A dynamic lightweight Architecture for Ad-hoc Infrastructures

Author(s):
Frei, Andreas; Alonso, Gustavo

Publication Date:
2004

Permanent Link:
https://doi.org/10.3929/ethz-a-006775292

Rights / License:
In Copyright - Non-Commercial Use Permitted
A dynamic lightweight Architecture 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, W-Lan has become standard. On the other hand not many real applications have taken the mobility into account. The reason maybe due to the lack of an appropriate platform for applications which could use generic basic functionalities like networking, discovery, activation of new behavior, etc. as it is used in enterprise applications. Jini and JXTA have not been able to be adapted to small devices as their resource usage goes beyond the availabilities of such devices. 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. We propose therefore to extend the OSGi API with dAOP functionality. On the other hand the dAOP solution has to be extended to be used in a container. The so obtained core layer is about 300 KBytes. The combination allows to plugin and adapt a distributed infrastructure to the devices resource capabilities like Bluetooth, GSM, or W-Lan. Compared to other distributed infrastructures everything is included in one package and usage of only parts of it are not possible.

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.) is one 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.

We propose therefore a new architecture for small devices which borrows concepts already known from enterprise environments. In such environments J2EE containers like JBoss or Weblogic play an important role. They allow to deploy new Java Enterprise Beans without restarting the whole container. These containers are now being extended with Aspect Oriented Programming (AOP) functionality [17, 6] to allow already running applications to be extended with new functionality. The combination of container architecture and dynamic AOP has already been used in infrastructure environments [29]. Yet, all of these solutions lack the possibility to be run on small devices. In order to create a dynamic architecture for resource constrained devices we followed the enterprise concepts and created a dynamic lightweight container for resource constraint devices. In addition, to be able to deploy new components in an ad hoc environment, a peer to peer architecture has been used for the small footprint container. Figure 1 shows the architecture with a local application on the left side and a distributed application on the right. Applications are assembled of other components residing in the container thus we call our architecture Java ADhoc Application BootStrap - Jadabs.

*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

By using this architecture we are now able to bootstrap an application depending on the devices capabilities. A node does not need anymore a full version of Jini or JXTA when only a subset is needed or none at all. For example JXTA consists of several layers for a peer to peer infrastructure. In an environment where only the basic layer is needed for messaging the memory used for the higher layers could be saved for other purpose. Once the environment changes for a mobile node a more enhanced distributed infrastructure may be needed. This new functionality can then be plugged in replacing the old one or adapting it.

An other example would be to change the connections of mobile devices at runtime. As such devices are getting more and more connected through different communication possibilities like GSM, Bluetooth, Wireless Lan (e.g., IEEE 802.11b). It remains an open issue for the user which connection has to be chosen. Sometimes a path over different technologies may have to be built in order to get the cheapest or fastest connection. Instead of sending a message over the GSM net it might be better to use a Bluetooth connection to a free GSM gateway.

The examples illustrate the flexibility of the platform and the advantages of the form of adaptation we propose. We are targeting an infrastructure which goes beyond usual applications for small devices. By including dynamic AOP in a container for mobile devices we are able to do runtime changes on the interface level. Such changes adapt the behavior of whole infrastructures with acceptable overhead even on small devices. Our proposed peer to peer implementation on top of this core container can be regarded as an example. The application itself can be designed taking advantage of the dynamic container.

The paper is organized as follows. In section 2 we describe the architecture of Jadabs and the implementation for the core layer including benchmarks. Section 3 describes the distributed infrastructure including benchmarks. A messenger scenario is described in section 4. Section 5 compares our approach to other work. Section 6 concludes the paper with a summary and future work.

2 System Architecture

Our architecture targets devices ranging from mobile phones over PDAs to Laptops. We must therefore meet the following:

- **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 but for devices in the range of PDAs, mobile phones, etc.

- **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 (much less restarting the application, manual configuration or user driven installation of components).
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 component 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).

Figure 1 shows our architecture which fulfills all three requirements. On the base layer we support currently J2ME and the normal Java Runtime Environment. J2ME is a subset of the JRE and can be further divided into the Connected Limited Device Configuration (CLDC) and the more enhanced CDC. Currently we target the CDC as the CLDC is even more restrictive regarding the functionality of the virtual machine.

To fulfill the three requirements the architecture has been subdivided into a core layer and the dynamic resources. 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.

![Diagram](a) normal (b) with Proxy

Figure 2: Service Oriented Architecture (SOA)

The core layer fulfills the first two requirements, reduced footprint and dynamic adaptation. Two design concepts for separation of concerns are used. First, the concept of a Service Oriented Architecture (SOA) (Figure 2a). This allows to develop components independent of each other. 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 concept is similar to the service registration and lookup mechanism of Jini. Except in SOA, it is done locally and no registry service is needed. A SOA significantly improves the decoupling of components. On the other hand 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. An other disadvantage is a continuous check at the service client side if the service is still available before a call is made. Therefore we propose a combination with a second concept known as dynamic Aspect Oriented Programming (dAOP). By using this combination we are now able to dynamically adapt the behavior of registered services at runtime. For example a method call on a service may now be crosscutted and do AOP advice calls like before, after, or around.

With the core layer as basis the dynamic resource layer is now capable of dynamically changing the application or the distributed infrastructure. Such change may be, e.g., loading a new component or adapting one with new behavior. Figure 1 shows two application stacks. On the left
side a local application is shown. Applications are increasingly built out of different third-party components. The core layer can therefore be very well used for only local applications to take advantage of the SOA and dAOP concepts. By including a manager component the local applications can then be configured. To use our dynamic container in a mobile environment a distributed infrastructure is required, Figure 1 right application stack. This solves our third requirement of a flexible architecture. The infrastructure itself may be adapted to the requirements of the current environment.

The two application stacks shown are just two examples. It is even possible to transform the application stack into a distributed infrastructure at runtime as proposed in [2].

2.1 Implementation

Combining SOA and dAOP into a reduced footprint container has been the main concern. Several solutions exist for either of these two concepts. We chose the combination which allowed to use the core layer on many different platforms. As a common platform ranging from small devices up to normal machines we chose at least the J2ME/CDC. This allows to run the higher layers on any such platform. On the other hand it limits the capabilities for SOA and dAOP. In the following we introduce the chosen solution. As neither the SOA nor the dAOP solution have been designed to be used together we explain the enhancements made.

Service Oriented Architecture (SOA)

As a Service Oriented Architecture we chose the API defined by the Open Service Gateway Infrastructure (OSGi) Alliance [25]. Several implementations are available [9, 16, 31] whereof we chose Knopferfish [19] due to its small footprint of about 200 KBytes. In OSGi the deployment of components is packaged into so called bundles which are jar files. These bundles are activated in the OSGi framework and services can be registered as illustrated in Figure 2a).

As the OSGi API does not support the integration of AOP functionality we introduce an additional 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 uses then the underlaying AOP concept for crosscutting concerns.

Algorithm 1 dAOP/OSGi extension

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>interface AOPContext</td>
</tr>
<tr>
<td>2</td>
<td>{</td>
</tr>
<tr>
<td>3</td>
<td>AOPServiceRegistration registerAOPService(</td>
</tr>
<tr>
<td>4</td>
<td>Class clazz, Object service, Dictionary properties);</td>
</tr>
<tr>
<td>5</td>
<td>}</td>
</tr>
<tr>
<td>6</td>
<td>AOPService getAOPService(ServiceReference reference);</td>
</tr>
<tr>
<td>7</td>
<td>}</td>
</tr>
</tbody>
</table>

The AOP concept chosen takes use of the proxy concept. This way crosscuts can be dispatched inside the proxies. The developer has therefore to be aware of two design principles when designing such dynamic services. Algorithm 2 shows how services are registered in OSGi normally and with our adapted interface. 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). Otherwise, the implementation is called directly and would circumvent the proxy which is then not able to intercept the method call. This leads to the second design principal which is already standard in component design especially for services to separate the interface from the implementation. The separation of interface and implementation is even forced through this AOP approach leading to better application designs.
Algorithm 2 Service Registration w/o AOP

```java
void start(BundleContext context) throws Exception {
    eventserviceimpl = new EventServiceImpl();
    // register Eventsystem as an AOP Service
    eventsvcreg = ((AOPContext) context).registerAOPService(IEventService.class, eventserviceimpl, null);
    // register Eventsystem as a normal Service
    context.registerService(EventServiceImpl.class.getName(), eventserviceimpl, null);
}
```

Aspect Oriented Programming

We chose as a dynamic AOP solution Nanning [24]. The dynamic AOP capabilities is primarily restricted by the platform and resource limitations of small devices. Several solutions exists [5, 28, 33] 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, the proxy, will be called and 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 later on to remove the bundle again. In Nanning the proxies are created with the system classloader knowing the interfaces and implementations. This would require that the aspect bundles are added to the systems classpath. Furthermore, the aspect classes could then not be removed again. The reason lies in the VM where classes can only be removed when the classloader is removed.

Nanning has therefore been adapted to create proxies with the bundles classloader. The implementation of the previously explained OSGi extension passes then the bundles classloader to adapted Nanning library.

2.2 Discussion of dynamic AOP in SOA

Our core layer of Jadabs is based on the service oriented architecture defined by OSGi. Services in this architecture can be registered and unregistered depending on the changes in the environment. The programmer of such services has therefore to keep track of changes in the environment. Services which are removed from the system may leave a client in an improper state with dangling pointers to unavailable services. An other problem occurs when a service user has not been designed for multiple services. As a dynamic system may change anytime new services can be registered which would then change the behavior of the the service client.

By using dynamic AOP we can untighten the open service oriented architecture even more. Such a dynamic AOP/SOA solution allows to build an ad hoc environment with many independent services.

Stale References

The key concept behind service oriented architectures is the registration of a service by a provider and the lookup of the service by a client (Figure 2a). In case the service provider is removed all its registered services will get unregistered. This has a significant drawback as it leads to stale references (see [25], p. 72). The users of that service has to check before its usage if the service is still available. The service user is therefore responsible for checking the references himself as in
case a service is removed references would point to nonexisting locations and throw a \textit{NullPointerException}.

OSGi proposes a solution to the stale references problem by using the \textit{Service Tracker} and \textit{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 has to be implemented by the bundle developers themselves.

Another solution to the stale references problem has been proposed with the \textit{Service Binder} [15] an automatic service dependency management. The registering and unregistering 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 unregistered when the service is removed. This observer concept is very similar to the \textit{Service Tracker}. The \textit{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 the Service Binder concept the service has to provide observer methods. With our proxy approach the observer pattern is hidden inside the proxy and transparent to the user.

Once the real implementation is unregistered by the provider the reference in the proxy is removed. A proxy which contains now an empty implementation needs in case of a return value to return a default value or throw an appropriate exception. The default behavior has to be specified when the proxy is generated and can be changed any time later. To specify the default behavior we use meta tags inside the interfaces. The meta tags are then queried when the proxy is created.

2.3 Benchmark

As a first benchmark we measured 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 is an iPAQ H3970, 400 MHz with familiar Linux 0.7.1. As JRE we use a 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.

<table>
<thead>
<tr>
<th>VM/devices</th>
<th>norm</th>
<th>proxy</th>
<th>after</th>
<th>arnd</th>
<th>before</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvm/ipaq</td>
<td>0.16</td>
<td>8.08</td>
<td>8.47</td>
<td>7.55</td>
<td>8.52</td>
<td>8.18</td>
</tr>
<tr>
<td>cvm/lap</td>
<td>0.02</td>
<td>0.57</td>
<td>0.63</td>
<td>0.56</td>
<td>0.62</td>
<td>0.60</td>
</tr>
<tr>
<td>jre/ipaq</td>
<td>0.45</td>
<td>13.5</td>
<td>15.5</td>
<td>13.6</td>
<td>15.8</td>
<td>15.0</td>
</tr>
<tr>
<td>jre/lap</td>
<td>0.01</td>
<td>0.08</td>
<td>0.09</td>
<td>0.08</td>
<td>0.09</td>
<td>0.09</td>
</tr>
</tbody>
</table>

Table 1: AOP overhead of 16 method calls in millisecond

In a first measurement (\textit{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 following measurements an aspect has been inserted on every method to measure the \textit{after}, \textit{around}, and \textit{before} overhead with the average in the last column.

It can be seen switching on the proxy has a significant impact for 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 used memory at runtime the CVM used about 5 MByte
whereas the Blackdown over 20 MByte for running this 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 percentage overhead 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
then one method calls. In the next section the distributed infrastructure uses the proposed dynamic
container. The overhead in such an infrastructure decreases significantly.

3 Distributed Infrastructure

The proposed dynamic lightweight container can now be used to build applications out of different
components. Our main target is the distributed mobile infrastructure, shown on the right side of
Figure 1. Adding new components or changing the behavior of running services are then performed
transparent to the user.

The peer to peer implementation takes over concepts from JXTA [7, 18]. By disassembling
their layers and taking over the core functionality we get instead of one big package many smaller
components which can be plugged together as required. This flexible architecture allows to use
larger distributed concepts in resource constraint devices. It follows a description of the different
components designed as components with services and aspect behavior.

3.1 Message Layer

The first layer in Jadabs is a basic messaging layer to 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 layer is very general and
independent of a RPC communication layer communication can be set up in an ad hoc manner
also with platforms other than Java.

Any implementation of this interface has then to be conform with the JXTA message specifi-
cation in [1]. This allows to connect any small device to the JXTA’s infrastructure.

UDP, TCP, Bluetooth

More and more devices are emerging with one or more network layers like mobile phones with GSM
and Bluetooth or PDAs with Bluetooth and in some cases also W-Lan. This requires an infrastruc-
ture where a messaging implementation can be plugged in when the connection is available. As
our architecture allows to plugin new implementations at runtime we can configure devices later
on when the network is available.

We build up our implementations on top of the UDP, TCP, and Bluetooth network layer. Each one of this implementation is independent of the other and packaged as an OSGi bundle. The
message layer itself is registered as a dAOP service allowing the concrete network implementation
to be plugged in as an aspect.

As long as for example the Bluetooth network layer is not available the bundle is not required
keeping free the memory for other bundles. Along with freeing memory the processor utilization is
optimized as not used 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 by another already
available network layer or can be copied to the device. Higher layers like the event system allow to
deploy such a bundle over an already established connection.

The three implementations, UDP, TCP, and Bluetooth have been chosen for our ad hoc mes-
senger scenario. 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 in the Internet.
Dispatcher

In a system where different networks are available, for example W-Lan and Bluetooth, a dispatcher is needed to send the message to the appropriate network layer. For example a node has UDP as an initial network layer started. Once a new network layer is available, for example when the Bluetooth is attached, the node may now receive messages over two different network layers. A message which is now sent needs first to be checked which network is appropriate. 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 query the network implementations for which one is responsible to send the message. As in the Bluetooth implementation a list of connected other Bluetooth devices is kept the implementation can decide if it is responsible for sending the message. The UDP implementation can be chosen as default in case the message can not be sent over Bluetooth.

The functionality of the dispatcher could be adapted for 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

The message layer which only operates on simple messages has been extended with the event system layer. This additional bundle adds a type system on top of the messaging layer and allows to publish and subscribe for events sent over the message layer. The publish and subscribe model is based on the concepts of [8, 10, 14] and the type- and attribute-based subscription model of [27, 4]. 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 subscription through the Internet.

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 a core functionality is the possibility to discovery new devices and services in the environment. Many discovery technologies have emerged for different platforms and environments like Jini [3], UPnP [34], and JXTA [1]. Jini and UPnP are supported through OSGi service APIs.

In our implementation we focus on the concepts of 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.

To extend an already set up interface we merge the event system API with the discovery API.

3.3 Discussion

The proposed distributed infrastructure implementation strongly relates to the JXTA specification. Our main goal is here not to implement a new distributed infrastructure concept. It should be compatible on different layers with the chosen JXTA’s peer to peer concept. By implementing different layers of JXTA on top of our proposed dynamic lightweight container we are able to insert and extend the layers as needed. This way resource constraint devices, not capable of running a 2 MByte JXTA infrastructure, may still participate in such an infrastructure.
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 layer the UPnP messaging layer could be taken for UPnP devices. Dynamically new protocols can be added or removed transparent to the other components in the system. ReMMoC [12] introduces for this purpose a reflective middleware for mobile computing. The mobile device is then able to configure and reconfigure itself between a number of discovery and binding personalities like UPnP, SDP, and Jini.

We regard 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. Instead of the method \texttt{publish(event)} a \texttt{publish(event, timeout)} is supported. This is only a slight modification of the already known publish method where continuously the event is published. 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 3, scenario 1). As a last benchmark we looked at the production overhead of the full system when sending events between two nodes (Figure 3, scenario 2). The test setup is similar to the container benchmark in the previous section.

#### Framework Benchmark

In this benchmark we measure the Proxy overhead of the Jadabs architecture with the event system layer and message layer (Figure 3 1). 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.

We did 10’000 calls and achieved a standard deviation of less then 1.5%.

<table>
<thead>
<tr>
<th>VM/device</th>
<th>normal</th>
<th>proxy</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>jre/lap</td>
<td>0.232 ms</td>
<td>0.247 ms</td>
<td>6.5%</td>
</tr>
<tr>
<td>cvm/ipaq</td>
<td>15.8 ms</td>
<td>17.0 ms</td>
<td>7.3%</td>
</tr>
</tbody>
</table>

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 quite smaller then compared to the several times slower performance in the
Real Scenario Benchmark

In the real scenario benchmark we measured the round trip time of an event from a laptop to an iPAQ. Figure 3 - (2) 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 generated out of the message and the subscribed benchmark service notified. In the 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).

<table>
<thead>
<tr>
<th></th>
<th>normal</th>
<th>proxy</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lap - iPAQ</td>
<td>40.46 ms</td>
<td>41.45 ms</td>
<td>2.5%</td>
</tr>
</tbody>
</table>

Table 3: Proxy overhead for real scenario

The measurement shows an overhead of 2.5% by using the proxy approach.

By looking at a real scenario with the dynamic container we have shown that the proxies approach is a feasible solution for small device where only a J2ME/CDC implementation is available. In small devices where a lot of time the communication is involved or also user actions have to be performed the proxy overhead can be regarded as negligible.

4 Application

The Jadabs architecture has been used to build a messenger scenario as shown in Figure 4. With this messenger application a user can communicate with other users over different networks and in a secure way. Each messenger application on the PDAs has only a basic architecture at startup. New behavior like dispatching, discovery, and security is added transparently to the user at a later time.

We used iPAQs running Linux and on top of J2ME/CDC a basic Jadabs architecture. The basic Jadabs architecture 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 as with TCP to many package collisions occurred. This allows 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 have been solved inside the event system. We discovered that UDP is very reliable when not to many nodes are communicating. Therefore the reliability can also be switched off.

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 in an other network it may be able to reach the news-server with this discovery extension again.

In this scenario we controlled the nodes over a controller. We are now in progress of extending Jadabs with a discovery functionality which allows to do automatic configuration of such infrastructures.

5 Related Work

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

The trend to smaller component containers then J2EE have already started by 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 [22], JBoss [13], and Spring [32] 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 relay 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 OSGi [25], the Open Service Gateway infrastructure, or Fractal [11] 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 brought acceptance in the industry as well as the open source community.

ReMMoC [12, 26] is a reconfigurable reflective middleware to support mobile application development. It uses OpenCOM [20] 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 [23]. 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. 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 concept independent of any technology.

Prism [21] 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 fare no runtime changes can be performed as required in a dynamic reconfigurable environment.

Midas [30, 29] is a dynamic middleware which combines the dynamic AOP implementation PROSE [28] with the Jini infrastructure. By using Jini the infrastructure is bound to only Java environments additional 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 only bound to Java environments.
JXTA [18, 1, 7] 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 to implement 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 when it comes to deploy on small devices. 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 it is not need 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 then 2 MByte and 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. New functionality can then be loaded and activated when needed. The concept would even allow to bootstrap the full JXTA on more powerful nodes. 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 we propose a dynamic lightweight architecture for resource constraint 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.

For the distributed infrastructure we took over concepts from JXTA a peer to peer infrastructure. As JXTA is to big for small devices our goal is to integrate their concept with pluggable services to download and activate them when required. This leads to smaller distributed infrastructure which can be run on small devices. The dynamic lightweight container is therefore the core to bootstrap a distributed architecture. A message layer conform to JXTA’s message specification suffice as a first layer for communication. Different implementations are supported according to the devices possibilities like IEEE 802.11b, Lan or Bluetooth. As a second layer we propose the simple event system interface. Both layers are 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 can be plugged in when required. Even though we adapt the distributed infrastructure with the dynamic AOP concept the application can use the same concept.

The proposed solution with proxies extends the service oriented architecture of OSGi by the dynamic AOP concept. 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.

Currently we are in the progress of extending the discovery mechanism for autonomous configuration. A .NET implementation is under way and has already implemented the messaging infrastructure. As the loading and unloading of services in .NET is different an other 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


