. This interaction occurs through the local space. If there is no local component which can handle the validation, validation is performed remotely. If the validation component finds that all inserted data is valid, the data is sent further to the database. If the data is not valid, then the survey is returned to the viewer to allow the user to enter different data.

### 5.2. Evaluation

An advantage of the AOS system, is the ease of which push-deployment is possible. When the system detects a user with an AOS installation, the application is packaged accordingly using the protocol described in Section 4.2. The survey data is selected with the relevant viewer and validation component, and the application is then pushed along with the survey to the device as a Mobile Augmented Object. The viewer and validation component are installed separately, and can be upgraded or removed separately of each other at a later stage. If the system detects that the remote device is low on memory, only the viewer component will be installed with the survey.

The basic protocol used in J2ME OTA, relies on an application management system (AMS) to install applications dynamically. This AMS takes as input, a JAD or JAR file which is then loaded and installed after user

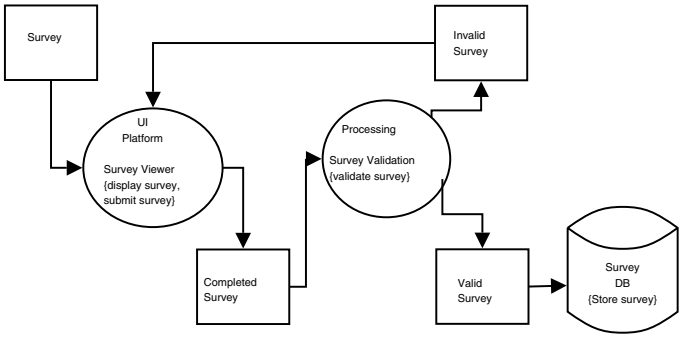


**Figure 4. The application components and data-flow**

a confirmation is given by the user. In a standard installation, the user will use the AMS or a web-browser to view a web-page containing links to available applications (JAD or JAR file). When the user selects one such file, the AMS will install the selected application. During execution of our protocol, the initial request for the application is interrupted by a capability query. This caused problems because it was only possible to have one open connection at a time. This thereby caused the web-browser request to be aborted. To overcome this issue, the protocol gives no reply to the HTTP request (unless in case of a failure), and the final result is pushed to the device to the local space (assumed installed). The space would then call the AMS which installed the application. The space would run on a dedicated known port, and the device would start up the application using the J2ME push-registry which initiates applications automatically when data is available on their port.

The amount of reuse acheived between platforms were limited in this application. The actual validation processing could be replicated, however it would only be worthwhile, for example if this processing was complex to re-implement. The increased complexity of the mobile application renders the question of whether it is worth the extra effort. In addition to the standard programming, the developer must create proxy objects, remote-access objects, and an extra connector. In future work we will be examining if it is possible to generate these objects automatically (or parts of them) based on a UML specification [6]. This would lessen the development burden, and make it easier to develop flexible mobile applications for multiple platforms. Design reuse is more likely achievable than code reuse.

## 6. Future Work and Conclusion

This paper has outlined an architecture for mobile devices, and shown how it can be used to support deployment in heterogeneous networks. The aim of the 3DMA project is to find requirements for how mobile applications should be built, propose an architecture for handling such applications and outline a methodology for development. Our methodology aims at providing developers with a way to design applications for mobile applications, in such a way that as must as possible of the same application can be utilized by devices regardless of current resource constraints and resource availability.

Mobile device developers, lack design guidelines for how such mobile applications are to be built. In this paper we have outlined initial steps towards a methodology for mobile applications. In our future work we will continue to investigate how to create applications in mobile environments, and we will aim at providing a more detailed methodology and libraries which developers can use for rapid development of adaptable mobile applications. Another future direction is to include the notion of personalisation into the mobile applications. This paper did not consider policies or user preferences during selection of services, however this is also part of future work. Our methodology will enable developers to easily incorporate workers into their applications to allow for context aware coordination and reconfiguration of both the remote and local services. We are designing a toolkit which will create pre- and post-conditions based on a UML specification and allow for context aware reconfiguration of the application.

There are several important aspects which have not yet been addressed properly in systems for over-the-air deployment of mobile applications. These systems mainly lack the ability to deliver stateful applications to various platforms by changing the protocol according to the given capabilities of the device. Typical systems assume that the remote device supports the correct protocol. Supporting dynamic deployment is vital in mobile environments. As new standards and platforms are being introduced, and users wield many different types of devices, the ability of OTA application servers to adjust protocols and content according to the capabil-

ities for the device and the context of the user becomes critical.

**Acknowledgments** The author is funded by an SAP-sponsored scholarship. This work is also funded by ARC Linkage Project LP0455394.

## References

[1] N. M. Belaramani, C. Wang, and F. C. M. Lau. Dynamic component composition for functionality adaption in pervasive environments. In *Proceedings of The 9th IEEE Workshop on Future Trends of Distributed Computing Systems*, San Juan, Puerto Rico, May 2003.

[2] P. Braione and G.P. Picco. On calculi for context-aware coordination. In *Proceedings of the 6th International Conference on Coordination Models and Languages*, 2004.

[3] Felix Bübl. Introducing context-based constraints. In *Fundamental Approaches to Software Engineering, 5th International Conference*, pages 249–263, April 2002.

[4] A. Cole, S. Duri, J. Munson, J. Murdock, and D. Wood. Adaptive service binding middleware to support mobility. In *Proceedings of The 23rd International Conference on Distributed Computing Systems Workshops*, Providence, Rhode Island, May 2003.

[5] N. Davies, S. P. Wade, A. Friday, and G.S. Blair. Limbo: A tuple space based platform for adaptive mobile applications. In *Proceedings of The 23rd International Conference on Open Distributed Processing/Distributed Platforms*, 1997.

[6] M. Dumas, T. Fjellheim, S. Milliner, and J. Vayssiere. Event-based coordination of process-oriented composite applications. In *Third International Conference on Business Process Management*, September 2005.

[7] T. Fjellheim, S. Milliner, and M. Dumas. The 3dma middleware infrastructure for mobile applications. In *Proceedings of the 2004 International Conference on Embedded and Ubiquitous Computing*, 2004.

[8] J. Flinn, S. Park, and M. Satyanarayanan. Balancing performance, energy and quality in pervasive computing. In *Proceedings of The 22rd International Conference on Distributed Computing*, 2002.

[9] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming*, 2(1):80–112, January 1985.

[10] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic. Adaptive offloading inference for delivering applications in pervasive computing environments. In *Proceedings of The 1st International Conference on Pervasive Computing and Communications*, Fort Worth, Texas, March 2003.

[11] Andrea De Lucia. Program slicing: Methods and applications. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149. IEEE Computer Society Press, Los Alamitos, California, USA, November 2001.

[12] G. P. Picco, A. L. Murphy, and G.-C. Roman. Lime: Linda meets mobility. In *Proceedings of The 21st International Conference on Software Engineering*, Los Angeles, California, May 1999.

[13] C. Taconet, E. Putrycz, and G. Bernard. Context aware deployment for mobile users. In *Proceedings of the 7th International Computer Software and Applications Conference*, 2003.

[14] Y. Weinsberg and I. Ben-Shaul. A programming model and system support for disconnected-aware applications on resource-constrained devices. In *Proceedings of the 24th international conference on Software engineering*, 2002.