A dynamic lightweight Platform for Ad-hoc Infrastructures

Andreas Frei, Gustavo Alonso
Department of Computer Science
Swiss Federal Institute of Technology Zürich
CH-8092 Zürich, Switzerland
frei, alonso@inf.ethz.ch

Abstract

Mobile devices like PDAs or mobile phones have become widespread. Similarly, network functionality like GSM, Bluetooth, or WLAN has become standard. Nevertheless, not many applications take mobility into account. The reason may be due to the lack of an appropriate platform supporting networking, discovery, activation of new behavior, etc. as it is used in enterprise applications. Hence, in this paper we propose a new dynamic lightweight platform - Jadabs - consisting of a small footprint core layer and a modularized pluggable distributed infrastructure. Jadabs uses the service oriented architecture of OSGi and combines it with a dynamic Aspect Oriented Programming approach. In the paper we show how to extend OSGi with dynamic AOP functionality and made the latter to work in a container. The resulting platform is about 300 KBytes in size.

1. Introduction

Adaptability to changes (bandwidth, peers, network conditions, services) and the constant change from one environment to another (different base stations, different domains, ad hoc networks, etc.) are some of the biggest challenges that mobile devices face today. For the most part, current devices for mobile computing lack the ability to adapt, thereby restricting the flexibility of the applications.

To solve this problem, we propose a new architecture for small devices that borrows concepts already known from enterprise environments. In such environments, J2EE containers such as JBoss or Weblogic play an important role in supporting the deployment of Java components. In addition, Aspect Oriented Programming (AOP) techniques [5, 22] can be used to dynamically extend applications within such containers as shown by systems that combine containers with dynamic AOP [32]. Yet, all of these solutions lack the ability to be run on small devices. In order to provide a dynamic architecture for resource constrained devices, we created a dynamic lightweight container offering enterprise level capabilities for such devices. Figure 1a shows the architecture with a local application and a distributed application in Figure 1 b. As shown in the figure, applications are assembled of other components residing in the container, which is why we call our architecture Java ADhoc Application BootStrap – *Jadabs* [21].

With this architecture we are able to dynamically assemble an application out of components. These components can then later on be adapted at runtime as need dictates. For example, mobile devices have an increasing number of network interfaces such as GSM, Bluetooth, or Wireless LAN. With our infrastructure, an application could be automatically redirected to use the best possible network according to different criteria. In the paper we show how to use the Jadabs platform to build a multi-hop, multi-network bridge across several devices. This example illustrates the flexibility of the platform and the advantages of the form of adaptation we propose. The contribution of the paper is that the infrastructure we propose goes beyond the usual applications for small devices. This is the result of the combination of dynamic AOP and a container architecture, as we are able to do arbitrary runtime changes to adapt the behavior of devices and whole infrastructures with acceptable overhead. Our proposed peer to peer messaging system built on top of the core container can be regarded as a further example of the potential of the Jadabs platform.

The paper is organized as follows. In section 2, we describe Jadabs and implementation of the core layer. Section 3 describes the distributed infrastructure. A messenger scenario is described in section 4. Section 5 compares our approach to related work. Section 6 concludes the paper.

The work presented in this paper was supported (in part) 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 grant number 5005-67322.

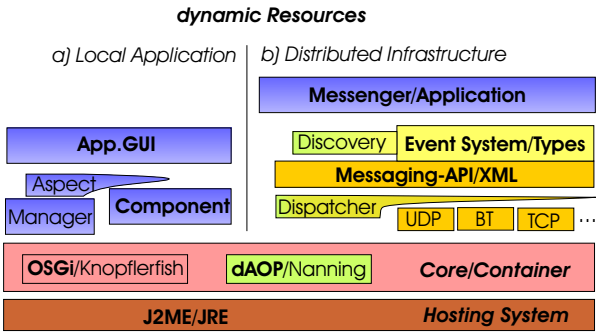


Figure 1. Jadabs Layers and dynamic Resources

2. System Architecture

2.1. Requirements

Our architecture must meet the following criteria:

- **Reduced footprint**. The infrastructure must be small enough to fit into mobile devices without impairing the actual applications running on those devices. At this stage we are not aiming for micro-devices such as those found in sensor networks or wearable networks [19] but for devices in the range of PDAs.
- **Dynamic adaptation**. Adaptation must happen transparently. It must be dynamic (extensions can be acquired or discarded at any point in time) and it must not require stopping the application and restarting. Manual configuration or user driven installation of components should also be avoided.
- **Flexible architecture**. In our model, components or new services are provided by the computing and network environment in which a node resides. For this to be realistic, there cannot be any constraints on who can provide such components or services that help a mobile device to adapt. Thus, adaptations can be provided by peers (any other node in the same network), base stations, or specialized servers. We do not distinguish either what type of network is being used (whether wireless with a base station, ad hoc networks, or combinations thereof).

2.2. Design Overview

Figure 1 shows our architecture. On the base layer we currently support J2ME and the normal Java Runtime Environment. J2ME is a subset of the JRE and can be further divided into the Connected Device Configuration (CDC) [38] and the more limited CLDC [39] version. Currently, we target the CDC as the CLDC is even more restrictive regarding the functionality of the Java virtual machine.

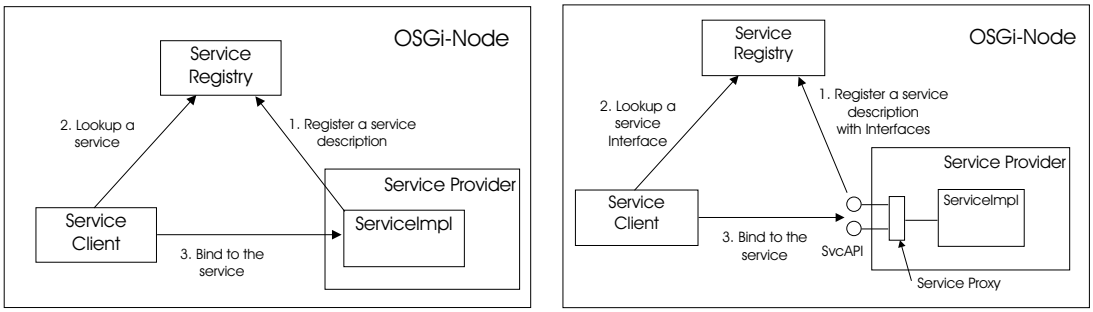
The architecture of Jadabs has been subdivided into the core layer and the dynamic resources layer. The core layer acts as container for the dynamic resources. The dynamic resource layer contains applications which are built out of components. This layer is able to load and unload components at runtime without disrupting the applications.

The core layer fulfills the first two requirements, **reduced footprint** and **dynamic adaptation**. To do so we combine two different approaches: Service Oriented Architecture (SOA) and dynamic AOP (dAOP). First, SOA (Figure 2a) allows to develop components independent of each other as follows. A *Service Provider* component registers the visible part of its component, the *Service*, in its local registry (1). A *Service Client* component is then able to lookup the registered service (2) and bind to the *Service Implementation* (3). This is similar to the service registration and lookup mechanism of Jini, except that instead of the remote Jini registry, the service is registered locally. SOA significantly improves the decoupling of components but it does not fully solve *dynamic adaptation*. A service which is going to be adapted with new functionality has to be stopped and restarted. This leads to state loss. Another disadvantage is a continuous check at the service client side to make sure the service is still available before a call is made.

To deal with adaptation, we use dynamic Aspect Oriented Programming (dAOP). Runtime adaptation occurs through the dynamic inclusion of aspects within the code of an application using, e.g., *join points* such as *before, after, or around* method. These join points cause the dynamically acquired extension to be executed before executing, after executing or instead of, a given method.

The resulting core layer is thus very powerful from a design point of view as it allows service oriented design and the dynamic adaptation of the services. In many applications, the core layer is all that is needed as it already provides a very powerful platform for mobile applications. The only addition needed is a *manager component* (Figure 1a).

When the adaptation involves messaging and coordination between a number of devices, additional infrastructure is needed. Jadabs uses the core layer to implement the dynamic resource layer. This layer can contain arbitrary functionality. It is this layer that helps Jadabs meet the **flexible architecture** requirement as it can be used to implement (or acquire dynamically) any infrastructure that may be needed. In this paper, we discuss how to use this layer to implement a peer-to-peer messaging platform that is JXTA compatible but modular, much leaner, and adaptable at run time. In previous work, we have shown how to use this layer to implement an event based messaging system [13].



(a) normal (b) with Proxy

Figure 2. Service Oriented Architecture (SOA)

2.3. Implementation

Combining SOA and dAOP into a reduced footprint container has been the main challenge of our implementation effort. Several solutions exist for either of these two approaches. We chose the combination which allowed us to use the core layer on many different platforms. As a common platform that works in devices ranging from PDAs to desktop computers we chose the J2ME/CDC, which offers a compromise between the devices that can be used and the limitations it imposes on SOA and dAOP. In addition, neither the SOA nor the dAOP solutions adopted were designed to be used together. Thus, they also had to be changed to build Jadabs.

As a foundation for the Service Oriented Architecture we use the API defined by the Open Service Gateway Infrastructure (OSGi) Alliance [29]. Several implementations are available [10, 20, 34]. Of these, we use Knopflerfish [24] due to its small footprint of about 200 KBytes. In OSGi the deployment of components is packaged into so called bundles which are stored in jar files. These bundles are activated in the OSGi framework and services can be registered as illustrated in Figure 2a.

Aspect Oriented Programming

We chose as a dynamic AOP solution Nanning [28]. The dynamic AOP capabilities are primarily restricted by the platform and resource limitations of small devices. Several solutions exists [4, 31, 40] but none of them have been used in a lightweight container for small devices. Nanning has been chosen due to its small footprint and easy adaptability in a small container.

Nanning is a dAOP solution with a proxy oriented approach. Instead of calling a service directly, a representative or proxy is called which is then responsible for forwarding the call to the real service implementation. Figure 2b shows the registration of the service by the provider. The dynamic proxy concept is available since Java 1.3 and therefore also for J2ME/CDC VMs. By using this API, a proxy can be generated to the given interfaces and the actual instance.

Every bundle in OSGi is loaded with its own classloader. This allows a removal of the bundle at a later stage if requested. In Nanning the proxies are created with the system classloader which has access to the interfaces and implementations. This would require to add the aspect bundles to the systems classpath. Furthermore, the aspect classes could not be removed again because classes can only be removed when their classloader is removed.

Nanning has therefore been adapted to create proxies with the bundles classloader. The implementation of the OSGi extension (see below) then passes the bundles classloader on to the adapted Nanning library.

Service Oriented Architecture

As the OSGi API does not support the integration of AOP functionality, we use a slightly modified OSGi API. Algorithm 1 shows the *AOPContext* interface which is an extension to the OSGi *BundleContext*. The *AOPServiceRegistration* and *AOPService* interfaces gathered through the *AOPContext* allow to query the registration and the service. The implementation then uses the underlaying AOP engine for crosscutting concerns.

Algorithm 1 dAOP/OSGi extension

```
1  interface AOPContext
2  {
3     AOPServiceRegistration registerAOPService(
4        Class clazz, Object service, Dictionary properties);
5
6     AOPService getAOPService(ServiceReference reference);
7  }
```

The AOP engine makes use of proxies so that crosscuts can be dispatched inside the proxies. Algorithm 2 shows how services are registered in OSGi normally and with our adapted interface.

Algorithm 2 Service Registration w/o AOP

```
1  void start(BundleContext context) throws Exception
2  {
3     eventserviceimpl = new EventServiceImpl();
4
5     // register Eventsystem as an AOP Service
6     eventsvcreg = ((AOPContext) context).registerAOPService(
7        IEventService.class,
8        eventserviceimpl, null);
9
10    // register Eventsystem as a normal Service
11    context.registerService(
12       EventServiceImpl.class.getName(),
13       eventserviceimpl, null);
14 }
```

The developer should be aware of two design principles when designing such dynamic services. First, the service should only be called on the interface level. This requires to register the service only with the interfaces (line 7) compared to the possible class registration example (line 12). However, if the implementation is called directly, it will circumvent the proxy which is then not able to intercept the method call. This leads to the second design principle, the separation of interface and implementation, which is actually enforced through our dAOP approach.

At the core layer, services can be registered and unregistered as a way to adapt to changes in the environment. However, the programmer is responsible for keeping track of what services have been added and removed. Otherwise clients may be left in an improper state with dangling pointers to unavailable services. Similarly, services my change or be replaced by others, at which point clients also need to be adapted to be able to use the new service. This is known as the *stale references* problem [29].

OSGi proposes a solution to the stale references problem by using the *Service Tracker* and *Service Listener* classes. They allow to listen for service events which are generated once a service is registered and unregistered. The observer pattern behind this mechanism must be implemented by the bundle developers themselves.

Another solution to the stale references problem has been proposed as part of the *Service Binder* [18], an automatic service dependency management. The registering and the removal of services is externalized into an XML descriptor. The descriptor defines the methods which have to be called to register a service once it is available or unregister when the service is removed. This observer concept is very similar to the *Service Tracker*. The *Service Binder* extends this functionality by a cardinality checker. This allows to track the life cycle of the bundle depending on the numbers of required services. This solution requires a listener registration mechanism in every service client and XML files which have to be maintained.

In Jadabs, we use the Service Binder approach. However, in the original Service Binder approach, the service has to provide observer methods. With our proxy approach the observer pattern is hidden inside the proxy and transparent to the client. Thus, clients do not need to be aware of service changes since this is done automatically inside the proxy. For instance, once a service is removed by the provider, the reference in the proxy is also removed. A call to that service will cause the proxy to either return a default value or throw an exception. The default behavior has to be specified when the proxy is generated and can be changed at a later time. To specify the default behavior we use meta tags inside the interfaces. The meta tags are queried when the proxy is created.

2.4. Benchmark

As a first benchmark we measured the performance of the dynamic lightweight container. We analyzed the overhead of the dynamic AOP approach in different platforms: Java Runtime Environment (JRE), the J2ME/CDC CVM, and on two different machines, a laptop and a PDA.

The laptop we used was an IBM Thinkpad A31 with 1.8 GHz running under Fedora Core 1. The PDA was an iPAQ H3970, 400 MHz with familiar Linux 0.7.1. As JRE we used version 1.4.1 on the laptop and on the iPAQ the Blackdown 1.3.1. The CVM is the Sun implementation compiled with JRE 1.3.1 for the laptop and cross-compiled for the iPAQ.

In the proxy benchmark we measured the overhead of method calls with varying types and numbers of arguments. One run contains 16 methods with an empty body. The numbers presented in table 1 show the average of 1000 runs on the iPAQ and 10'000 runs on the laptop.

VM/devices	normal	proxy	after	around	before	avg
cvm/ipaq	0.16	8.08	8.47	7.55	8.52	8.18
cvm/lap	0.02	0.57	0.63	0.56	0.62	0.60
jre/ipaq	0.45	13.5	15.5	13.6	15.8	15.0
jre/lap	0.01	0.08	0.09	0.08	0.09	0.09

Table 1. AOP overhead of 16 method calls in milliseconds

In a first measurement (*normal*) we measured the time of one run under normal circumstances without AOP or proxy. In the second measurement the *proxy* is switched on and every method call goes through the proxy. In the following measurements an aspect has been inserted on every method to measure the *after*, *around*, and *before* join point overhead with the average in the last column.

As can be seen, switching on the proxy has a significant impact on the execution time of empty methods. The overhead of using AOP in the proxies varies: cvm/ipaq (1.3%), cvm/lap (5.1%), jre/ipaq (10.8%), and jre/lap (5.7%).

The measurements show also that the Blackdown VM on the iPAQ is significantly slower then the CVM from Sun. When looking at the memory usage at runtime, the CVM uses about 5 MByte whereas the Blackdown uses over 20 MByte for running these tests. The best solution is therefore to use the JRE 1.4.1 on the laptop and the CVM on the iPAQ.

Even though the proxy overhead is significant for empty methods, the overhead relative to the total time of the method call decreases the longer it takes to execute the method. This is especially true for proxies over interfaces. Usually interfaces hide the implementation with more than one method calls. As we are going to show in the next section, the overhead imposed by the dynamic container are acceptable when using it to build a distributed infrastructure.

3. Distributed Infrastructure

The proposed dynamic lightweight container that is implemented at the core can now be used to build applications out of different components. In this section we give an example of how to use the Jadabs core to build a distributed infrastructure. Our main target is the distributed mobile infrastructure, as shown in Figure 1b. It is a peer-to-peer messaging infrastructure that borrows from JXTA [6, 23]. Unlike JXTA, our implementation is modular so that different modules can be acquired dynamically as needed.

3.1. Message Layer

The first level in the the distributed resources layer is a basic messaging system that will allow nodes to communicate with each other through a simple interface (Figure 1). The interface is related to the JXTA's JXME subproject which implements a proxy-less messaging layer for small devices running on J2ME/CLDC. The message layer is therefore simple enough to run on small devices. As this messaging system is very general and independent of a specific communication protocol, it can be used to communicate in an ad hoc manner also with platforms other than Java. To ensure wide use, we maintain compatibility with the JXTA message specification in [1].

UDP, TCP, Bluetooth

An increasing amount of devices have several network interfaces, like mobile phones with GSM and Bluetooth or PDAs with Bluetooth and in some cases also WLAN. Unfortunately applications are in many cases built for one type of network and ignore the advantage to use the same application on another network. Our architecture allows to plugin new extensions at runtime and configure the devices depending on which network is available.

In our case, we use UDP, TCP, and Bluetooth. They reflect the properties of the different infrastructural facilities. UDP is used for devices with W-LAN, Bluetooth for devices supporting only the Bluetooth network layer, and TCP to connect gateways to the Internet.

Each one of these extensions is independent of the other and packaged as an OSGi bundle. The messaging layer itself is registered as a dAOP service allowing the concrete network implementation to be plugged in as an aspect. Therefore, the bundle is not required until, e.g., a Bluetooth network is available thus keeping free the memory for other bundles. Along with freeing memory the processor utilization is optimized as idle threads handling the network layer are not running.

Once a new network layer is available, for example by inserting a W-LAN card into a PDA, the UDP implementation can be set up. The UDP implementation can be deployed from a locally available copy or it can be obtained from an external device (a peer, a base station, etc.). Upper layers like the event system (see below) allow to deploy such a bundle over an already established connection.

Dispatcher

In a system where different networks are available, for example WLAN and Bluetooth, a dispatcher is needed to send the message to the appropriate network. For example a node has UDP as an initial network interface started. Once a new network interface is available, for example Bluetooth, the node may now receive messages over two different network layers. A message which is sent subsequently needs to be routed by the framework to the appropriate network. To do so, we introduce a dispatcher bundle which is able to send messages between different networks.

The dispatcher bundle can be deployed and activated in the node. By using the dynamic AOP functionality the dispatcher hooks into the already running UDP implementation (Figure 1) and dispatches the messages. The dispatcher takes over the role of a routing component which is able to find the responsible network implementation for sending the message. The Bluetooth implementation can refer to a list of other connected Bluetooth devices to decide if it is responsible for sending the message. The UDP implementation can be chosen as default in case the message cannot be sent over Bluetooth.

The functionality of the dispatcher could be adapted to further requirements. A quality of service component may decide for example to use the UDP implementation for wireless scenarios and the TCP implementation in LAN infrastructures.

3.2. Event System Layer

The message system can thus operate over a variety of network interfaces (UDP, TCP, Bluetooth) but can only exchange simple messages. Thus, we have extended the distributed resource layer with an event system. This additional bundle adds a type system on top of the messaging and allows to publish and subscribe to events sent over the messaging system. The publish and subscribe model is based on the concepts of [9, 11, 17] and the type- and attribute-based subscription model of [3, 30]. With this simple API we are able to take advantage of a very powerful distributed messaging framework.

Whereas a small device does not have to implement all features of an event system, more powerful nodes can take over the distribution and subscription. Therefore, a small device only requires the possibility to understand new types, subscribe and publish events. More enhanced nodes may then take over the task of storing such events for later usage or routing subscriptions.

The event system has been designed as a dynamic AOP service and is registered at activation time through the proxy. This allows to change the event system API later on depending on new requirements.

Discovery

In ad hoc environments an important ability is the possibility to discover new devices and services in the environment. Many discovery technologies have emerged for different platforms and environments like Jini [2], UPnP [41], and JXTA [1]. Jini and UPnP are supported through OSGi service APIs.

In our implementation we use JXTA's peer to peer group. Our main goal is here to be backward compatible to JXTA and be able to run such a discovery mechanism inside the event system. By extending the event system API with a discovery API we can take advantage of the simple API of the event system.

Instead of the method *publish(event)* a *publish(event, leasetime)* extends the already deployed interface. This is only a slight modification of the already known publish method. The extended interface allows to resend the event for the time specified as *leasetime*.

3.3. Discussion

The proposed distributed infrastructure implementation strongly relates to the JXTA specification. Our main goal here is not having to implement a new distributed infrastructure. It should be compatible on different layers with the chosen JXTA's peer to peer concept. By implementing different parts of JXTA on top of the Jadabs core, we are able to insert and extend the modules as needed. This way resource constrained devices, not capable of running a 2 MByte JXTA infrastructure, may still participate in such an infrastructure by using only part of it.

So far we concentrated on the JXTA peer to peer concept. With the dynamic behavior of our proposed infrastructure we would be able to add additional infrastructure support. For example, instead of the JXTA messaging system, the UPnP messaging system could be used for UPnP devices. Similarly, any new protocols can be added or removed dynamically and transparently to the other components in the system.

We consider the event system as a key component in a distributed environment. With its simple interface, a broad range of application needs can be solved. For example the peer group mechanism from JXTA can be easily implemented with the proposed event system. The proposed discovery interface looks very similar to the event system interface. Furthermore, we use the event system in a remote manager component to deploy components.

3.4. Benchmark

In the following benchmark we measured the overhead of having two layers calling methods through proxies (Figure 3a). As a second benchmark we looked at the production overhead of the full system when sending events between two nodes (Figure 3b). The test setup is similar to the container benchmark in the previous section.

Framework Benchmark

In this benchmark we measured the proxy overhead of the Jadabs architecture with the event and messaging systems (Figure 3a). We measured the framework by publishing an event and stopping the timer before the message is sent through the UDP connection. This way we can measure the overhead of going through two proxies.

The measurement was repeated 10'000 times with a standard deviation of less than 1.5%.

VM/device	normal	proxy	overhead
jre/lap	0.232 ms	0.247 ms	6.5%
cvm/ipaq	15.8 ms	17.0 ms	7.3%

Table 2. Proxy overhead for framework

The numbers in Table 2 show an overhead of 6.5 % for the JRE/Laptop and 7.3% for the CVM/iPAQ. This is considerably smaller than the much higher relative overhead observed in the container benchmark from section 2. This

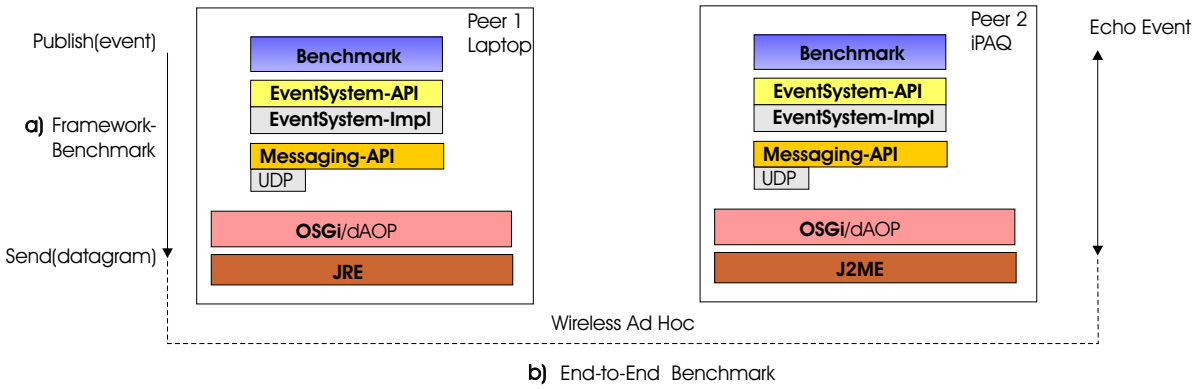


Figure 3. Benchmark Scenarios

is because in that benchmark we were calling empty extensions. As soon as these extensions incorporate useful functionality (as in this second experiment) the cost of running the extension hides most of the overhead introduced by Jadabs.

End-to-End Benchmark

In the scenario benchmark we measured the round trip time of an event from a laptop to an iPAQ. Figure 3b shows the steps involved. An event is published by a benchmark service, *Benchmark*. The event is sent through a wireless UDP connection in ad hoc mode to the iPAQ. Once received on the iPAQ, the event is extracted from the message and the subscribed benchmark service is notified. Upon notification the event is published again and sent back to the original publisher where the timer is stopped. The results show the average round trip time of 10'000 published events (Table 3).

	normal	proxy	overhead
Lap - iPAQ	40.46 ms	41.45 ms	2.5%

Table 3. Proxy overhead for real scenario

The measurement shows an overhead of 2.5% when using the proxy approach, as opposed to the static compiled benchmark (normal). With this, we can conclude that the observed overhead introduced by Jadabs is negligible compared with the design advantages it provides. Although the absolute overhead is high, when the cost of executing extensions is factored in, the overhead drops dramatically. Furthermore, when those extensions involve messages over a network, the overhead of Jadabs is completely hidden behind the cost of the functionality added to the application.

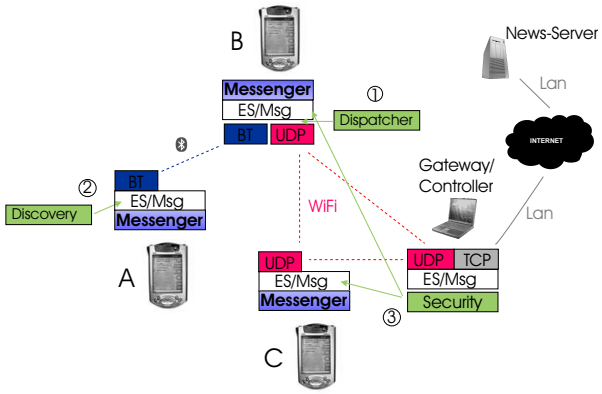


Figure 4. Messenger Scenario; 1) dispatching Bluetooth layer to UDP layer, 2) discovering the gateway, 3) forcing encryption in WLAN

4. Application

To illustrate its potential, we have used the complete Jadabs platform described above to build a messenger system (Figure 4). With this messenger application a user can communicate with other users over different networks in a secure way. Each messenger application on the PDAs starts off with the most basic functionality needed to send and receive messages. New behavior like dispatching, discovery, and security is transparently and dynamically added.

We used iPAQs running Linux and on top of J2ME/CDC a basic Jadabs platform. The basic Jadabs deployment depends on the PDAs hardware capabilities. PDA A has for example only a Bluetooth adapter, C only a Wireless Card, and B both. Through the controller we are now able to extend node B with a dispatching functionality. A bundle containing the dispatcher functionality is therefore sent through

the UDP connection from the controller to B where it is activated. Node B is now able to bridge the two wireless networks, W-LAN and Bluetooth. We used the UDP layer in the W-LAN ad hoc network since with TCP too many package collisions occurred. This allows us to extend node B and C at once with a security extension over multicast. When using TCP, the same extension would have to be sent twice taking twice that much bandwidth.

The reliability of events has been solved inside the event system through well know mechanisms. We discovered that UDP is very reliable when not too many nodes are communicating. Thus, message reliability is a feature that can be switched off when only few nodes are communicating.

After extending node B with the dispatching and security functionality, node A is now able to send messages to the Internet. A message sent by A is received over Bluetooth in B, encrypted and dispatched to the W-LAN. Finally, at the gateway the message can be sent out through the normal TCP connection.

Over W-LAN and Bluetooth the controller is now able to install a discovery extension in node A. This allows A to discover news-servers. Once node A reconnects to another network it may be able to reach the news-server with this discovery extension again.

In this scenario we use a central controller to send extensions. We are now in progress of extending Jadabs with a discovery functionality which will allow us to do automatic configuration of such infrastructures.

5. Related Work

We first give an overview of component containers and continue with distributed infrastructures.

The trend to build component containers smaller than J2EE has already started in many projects. By building up a container on a micro kernel the middleware can get much smaller and has to activate only those components needed. Merlin [25], JBoss [16], and Spring [37] are going in this direction. Even though their micro kernel is getting smaller, in case of JBoss about 650 KBytes, and Spring 300 KBytes they still rely on services available only in an infrastructure network like JNDI in JBoss or heavy use of XML in Merlin and J2EE services in Spring. Spring includes a similar dynamic AOP approach as proposed in Jadabs.

Micro-kernel architectures which also fit into smaller devices are available from the Open Service Gateway infrastructure OSGi [29], or Fractal [12] from ObjectWeb. Fractal and OSGi are similar in the service binding concept whereas OSGi defines more services for an ad hoc service infrastructure like Jini and UPnP. OSGi has been chosen due to its broad acceptance in the industry as well as the open source community.

ReMMoC [14, 15] is a reconfigurable reflective middleware to support mobile application development. It uses OpenCOM [8] as its underlying component technology. OpenCOM is implemented in C++ and available for a few platforms. An upcoming new release should support more platforms based on XPCOM [27]. ReMMoC is a similar platform for mobile devices as proposed in this paper, but based on C++. Whereas Java has become a de facto standard, no dynamic lightweight platform exists to such a large extend as proposed in this paper. By using Java a huge resource of already existing applications and components can be run on top of the core layer. Furthermore, we propose a distributed infrastructure based on JXTA which is a peer to peer protocol independent of a network technology or an operating system.

Prism [26] is a software architecture which provides programming language-level constructs for implementing components, connectors, configurations and events. The middleware is assembled before deployment according to the required components. So far no runtime changes can be performed as are required in a dynamic reconfigurable environment.

Midas [32, 33] is a dynamic middleware which combines the dynamic AOP implementation PROSE [31] with the Jini infrastructure. By using Jini the infrastructure is bound to Java environments only . Additionally, it requires a centralized lookup server which is not always available in an ad hoc environment. Furthermore, PROSE requires a Java Virtual Machine supporting the JVM Debugger Interface which is not available for all Virtual Machines in small devices. In our Jadabs architecture we use the proxy concept which is available in the J2ME/CDC implementation. As our distributed infrastructure depends on XML protocols we are not restricted to Java environments.

GaiaOS [7, 36] is a component based meta-operating system, that runs on top of existing systems, such as Windows2000, WindowsCE, and Solaris. GaiaOS is used in a middleware infrastructure for active spaces [35]. The system focuses on the management of active space resources and provides location, context and event services. In Jadabs we focused on the adaptation of already running systems and provide extension possibilities on all dAOP registered interfaces. This extends the configuration and extension capabilities of GaiaOS. Furthermore, we use a peer to peer messaging infrastructure without the need of a centralized naming service.

JXTA [1, 6, 23] is a peer to peer infrastructure built for the Internet infrastructure. Its messaging and peer group infrastructure is based on XML where nodes communicate with XML messages. This allows the implementation of such a peer to peer architecture on any device and platform. Implementations are already available for many platforms like Java, Python, Perl, Ruby, C. As JXTA is too big to fit

on a mobile device, a subproject, the JXTA-JXME, is implementing the messaging infrastructure for mobile devices based on J2ME/CLDC. The full JXTA version lacks in two points. First, there is no mechanism to load new services into a device. In a dynamic environment new services need to be loaded on a device and unloaded again when they are not needed anymore. By using OSGi in our architecture we are able to load and unload such dynamic resources. Second, JXTA has grown in its functionality and requires more than 2 MByte since it comes with everything included. In Jadabs we require only a small micro kernel of about 300 KBytes and additional packages have an average size of 10 KBytes. Thus, new functionality can be loaded and activated when needed. Our concept would even allow to use the full JXTA as a bundle on a more powerful node. By taking over the peer to peer concept and messaging specification of JXTA we are able to connect our dynamic infrastructure to JXTA nodes.

6. Conclusion

With Jadabs [21] we propose a dynamic lightweight architecture for resource constrained devices. By combining the service oriented architecture of Knopflerfish, an OSGi implementation, and a dynamic AOP approach, Nanning, we can build a dynamic lightweight container. Jadabs is a small container solution for small devices like PDAs up to desktop machines with a footprint of about 300 KBytes. By supporting a small container for such a wide range of devices, we are able to provide a dynamic ad hoc infrastructure which can be configured depending on the capabilities of the device.

The proposed solution uses proxies to extend the service oriented architecture of OSGi by the dynamic AOP concept. As our benchmarks indicate the overhead of 2.5% in real distributed scenarios can be regarded as negligible. By combining an SOA and dynamic AOP approach we can also solve the stale reference problem in SOA.

In this paper we showed how to use Jadabs to build a distributed peer to peer infrastructure similar to JXTA. As JXTA is too big for small devices our goal was to refactor their monolithic architecture that are pluggable services downloaded and activated when required. This leads to smaller distributed infrastructure which can be run on small devices. The dynamic lightweight container is therefore the core for a distributed architecture. A message layer concurrent to JXTA's message specification suffice as a first layer for communication. Different implementations are supported according to the device possibilities like IEEE 802.11b, LAN or Bluetooth. As a second layer we propose the simple event system interface. Both layers are registered as dynamic AOP services. This allows to extend or replace the implementation. In case of the message layer a dispatching service can be integrated and for the event system a discovery bundle can be plugged in when required. Even though we adapt the distributed infrastructure with the dynamic AOP concept, a local application can use the same concept.

Currently, we are in the progress of extending the discovery mechanism for autonomous configuration. A .NET implementation is under way and features a working messaging infrastructure. As the loading and unloading of services in .NET is different, another solution is being researched. To show the feasibility of our approach for even smaller devices like mobile phones we are working on a J2ME/CLDC solution.

References

[1] JXTA v2.0 Protocols Specification, Oct. 2003. http://spec.jxta.org/nonav/v1.0/docbook/-JXTAProtocols.html.

[2] K. Arnold, A. Wollrath, B. O'Sullivan, R. Scheifler, and J. Waldo. *The Jini Specification*. Addison-Wesley, Reading, MA, USA, 1999.

[3] S. Baehni, P. T. Eugster, and R. Guerraoui. OS Support for P2P Programming: a Case for TPS. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, Vienna, Austria, July 2002.

[4] Bob Lee. dynaop, 2004. https://dynaop.dev.java.net.

[5] J. Bonér and A. Vasseur. AspectWerkz. http://aspectwerkz.codehaus.org.

[6] D. Brookshier, D. Govoni, and N. Krishnan. *JXTA: Java P2P Programming*. SAMS, Mar. 2002.

[7] R. Cerqueira, C. K. Hess, M. Román, and R. H. Campbell. Gaia: A Development Infrastructure for Active Spaces. In *Workshop on Application Models and Programming Tools for Ubiquitous Computing (held in conjunction with the UBICOMP 2001)*, Sept. 2001.

[8] M. Clarke, G. Blair, G. Coulson, and N.Parlavantzas. An Efficient Component Model for the Construction of Adaptive Middleware. In *Proceedings of Middleware 2001*, Nov. 2001.

[9] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9), 2001.

[10] Eclipse. Equinox, 2004. http://www.eclipse.org/equinox/.

[11] L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann. Engineering event-based systems with scopes. In B. Magnusson, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *LNCS*, pages 309–333, Malaga, Spain, June 2002. Springer-Verlag.

[12] Fractal. ObjectWeb, Open Source Middleware, 2004. http://fractal.objectweb.org/.

[13] A. Frei, A. Popovici, and G. Alonso. Eventizing Applications in an Adaptive Middleware Platform. Technical Report TR 451, Swiss Federal Institute of Techonolgy Zurich, Mar. 2004.

[14] P. Grace, G. S. Blair, and S. Samuel. Middleware Awareness in Mobile Computing. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03)*.
[15] P. Grance, G. Blair, and S. Samuel. Interoperation with Servies in a Mobile Environment. Technical Report MPG-03-01, Lancaster University, 2003.
[16] J. Group. Jboss. http://www.jboss.org.
[17] M. Haupt, M. Mezini, M. Cilia, and A. P. Buchmann. Towards Event-Based Aspect-Oriented Runtime Environments. Technical Report TUD-ST-2002-01, Software Technology Group, Darmstadt University of Technology, Alexanderstrasse 10, 64289 Darmstadt, Germany, 2002.
[18] H.Cervantes and Richard S. Hall. Automating Service Dependency Management in a Service-Oriented Component Model. In *Proceedings of the 6th Workshop on Component-Based Software Engineering (CBSE)*, May 2003.
[19] J. Hill, M. Horton, R. Kling, and L. Krishnamurthy. The platforms enabling wireless sensor networks. *Commun. ACM*, 47(6):41–46, 2004.
[20] IBM. Service Management Framework, 2004. http://www-306.ibm.com/software/wireless/smf/.
[21] Jadabs. Dynamic Lightweight Infrastructure for Small Devices. http://jadabs.berlios.de.
[22] JBoss. Aspect Oriented Programming, 2004. http://www.jboss.org/developers/projects/jboss/aop.
[23] JXTA Project. http://www.jxta.org.
[24] Knopflerfish OSGi. http://www.knopflerfish.org.
[25] Merlin. The Apache Avalon Project, 2004. http://avalon.apache.org/merlin.
[26] M. Mikic-Rakic and N. Medvidovic. Adaptable Architectural Middleware for Programming-in-the-Small-and-Many. In *Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference*, pages 455–473. Springer-Verlag, June 2003.
[27] Mozilla . XPCOM. http://www.mozilla.org/projects/xpcom.
[28] Nanning Aspects. http://nanning.codehaus.org.
[29] OSGi, Open Service Gateway Initiative. *OSGi Service-Platform*. IOS Press, release 3 edition, Mar. 2003.
[30] P. Pietzuch and J. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, Vienna, Austria, July 2002.
[31] A. Popovici, G. Alonso, and T. Gross. Just In Time Aspects: Efficient Dynamic Weaving for Java. In *2nd Intl. Conf. on Aspect-Oriented Software Development, Boston, USA*, Mar. 2003.
[32] A. Popovici, G. Alonso, and T. Gross. Spontaneous Container Services. In *Proceedings of the 17th Europeean Conference for Object-Oriented Programming*, pages 29–53. Springer-Verlag, July 2003.
[33] A. Popovici, A. Frei, and G. Alonso. A proactive middleware platform for mobile computing. In *Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference*, pages 455–473. Springer-Verlag, June 2003.
[34] Richard S. Hall. Oscar, An OSGi framework implementation, 2004. http://oscar-osgi.sourceforge.net.
[35] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. A Middleware Infrastructure for Active Spaces. 1(4):74–83, Oct. 2002.
[36] M. Román, F. Kon, and R. H. Campbell. Design and Implementation of Runtime Reflection in Communication Middleware: the dynamicTAO Case. In *Workshop on Middleware, ICDCS'99*, May 1999.
[37] Spring. Java/J2EE Application Framework, 2004. http://www.springframework.org.
[38] Sun Microsystems. J2ME Connected Device Configuration (CDC); JSR 36, JSR 218. http://java.sun.com/products/cdc.
[39] Sun Microsystems. J2ME Connected Limited Device Configuration (CLDC); JSR 30, JSR 139. http://java.sun.com/products/cldc.
[40] D. Suve, W. Vanderperren, and V. Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29. ACM Press, 2003.
[41] UPnP Device Architecture v1.0.1 Draft. http://www.upnp.org/resources.