AlfredO: An Architecture for Flexible Interaction with Electronic Devices

Jan S. Rellermeyer, Oriana Riva, and Gustavo Alonso

Systems Group, Department of Computer Science, ETH Zurich
8092 Zürich, Switzerland
{rellermeyer,oriva,alonso}@inf.ethz.ch

Abstract. Mobile phones are rapidly becoming the universal access point for computing, communication, and digital infrastructures. In this paper we explore the software architectures necessary to make the mobile phone a truly universal access point to any electronic infrastructure. We propose AlfredO, a lightweight middleware architecture that allows developers to construct applications in a modular way, organizing the applications into detachable tiers that can be distributed at will to dynamically configure multi-tier architectures between mobile phones and service providers. Through AlfredO, a phone can lease on-the-fly the client side of an application and immediately become a fully tailored client. Our experimental results indicate that AlfredO has very little overhead, it is scalable, and yields very low latency. To demonstrate the feasibility and potential of the platform, in the paper we also describe AlfredOShop, a prototype application for spontaneously controlling information screens from a mobile phone.

Key Words: Mobile Phones, Software as a Service, Universal Interface, OSGi

1 Introduction

The mobile phone is quickly transforming itself from a mobile telecommunication device into a multi-faceted information manager that can support not only communication among people, but also the processing and manipulation of an increasingly diverse set of interactions. The trend of a phone as a point of convergence for the user's activities, in some respects, has already begun. South Korea Telecom has introduced mobile payment technology and added RFID readers to phones to allow people to get information about shopping products [1]. Nokia has integrated GPS receivers to enable sports activity tracking, car navigation, and multimedia city guides [2, 3]. Motorola is researching how to allow its nomadic devices to interact with a car's components: if the car airbags deploy, the phone makes an emergency call; if the driver is maneuvering on a busy road, an incoming phone call is postponed; and if an urgent calendar entry is approaching, it can pop up on the car's display [4].

Applications of this type are usually based on ad-hoc implementations and customized to specific scenarios. In this paper, we investigate the software architectures required to support rapid prototyping of such applications. Our objective is to allow a mobile phone to acquire on-the-fly the necessary elements to immediately turn into a fully tailored client for interaction with any encountered electronic device.

In the past, distributed systems have supported interactions among embedded devices by either statically preconfiguring the execution environment or by dynamically migrating code, data, and service state from one device to another. However, the lack of flexibility of the former approach and the increased security risks of the latter have hampered their actual deployability in mobile environments. To overcome these problems and make our approach feasible for resource-constrained mobile phones, we propose AlfredO, a lightweight middleware architecture that enables users to flexibly interact with other electronic devices while providing ease of maintenance and security for their personal devices.

AlfredO stems from two key insights. Our first insight is that most interactions with electronic devices such as appliances, touchscreens, vending machines, etc., are usually short-term and ad-hoc. Therefore, the classic approach of preinstalling device drivers for each target device is not practicable. Instead, we propose to adopt a software distribution model based on the concept of software as a service. Each target device represents its capabilities as modular service items that can be accessed on-the-fly by any mobile phone. Our second insight is that the evolution of client-server computing from mainframes hooked to dumb user terminals to two-tier architectures (i.e., the classical client-server architecture) and to three-tier architectures (e.g, Web applications) has shown how partitioning server functionality yields better overall performance, flexibility, and adaptability. Therefore, we model each service item as a decomposable multi-tier architecture consisting of a presentation-tier, a logic tier, and a data tier. These tiers can be distributed to the interacting mobile phone thus configuring multi-tier architectures between the mobile phone and the target device.

AlfredO provides several benefits:

- *Scalability* and *ease of administration*: with AlfredO a resource-constrained mobile device such as a mobile phone becomes capable of supporting an unbounded number of diverse interactions. Instead of downloading, installing, and constantly updating the software necessary to interact with every conceivable target device, a mobile phone can simply acquire a stateless interface to the service of interest.
- *Flexibility*: AlfredO permits configuring flexible client-server architectures. A mobile phone, for instance, can host a thin client that simply acquires the presentation tier of the target service for the time of the interaction and discards it upon completion. Alternatively, a phone may also decide to acquire parts of the service logic tier with the aim of providing faster performance and responsiveness even in high latency networking environments.

- *Device independence*: To cope with the diversity of the input/output capabilities of the appliances and electronic devices a phone may need to interact with, AlfredO completely decouples the abstract design of a user interface from its implementation. Thereby, different renderings of the same abstract interface can be implemented on different devices. For example, a user interface can be rendered in one way on a notebook with mouse and large screen, in a different way on a phone with joystick and small screen, and in another way on a touchscreen.
- *Security*: AlfredO allows a phone to become a fully functional client by simply acquiring the presentation-tier of the target service. This can be achieved by simply shipping a "description" of the device's user interface to the mobile phone and letting the phone implement the actual user interface based on the abstract specifications. As this description file is not allowed to access the phone's local resources, this approach provides the security benefits of a sandbox model.
- *Efficiency*: AlfredO comes on a phone with a very low footprint of less than 300 kBytes. Yet, it permits interacting with a large variety of electronic devices while remaining latency-efficient. Our experiments show that a phone such as the Nokia 9300i can manage even 40 concurrent service interactions with an invocation latency of less than 150 msec over 802.11b WLAN. Furthermore, with AlfredO a phone is capable of turning in a fully operational client of a target service provider in a few seconds. This provides an end-user experience fully comparable to that of many other common applications available on phones, such as text editors, file managers, web browsers, etc.

We have implemented AlfredO using R-OSGi [5], a middleware platform that allows applications to be distributed using the modularity features of OSGi [6]. The OSGi framework implementation underneath is the very resource-efficient Concierge [7] platform. The next section gives an overview of the R-OSGi platform. Section 3 describes the design of AlfredO and gives insights into its implementation. Performance results are analyzed in Section 4. *MouseController* and *AlfredOShop*, two prototype applications built using AlfredO, are presented in Section 5. We discuss related work in Section 6 and conclude the paper in Section 7.

2 R-OSGi Overview

The R-OSGi [5] middleware extends the notion of OSGi [6] services to applications that run in a distributed manner. OSGi is an open standard which is maintained by the OSGi Alliance, a not-for-profit industry alliance with many major players of the software industry (like IBM, SAP, or Oracle) but also device vendors (like Nokia, Ericsson, or Motorola) involved. Traditionally, OSGi has been used to decompose and loosely couple Java applications into software modules. These modules encapsulate different parts of the whole functionality and their lifecycle can be individually controlled at runtime. For instance, each single functional module can be updated with a newer version without restarting

the application, which makes OSGi popular for developing long-running applications such as the firmware of hardware devices, or extensible applications like the Eclipse IDE [8]. Modules typically communicate through services, which are ordinary Java classes published under a service interface in a central service registry. Through the service registry service consumers can retrieve a direct reference to the service object of interest. Hence, OSGi provides a very lightweight communication model that avoids performance-adverse indirections known from container systems such as EJB [9].

2.1 Key Principles

With R-OSGi, many of the benefits provided by the OSGi paradigm can be leveraged to distributed systems. OSGi modules are distributed across several devices and the R-OSGi middleware transparently manages interactions between services located on different devices by exploiting the loose coupling of services. Typically, a service consists of an implementation (i.e., an instance of a class), one or multiple service interfaces under which the service is published, and a set of service properties. Since the concrete implementation of a service is hidden behind the service interface registered with the local service registry, R-OSGi can dynamically build proxies for remote modules which exhibit the same service interface as the one registered with the local service registry. Thereby, remote modules invoke service functions as if they were locally implemented and thus remain transparent to the network communication involved.

The typical assumption of static and immutable composition of software does not apply to the OSGi model. Instead, OSGi provides a platform where modules are dynamic and applications are prepared to react upon service failures or other kinds of interruptions. Hence, the potentially harmful side effect of introducing a network link into an application does not break the application model. Furthermore, disconnections between services can be mapped to module unload events, which the software can handle gracefully.

Remote service invocations are essentially synchronous and blocking remote communications. R-OSGi additionally supports asynchronous non-blocking interactions through remote events. Likewise, this addition does not introduce any new concept to the application model. The OSGi specification already contains an event infrastructure that many applications use when running on a single Virtual Machine (VM). R-OSGi transparently forwards such events when it detects that a connected remote machine has a registered handler for a specific event type.

2.2 Service Proxies

In the simplest case, a machine publishes a service under a service interface and a client machine acquires access to this service by establishing a connection to its machine. As part of the handshake, the meta-information about registered services is exchanged. These service descriptions are synchronized between the devices so that changes of services or unregistration events are immediately

visible to all connected machines. When a client wants to access a service, the service interface is shipped through the network and a *local proxy* for the service is created from this interface. This proxy is then registered with the local service registry as an implementation of the particular service. If it happens that the service interface references types provided by the original service module located on the remote machine, the corresponding classes will also be transmitted and injected into the proxy module (*type injection*).

However, the proxy itself can also provide more functionality than solely delegating service calls to the remote machine. *Smart proxies* implement the idea of moving parts of the service functionality to the client VM. The remote service can provide an abstract class as a smart proxy that is shipped to the client. All implemented methods run locally on the client machine whereas abstract methods are implemented as remote calls. Therefore, in this way, the service can explicitly push computation to the client side, if the client allows.

3 System Design and Implementation

Our approach aims to turn nearly-ubiquitous mobile phones into universal interfaces to the surrounding electronic world. Mobile phones nowadays have sufficient computation power to participate in sophisticated applications. However, they have by design inherent characteristics which distinguish them from typical general-purpose mobile computing devices, such as laptop computers. Phones have a different form factor, different display sizes and screen resolutions, and different input devices. Treating mobile phones like laptop computers overstrains their capabilities and provides unfeasible solutions. On the other hand, considering mobile phones as downsized versions of conventional computers neglects the benefits and unique capabilities they offer, such as built-in cameras and various sensor devices. Our goal is to look at the phone platform in its own right and leverage as much as possible its unique characteristics.

AlfredO incorporates three main mechanisms: (1) a *service-based software distribution model* for the support of an unbounded number of service interactions between phones and other electronic devices, (2) a *multi-tier service architecture* to flexibly configure the service interaction, and (3) a *device-independent presentation model* to achieve device independence and provide interface customizability.

3.1 Service-based Software Distribution Model

When a phone needs to interact with an electronic device available in the surrounding environment, from where does it obtain the required software? A simple approach would be to preinstall the necessary software on the phone and require a third party to authenticate it. Yet, this approach would result in poor flexibility as mobile phones will more likely need to interact with devices casually encountered in the environment. Furthermore, each time the original software is updated, the update needs to be propagated to all phones where the software was

previously installed. As the number and type of electronic devices increase, explicitly distributing, installing, updating software on each phone would become an unmanageable task.

Another possible approach is to dynamically transfer the software from the electronic device (or from the Internet) to the phone at the beginning of the transaction. Unfortunately, this approach would expose the phone to several security risks since in the common case the interaction occurs with unknown devices. Furthermore, downloading, installing, and configuring all necessary software is a time-consuming task that very often requires the user involvement and that consumes lots of communication and computational resources.

Our solution to software distribution is based on the concept of *Software as a Service (SaaS)*, which has been traditionally applied to Internet services. According to this logic, the new business model for most Internet's commodity software is not selling software, but building services enabled by that software. We believe SaaS can bring interesting benefits also to mobile phones, especially due to the impossibility for such resource-constrained devices to possess all software necessary for every possible interaction.

We adopt a service-based software distribution model where software available on electronic devices is made available to mobile phones in the form of flexible service items. Specifically, we package the functions provided by each electronic device as modular services that can be invoked, decomposed, and distributed using the service-oriented architectural approach of R-OSGi. In R-OSGi, services encapsulate whole functional units and dependencies between services are typically restricted to semantical dependencies at the application level. In the simplest case, a phone acquires on-the-fly the interface of a service of interest and discards it once the interaction is completed. In this way, phones are released from the duty of downloading, installing, and maintaining the software necessary to interact with all surrounding devices and the number of possible interactions can therefore grow unbounded. Furthermore, by letting phones acquire interfaces to arbitrary services high flexibility is provided and a phone's functonalities are not limited anymore to what their software platform and middleware layers are pre-configured for.

Another advantage that this service-based distribution model brings to mobile phones is its concept of software as a "process". Instead of software products that need to be engineered to exactly follow the given specifications, this model allows software to undergo frequent changes thus flexibly integrating a user's new requirements, technological advances, and emerging data models as soon as they become available. Hence, software on electronic devices can be changed and upgraded without compromising their interactions with the external world.

3.2 Multi-tier Service Architecture

We envision most interactions between mobile phones, called *clients*, and other electronic devices, called *target devices*, will occur in an ad-hoc manner. A mobile phone may contact a target service directly if its address is known (e.g., the contact address is provided at the bottom of the touchscreen) or upon service

discovery. R-OSGi supports several service discovery protocols such as SLP [10, 11]. Alternatively, the target device itself may periodically broadcast invitations to nearby devices. AlfredO makes the information about new devices available to the user and the user can decide whether to connect to a discovered device. Once the connection is established, the two devices exchange symmetric leases that contain the name of the services that each device offers. Thereby, the user can choose which service to invoke.

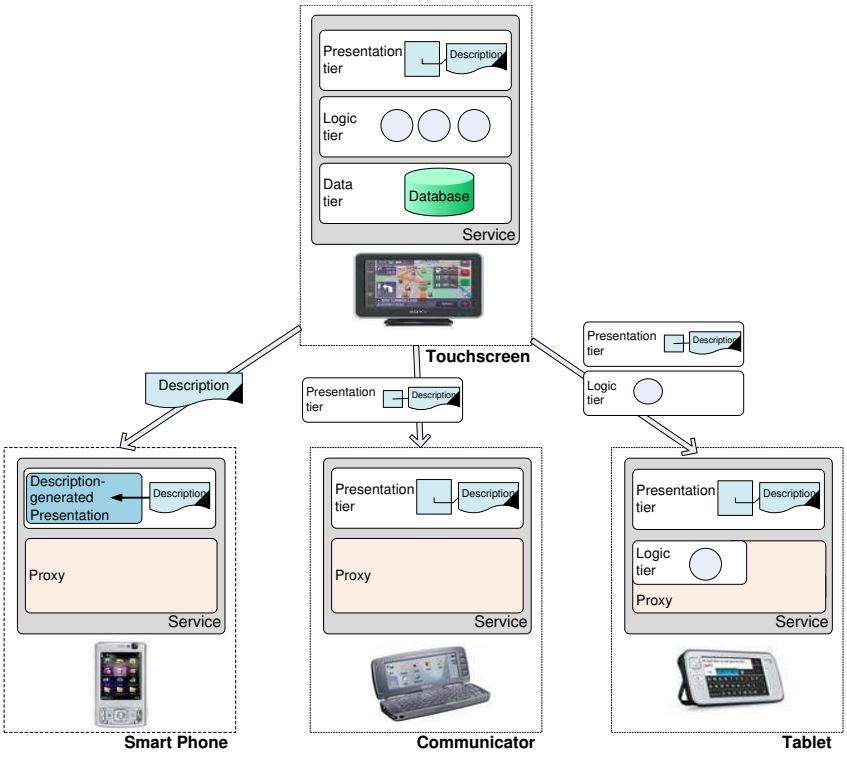

Fig. 1. Multi-tier service architecture

As Figure 1 shows, in our approach, services are built using a multi-tier software architecture consisting of a presentation tier (i.e., the user interface), a logic tier (i.e., computational processes), and a data tier (i.e., data storage). Tiers can be distributed according to different distribution logics and the boundaries of distribution can be adjusted dynamically. Typically, at the beginning of an interaction, the phone and the target device agree on the distribution configuration. This decision may depend on the phone's capabilities as well as its current execution context. For example, if a phone has low free memory, only the presentation tier is shipped to the phone, whereas if the communication link is unstable also the logic tier is shipped, thus reducing the communication overhead.

In the current implementation, the data tier always resides on the target device, while the presentation tier always resides on the client. By default the service logic tier is not transferred to the mobile phone, but we support also the case in which parts of the service logic are transferred to the mobile phone.

Initially, the target device provides the mobile phone with two elements: the interface of the service of interest and a service descriptor. The *service interface* is used by the R-OSGi framework running on the mobile phone to build a corresponding service proxy (see Section 2.2 for the definition of service proxy). The *service descriptor* consists of three parts. First, it contains an abstract description of the user interface (UI) necessary to support the interaction with the target service. As explained in the next section, based on the UI description each phone platform can generate a UI customized to the phone capabilities. Second, it includes a list of services on which the service of interest depends. Third, for each service in the dependency list it includes an abstract description of its requirements (e.g., other service dependencies, memory and CPU lower boundaries, etc.).

The default behavior is to generate a local proxy for the service interface and host only the presentation tier on the mobile phone. The client device runs the UI locally and triggers computation on the remote target device by interacting with the local proxy. As all computation and data management occur on the target device, this configuration minimizes the load on the resource-constrained phone. The mobile phone either self-generates a suitable UI based on the abstract description of the UI (see the example with the smart phone in Figure 1) or directly receives the UI from the target device (see the example with the communicator in Figure 1). We envision this will be the case for most interactions as they are likely to occur in unknown and untrusted environments. Indeed, a main advantage of this configuration is security. On the server side, the target device has full control on the implementation of its functions thus limiting attacks from malicious clients. On the client side, the device can decide which capabilities to expose to the target device in order to support the interaction. Furthermore, if only a stateless description of the UI is shipped to the mobile phone the configuration provides the security benefits of a sandbox model.

AlfredO also permits configuring more complex two-tier architectures, where the client not only acquires the presentation tier but also parts of the service logic (see the example with the tablet in Figure 1). The client can request additional services that appear in the list of service dependencies provided by the descriptor and run them locally. For each requested service, the client receives the associated descriptor (listing the service dependencies of the new service) and its service interface. In trusted environments, this approach can be effective in reducing the communication overhead and improving the application's responsiveness.

The descriptor provides a declarative description of the system comparable to other declarative approaches like XForms [12], but it allows for more flexibility. Indeed, our approach is not restricted to typical interfaces with input validation and content submission. Instead, it supports all the interaction patterns of the R-OSGi middleware, such as asynchronous communication through events, high-

volume data exchange through transparent stream proxies, and synchronous service invocations between services.

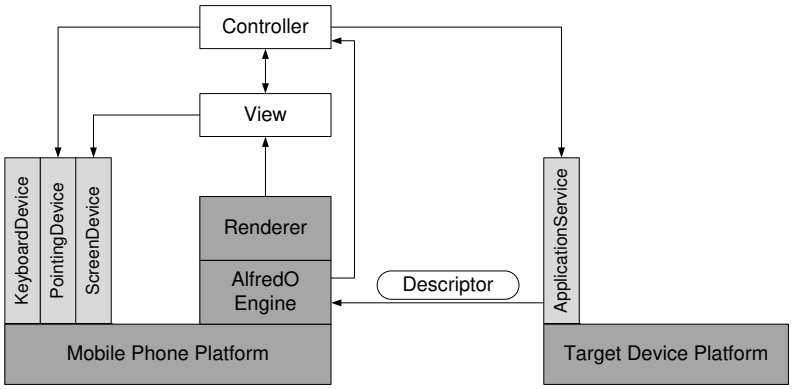

Fig. 2. An example of system configuration based on the service descriptor

The example in Figure 2 shows how a mobile phone can configure a customized client application capable of interacting with the remote target device. In a typical interaction some services will run on the mobile phone's platform (e.g., KeyboardDevice, PointingDevice, etc.) and others on the target device's platform (e.g., ApplicationService). The client device receives a descriptor of the target service and generates the application's *View* and *Controller*.

Instead of defining layouts that typically break on different screen resolutions and ratios, the UI is specified using abstract controls and relationships. The *Renderer* running on the mobile phone decides how to turn this abstract UI into an implementation (the application's View) that is tailored to the phone's hardware capabilities.

The *AlfredOEngine* generates the application's Controller based on the service requirements specified in the descriptor. The Controller defines how events generated through the UI (View) can affect the state of the application consisting of application data as well as configuration parameters and proxy settings. For example, at some point of the interaction, in order to improve the application's responsiveness the client can decide to acquire additional services currently running on remote devices. Likewise, the Controller also defines how events generated by the target device can modify the application's state. The Controller, for instance, may periodically poll a certain service method provided by the remote device and react to its changes by invoking another service method or by changing the implementation of a control command of the UI.

3.3 Device-independent Presentation Model

In our approach, we consider mobile phones as general-purpose platforms for interactions with various electronic devices and applications but without disre-

garding the specific characteristics of each device. Electronic devices provide a wide range of different input and output hardware capabilities. In many cases, these are customized to the functions each device is designed for. Clearly, a phone cannot offer every conceivable hardware capability, but capabilities of one device can be mapped to those of another one. For example, the mouse of a desktop computer is equivalent to the joystick of a phone or the knob of a coffee machine.

The service descriptor provides a device-independent specification of the UI. Ideally, an application developer should describe the input and output needs of his applications through this description, devices should provide specifications of their input and output capabilities, and users should specify their preferences [13]. The system can then self-implement a suitable interaction technique that takes all these requirements into account.

In AlfredO, the service logic remains agnostic to the specific hardware drivers available on each device. In other words, the logic tier builds on an abstract UI. Input and output capabilities that are used by a specific UI are modeled as OSGi services and accordingly their abstract definition is given by their corresponding service interfaces. All OSGi service interfaces are then organized in a hierarchy. For example, the *NotebookKeyboard* service implements the *KeyboardDevice* service interface which is used for entering characters as well as the *PointingDevice* service interface which is used for moving the mouse pointer through the cursor keys.

Depending on the capabilities offered by the interacting phone, the abstract description of the UI can be rendered differently, i.e, each phone generates the UI in a different manner. A device platform without a mouse or trackpoint can only build a GUI implementing the *KeyboardDevice* interface and without the *PointingDevice* service. Or a phone may have the choice to use a trackpoint or an accelerometer to implement the *PointingDevice* interface. Likewise, on a phone a *KeyboardDevice* interface may be implemented using the small keyboard of the phone or a handwriting detection that operates with a stylus. In principle, multiple devices can be federated to implement the abstract specifications of the given UI. Furthermore, the UI can be partly on the local phone, partly on the target device, and partly on other external devices. For example, in Figure 2, the phone may decide to use a notebook's screen with larger resolution; in this case, the *ScreenDevice* service would be implemented remotely by the notebook platform and invoked on the phone through a local proxy.

The implementation of the UI can use different rendering engines that are provided by the client platform. Currently, the default rendering engine produces a Java AWT [14] application where the abstract user interface is rendered with AWT panels. Another supported rendering engine is based on the SWT [15] toolkit. This is especially useful for devices for which an implementation of the Embedded Rich Client Platform (eRCP) [16] exists. As eRCP runs on top of OSGi, it requires only a small set of additional bundles to turn an eRCP device into an AlfredO client. For phone platforms that do not support any graphical toolkit, it is possible to use a web browser that is fed by a servlet [17] renderer.

This produces HTML enriched with AJAX [18]. In this case, the web browser can serve as a graphical environment to interact with the headless AlfredO platform.

4 Experimental Evaluation

The goal of this experimental evaluation is threefold. First, it quantifies the footprint of AlfredO on resource-constrained mobile phones. Second, it assesses the latency to acquire the presentation tier from a target device. Third, it evaluates the scalability of AlfredO in terms of number of parallel service interactions that can be supported between a mobile phone and any target device.

4.1 Resource Consumption

As AlfredO is based on a layered and decomposable architecture, the actual size of the software stack on a phone depends on the actual deployment and the size of the renderers utilized for generating the user interface. The minimal core platform consists of an OSGi framework, the R-OSGi system, and the AlfredO core functionality. Using the very lightweight Concierge [7] OSGi implementation, in total this amounts to a footprint of about 290 kBytes. The renderers typically have a footprint of around 40 kBytes, except the servlet renderer that has additional dependencies from the OSGi HTTP service implementation and adds a total of 160 kBytes when running with the Concierge HTTP service prototype.

To assess the runtime costs, we use our two prototype applications (*MouseController* and *AlfredOShop*), which are discussed in detail in the following section. The proxy bundle generated for the *MouseController* consumes 6 kBytes on the file system and the *AlfredOShop* proxy bundle takes 7 kBytes.

Runtime memory consumption is hard to measure on embedded devices like phones because it depends on the state and the timing of the garbage collector. In a controlled environment on a desktop Java VM, however, the *MouseController* consumes about 200 kBytes of memory and the *AlfredOShop* 30 kBytes. The higher memory footprint of the *MouseController* application is due to application-generated data (i.e., the RGB bitmap image that the application periodically receives from the controlled device and that is stored in the local memory).

Summarizing, the resource consumption of AlfredO is minimal and therefore very well suits the resource requirements of mobile phones. The whole software stack has a footprint that can nowadays be easily compared with the footprint of an average single page of an internet website. For comparison, a hardcopy of the first page of the ETH Zurich web site (which is not especially fancy), creates a storage footprint of 200 kBytes. Proxy bundles for services that are no longer available are not cached but immediately uninstalled as soon as the interaction is terminated. Therefore, an AlfredO client does not store outdated data over time. Compared to device drivers or web clients, this is clearly an advantage and allows much more versatile interactions. The memory footprint is also not an issue for today's mobile phones. Even when more complex services and user interfaces are involved, the memory is not a limiting factor.

4.2 Latency Performance

In these experiments we measure the time a phone client needs to contact and establish an interaction with a target service. In these tests the phone acquires only the presentation tier of the service. This includes the interface of the service of interest, a description of the service requirements and a description of the abstract UI. We use a Nokia 9300i that runs a 150 MHz ARM9 processor and offers both 802.11b wireless LAN and Bluetooth (BT) connectivity, the latter, however, only with the CLDC VM but not with the CDC VM. As R-OSGi runs on the CDC VM, for the experiments in BT networks we employ a second phone, a Sony Ericsson M600i that runs a 208 MHz ARM9 processor. Both phones interact with a regular desktop machine (single core Pentium 4 class).

We measure the initial start time necessary to contact a service and acquire its interface, build the proxy bundle, install it on the local R-OSGi framework, and get the proxy running on the mobile phone (i.e., Start proxy bundle). The experiment is run with the two different applications, the *MouseController* and the *AlfredOShop* service. The amount of data transferred to the phone accounts for about 2 kBytes for each application.

Table 1. Initial delay for service interaction on a Nokia 9300i over WLAN

	Nokia 9300i WLAN, in ms	
Operation	**MouseController**	**AlfredOShop**
Acquire service interface	94	110
Build proxy bundle	3125	3110
Install proxy bundle	703	703
Start proxy bundle	1000	359
Total start time	4922	4282

Table 2. Initial delay for service interaction on a SonyEricsson M600i over BT

	SE M600i BT, in ms	
Operation	**MouseController**	**AlfredOShop**
Acquire service interface	263	312
Build proxy bundle	1882	1881
Install proxy bundle	259	260
Start proxy bundle	892	246
Total start time	3296	2699

Table 1 and Table 2 report the results. The network communication necessary to acquire the service interface is not the dominant factor in determining the total start time. Building, installing, and starting the proxy on the phone takes much longer. Therefore, the time is not primarily influenced by the size of the service interface. On the other hand, it strongly depends on the phone platform

in use. On the SonyEricsson phone, which has a more powerful processor, the performance is in average 40% faster. However, from a user point of view, total start times of both applications are more than acceptable if compared to startup delays of typical phone applications. On the 9300i Nokia communicator, for instance, the startup time of the built-in *Document* text editor application is about 3 seconds, the startup time of the *FileManager* is around 6 seconds, and starting a web browser and displaying the default Nokia homepage takes about 17 seconds (assuming the phone is already connected to the Internet).

4.3 Scalability

An important goal of AlfredO is to ensure that phone clients and service providers can scale to a sufficient number of interactions. We first assess the scalability performance of the service provider and then of the phone client.

In the first set of experiments, the server runs on a typical desktop machine, i.e., a single core Pentium 4 class. To put the server under stress, multiple concurrent clients run on another machine of the same type. Client and server machines are connected through a 100 Mbit/s ethernet network link. Clients connect to the server and perform a service invocation of the same service method every 100 ms. In the tests, a new client instance is started every second. We measure the average invocation time of the last client instance, which is started when all other client instances are already running. The average is computed over a period of at least 90 seconds.

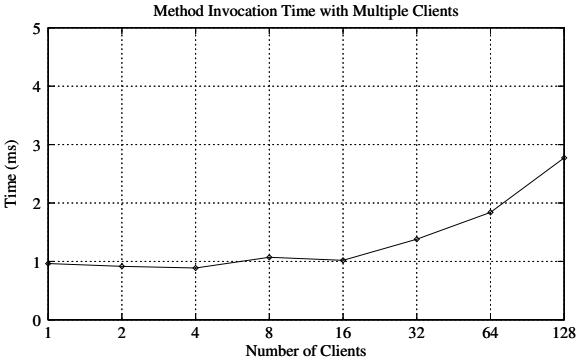

Fig. 3. Invocation time with multiple concurrent clients on a single machine

Figure 3 shows the results of this experiment. The server performs very fast and provides an average invocation time of only 1 ms. The invocation time slightly increases with an increasing number of clients but even with 128 clients the invocation time is below 2.5 ms.

However, with this configuration we could not run tests with a number of clients larger than 128. This is because the client machine reaches its upper

bound when running 128 Java VMs concurrently. To investigate the scalability boundary of our system, we therefore ran a second set of tests, in which the clients run simultaneously on a cluster of six machines and perform the same experiment as before. The six client machines are two-processor dual-core AMD opteron 2.2 GHz machines and are connected through a switched 1000 Mb/s ethernet network. The service provider is an identical machine in the same network.

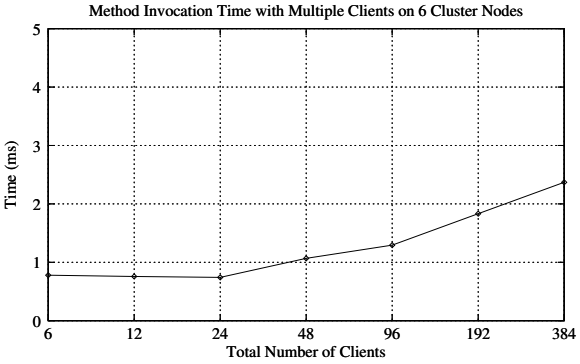

Fig. 4. Invocation time with multiple concurrent clients on six cluster nodes

As depicted in Figure 4, also in these tests AlfredO performs very efficiently. The server can handle 384 client interactions while providing an average invocation time of 2.2 ms. Given the latency increase observed with 768 concurrent clients (not shown in the figure), it can be estimated that the scalability limit is between 400 and 800 clients. Specifically, with 540 clients the latency is 3.6 ms, whereas 600 clients lead to a delay over 42 ms per invocation. Therefore, we can conclude that the upper boundary in this configuration is about 550 concurrent clients. This boundary on the server scalability enormously exceeds the requirements of the applications that we currently envision. A service running on a coffee machine, on a touchscreen in a shop, or on a vending machine may need to support an average of 2-3 concurrent users and a maximum of 30 concurrent users, which still represents only a 5% of the available service capacity.

In the second part of the study, we investigate the scalability of the client side. The client runs on a Nokia 9300i phone. This time we install 1024 distinct services on the server. The mobile phone is configured to get a new service every 10 seconds and then continuously invoke a service method on all acquired services every second. The measured values represent the average invocation time of the first instance in each of these time windows over multiple runs of the experiment. Figure 5 shows the results. The dotted line represents the latency baseline, an ICMP ping over the network link. As observed on the server side, AlfredO provides high scalability on the client side as well. The average latency is around 100 ms.

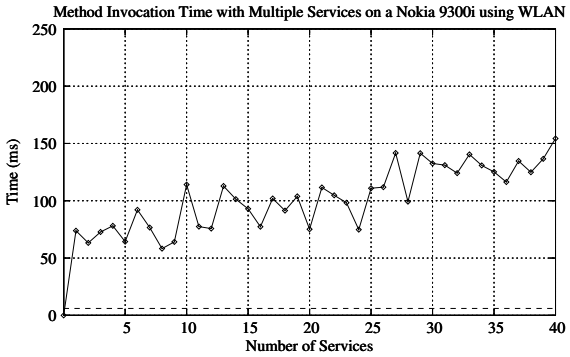

Fig. 5. Invocation time on a Nokia 9300i Phone over 802.11b WLAN

We then ran the same experiment on the Sony Ericsson M600i phone (see Figure 6) using the built-in Bluetooth 2.0 interface. The results are comparable to the previous platform even though WLAN has in theory an almost four times higher bandwidth. However, since the messages exchanged are fairly small, the bandwidth is not a dominating factor unless a larger amount of data is shipped through the network. For instance, the type of network employed had a larger impact on the latency to acquire the service interfaces (roughly 2 kBytes of data) in the experiments reported in Table 1 and Table 2.

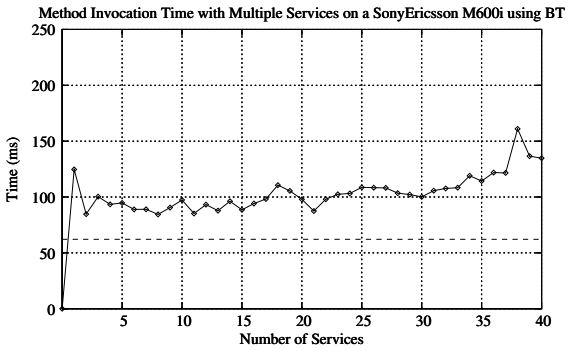

Fig. 6. Invocation time on a Sony Ericsson M600i phone over Bluetooth 2.0

5 Prototype Applications

Using AlfredO we have built two prototype applications: *MouseController* and *AlfredOShop*.

5.1 MouseController

To demonstrate how AlfredO allows a phone to quickly transforming itself in a universal remote controller we have built *MouseController*. This is a very simple but very powerful service that allows a mobile phone to control the movement of the mouse on a notebook's screen. Figure 7 shows how a browser application running on a notebook can be controlled using the communicator's cursor keys. In the figure, the user is minimizing the window opened on the notebook's screen.

The user interface of the same service (i.e., controlling the mouse pointer) is rendered in different ways on different phone clients depending on the capabilities of each particular device. For example, the description of the user interface retrieved by the phone specifies that input commands utilize the *PointingDevice* service interface. On a Nokia 9300i phone, this interface is implemented with the cursor keys of the keyboard. On an iPhone, the same interface is implemented using the integrated accelerometer, thus allowing the user to move the mouse pointer on the notebook's screen by moving the phone itself.

Fig. 7. MouseController running on a Nokia 9300i phone

On the phone's screen a small snapshot of the notebook's screen is displayed. Since the interactions causing the mouse to move are typically occurring at a high update rate, there is often not enough network bandwidth left to send the large updates of the snapshot back to the phone. Therefore, the application uses asynchronous events between the service and the phone and sends updates whenever there is enough bandwidth.

5.2 AlfredOShop

AfredOShop is a prototype application that allows users to interact with information screens using their mobile phones. For instance, by interacting with an

information screen placed behind a shop window, a user can browse and compare shop's products even when the shop is closed (e.g., when passing by in the night). The mobile phone is used as a remote controller of the screen on which the product description is visualized.

Figure 8 shows *AlfredOShop* running on Nokia 9300i phone while the user is browsing the details of the beds available in the shop. In this example, the information screen is a notebook computer that displays the shop's interface.

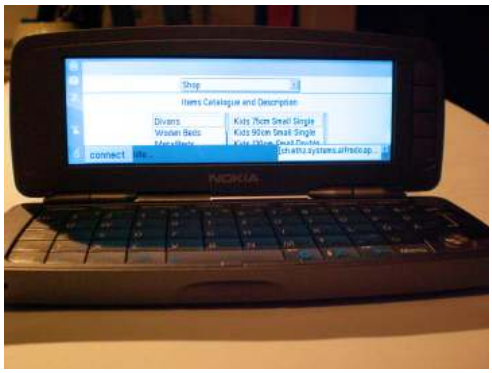

Fig. 8. AlfredOShop on a Nokia 9300i

Fig. 9. AlfredOShop on an Apple iPhone

Implementing this application using AlfredO brings several benefits both to the customer and to user. On the customer side, the application can contribute increasing the shop's revenue by making the shop accessible 24 hours a day. Furthermore, a shop's owner does not incur in any security risk because AlfredO provides him a full control on which information to display. On the user side, the interaction only requires a phone's keyboard and cursor. Security is guaranteed because only a passive description of the UI is retrieved from the information screen and no computation takes place on the actual phone.

Since the *AlfredOShop* application uses a rich user interface with multiple informational and control widgets, AlfredO plays an important role in adapting the user interface to different phone capabilities as well as to different screen sizes and output devices. On the Sony Ericsson M600i, AlfredO uses an AWT rendering. On the Nokia 9300i an eRCP renderer efficiently creates the service presentation in SWT. Furthermore, as the Sony Ericsson phone has a portrait-oriented display and the Nokia a landscape-oriented display the output interface is adapted accordingly. The iPhone platform currently does not run any Java implementation that supports the graphical toolkit of the device. However, the AlfredO servlet renderer can be used to generate an AJAX-enhanced set of

dynamic web pages that can be viewed and controlled through the built-in web browser (Figure 9). In terms of functionality, the AJAX version provides the same features as the other versions, such as explicitly connecting to a known service, getting informed of newly discovered devices, and switching between the user interfaces of different devices and their services.

6 Related Work

Research on distributed systems and ubiquitous computing has variously focused on the problem of how users can dynamically interact with devices embedded in the surrounding environment. Proposed solutions can be roughly grouped into two categories: those that assume an *a priori* configuration of the interacting devices and those that configure the devices on-the-fly by downloading the necessary software from the Internet or by migrating it from a nearby device.

For example, systems like Personal Server [19] provide the user with a virtual personal computer environment. Data and code necessary to interact with external input/output interfaces are pre-stored and pre-installed on the mobile devices. As these approaches require a pre-configured infrastructure they can suit only static environments.

The second class of systems allows for increased flexibility and can therefore suit dynamic environments. Technologies based on mobile code [20] have been considered in several domains, but they are usually disregarded because of their security and trust concerns. These security problems are alleviated by systems that rely on a third party (e.g., Internet) for authentication purposes. CoolTown [21] assume a web presence that connects all embedded devices. Each device advertises its presence and offered services through URLs. SDIPP [22] augments the Bluetooth service discovery protocol with web access. A user can download the required service interface directly from the nearby device using Bluetooth or from service directories implemented as web services.

Although these approaches can provide some flexibility, AlfredO achieves even higher flexibility by organizing the services into decomposable tiers that can be distributed to configure one-tier or two-tier architectures among the interacting devices. Security is also improved by transferring to the mobile phone only a description of presentation tier, thus allowing the device to self-implement its UI. Furthermore, AlfredO does not rely on Internet connectivity and targets the resource constraints of mobile phones: it is lightweight and highly efficient.

Web services have also been considered in this context. Microservers [23] embed web servers in Bluetooth devices and use WAP over Bluetooth for communication. Specifically tailored to mobile phones, Mobile Web Server [24], also known as Raccoon, provides a mobile phone with a global URL and with HTTP access thus enabling a mobile phone to host a universally accessible website. Even though web services are not employed in the current implementation, they could be utilized as well. We opted for R-OSGi because it provides a lightweight implementation optimized to minimize the resource consumption on phones.

We borrow the notion of abstract user interfaces that can be rendered in different ways on different devices from other research projects [25, 26], especially in the field of human-computer interaction [27]. However, these projects mostly focus on how to generate the user interface and typically rely on centralized infrastructures. Instead, our focus is on the system and infrastructure issues.

7 Conclusions

AlfredO enables mobile phones to become universal clients for interaction with an unbounded number of heterogeneous electronic devices. Ultimately, this approach blurs the boundaries between mobile phones, appliances, and other electronic devices and let resource-constrained mobile phones acquire larger value from services that reside elsewhere. Compared to previous approaches, AlfredO makes service interactions fully decomposable processes, thus providing high flexibility and customizable security. In addition, a mobile phone benefits from such an approach also in terms of easier administration (no need to install software) and automatic maintenance. Experience has shown that our implementation is highly efficient and it comes on phones with a file footprint of only 290 kBytes. Future work on AlfredO includes an online optimization mechanism to customize service distribution at runtime and an automatic distribution mechanism of the data tiers to provide transparent synchronization.

Acknowledgments

The work presented in this paper was partly supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under the grant number 5005-67322, and partly by the ETH Fellowship Program.

References

1. South Korea Telecom: SK Telecom Releases Upgraded Mobile RFID-based "Touch Book" Service. `http://www.sktelecom.com/eng/jsp/tlounge/presscenter/PressCenterDetail.jsp?f_reportdata_seq=3883` (2007)
2. Nokia: Nokia N95. `http://nokia.com/n95` (2007)
3. Nokia Research Centre: Sports Tracker. `http://research.nokia.com/research/projects/SportsTracker/index.html` (2006)
4. Motorola: Nomadic Device Gateway. `http://www.motorola.com/content.jsp?globalObjectId=8253` (2006)
5. Rellermeyer, J.S., Alonso, G., Roscoe, T.: R-OSGi: Distributed applications through software modularization. In: Proceedings of the ACM/IFIP/USENIX 8th International Conference on Middleware (Middleware'07). Volume 4834 of Lecture Notes in Computer Science., Springer (2007) 1–20
6. OSGi Alliance: OSGi Service Platform, Core Specification Release 4, Version 4.1, Draft. (2007)

7. Rellermeyer, J.S., Alonso, G.: Concierge: A service platform for resource-constrained devices. In: Proceedings of the 2007 ACM EuroSys Conference (EuroSys'07), ACM (2007) 245–258
8. Eclipse Foundation: Eclipse. `http://www.eclipse.org` (2001)
9. Sun Microsystems: JSR 220: Enterprise Java Beans, Version 3.0. (2006)
10. Guttman, E., Perkins, C., Veizades, J.: Service Location Protocol, Version 2. RFC 2608, Internet Engineering Task Force (IETF) (1999) Available at `http://www.ietf.org/rfc/rfc2608.txt`.
11. J. S. Rellermeyer: jSLP project, Java Service Location Protocol. `http://jslp.sourceforge.net` (2008)
12. W3C: XForms 1.0 (Third Edition). (2007)
13. Myers, B., Hudson, S.E., Pausch, R.: Past, present, and future of user interface software tools. ACM Trans. Hum-Comput. Interact. **7**(1) (2000) 3–28
14. Zukowski, J.: Java AWT Reference. O'Reilly (1997)
15. Eclipse Foundation: SWT: Standard Widget Toolkit. `http://www.eclipse.org/swt/` (2004)
16. Eclipse Foundation: embedded Rich Client Platform (eRCP). `http://www.eclipse.org/ercp/` (2006)
17. Sun Microsystems: Java Servlet Technology. (1994)
18. Garrett, J.J.: Ajax: A New Approach to Web Applications. (2005)
19. Want, R., Pering, T., Danneels, G., Kumar, M., Sundar, M., Light, J.: The personal server: Changing the way we think about ubiquitous computing. In: Proceedings of the 4th international conference on Ubiquitous Computing (UbiComp'02), Springer-Verlag (2002) 194–209
20. Fuggetta, A., Picco, G.P., Vigna, G.: Understanding code mobility. IEEE Transactions on Software Engineering **24**(5) (1998) 342–361
21. Kindberg, T., Barton, J., Morgan, J., Becker, G., Caswell, D., Debaty, P., Gopal, G., Frid, M., Krishnan, V., Morris, H., Schettino, J., Serra, B., Spasojevic, M.: People, places, things: Web presence for the real world. In: Proceedings of the 3rd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'00), IEEE (2000) 19
22. Ravi, N., Stern, P., Desai, N., Iftode, L.: Accessing ubiquitous services using smart phones. In: Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom'05). (2005) 383–393
23. Hartwig, S., Stromann, J.P., Resch, P.: Wireless microservers. IEEE Pervasive Computing **1**(2) (2002) 58–66
24. Nokia: Mobile Web Server. `http://wiki.opensource.nokia.com/projects/Mobile_Web_Server` (2008)
25. Ponnekanti, S., Lee, B., Fox, A., Hanrahan, P., Winograd, T.: ICrafter: A service framework for ubiquitous computing environments. In: Proceedings of the 3rd International Conference on Ubiquitous Computing (UbiComp'01). Volume 2201 of Lecture Notes in Computer Science., Springer-Verlag (2001) 56–75
26. LaPlant, B., Trewin, S., Zimmermann, G., Vanderheiden, G.: The universal remote console: A universal access bus for pervasive computing. IEEE Pervasive Computing **3**(1) (2004) 76–80
27. Nichols, J., Myers, B., Higgins, M., Hughes, J., Harris, T., Rosenfeld, R., Pignol, M.: Generating remote control interfaces for complex appliances. In: Proceedings of the 15th annual ACM Symposium on User Interface Software and Technology (UIST'02). Volume 2201., ACM Press (2002) 161–170