

# A Peer-to-Peer Light-Weight Component Model for Context-Aware Smart Space Applications

Alois Ferscha, Manfred Hechinger, Rene Mayrhofer and Roy Oberhauser

**Abstract**—Mobile Peer-to-Peer (P2P) computing applications involve collections of heterogeneous and resource-limited devices (such as PDAs or embedded sensor-actuator systems), typically operated in ad-hoc completely decentralized networks and without requiring dedicated infrastructure support. Short-range wireless communication technologies together with P2P networking capabilities on mobile devices are responsible for a proliferation of such applications, yet these applications are often complex and monolithic in nature due to the lack of lightweight component/container support in these resource-constrained devices. A threatening field of application is “smart space” control, i.e. software architectures to control various home appliances and embedded home facilities in a personalized, spontaneous and intuitive way. Future home environments are expected to be highly populated by ubiquitous computing technology, allowing to integrate various aspects of home activities seamlessly into walls, floors, furniture, appliances, and even clothing – thus raising the need for lightweight, versatile and component based software architectures to harness such technology rich environments.

In this paper we describe our lightweight software component model P2Pcomp that addresses the development needs for mobile P2P applications. An abstract, flexible, and high-level communication mechanism among components is developed via a ports concept, supporting protocol independence, location independence, and (a)synchronous invocations; dependencies are not hard-coded in the components, but can be defined at deployment or runtime, providing late-binding and dynamic rerouteability capabilities. Peers can elect to provide services as well as consume them, services can migrate between containers, and services are ranked to support Quality-of-Service choices. Our lightweight container realization leverages the OSGi platform and can utilize various P2P communication mechanisms such as JXTA. A “smart space” application scenario demonstrates how P2Pcomp supports flexible and highly tailorable mobile P2P applications.

**Index Terms**—Peer-to-peer computing, pervasive computing, context awareness, component framework, OSGi, JXTA, smart spaces.

Manuscript received March 3, 2004. This work was supported by the Peer-to-Peer Coordination project Nr. J580.0 of the alliance between Siemens AG and Johannes Kepler University Linz.

A. Ferscha is with the Institut für Pervasive Computing at the Johannes Kepler University of Linz, Austria (phone: +43-732-2468-8556; e-mail: ferscha@soft.uni-linz.ac.at)

M. Hechinger is with the Institut für Pervasive Computing at the Johannes Kepler University of Linz, Austria (e-mail: manfred@soft.uni-linz.ac.at)

R. Mayrhofer is with the Institut für Pervasive Computing at the Johannes Kepler University of Linz, Austria (e-mail: rene@soft.uni-linz.ac.at)

R. Oberhauser is with Corporate Technology, Siemens AG, Munich, Germany (e-mail: roy.oberhauser@siemens.com)

## I. INTRODUCTION

**S**MALL, mobile communications devices such as PDA’s, mobile phones, wearable devices, and smart tags are gaining increasing hardware, networking, software, and user interaction capabilities. As the pervasiveness of these devices increases, there is a correlating increase in the both the scale and the level of heterogeneous integration in these infrastructures.

Furthermore, the increasing expectations and demands for greater functionality and capabilities from these devices often result in greater software complexity for applications. Because these resource-constrained environments have not had the rich component and container support commonly available for enterprise development, the result in this context has often been a potpourri of “stovepipe” applications with few opportunities for reuse and unplanned integration without significant effort. Where functionality modularization was planned, e.g. with services, these have often been coupled to a single middleware or communication protocol (e.g. COM [12], RMI [19], MOM-based JMS [16], SOAP [14], JXTA [15]).

Thus, there is an increasing need to abstract and encapsulate the different middleware and protocols used to perform the interactions from the components involved in the interactions. Szyperski [26] defines components as “binary units of independent production, acquisition, and deployment that interact to form a functioning system.” Components consist of an object or cohesive group of objects with (a) clearly defined interface(s) that typically provide a service or set of services. The component model is an abstract description of the relevant aspects that are common to the components (such as communication mechanisms, packaging, etc.) as well as the runtime framework in which they are managed (the container). Unfortunately component models (such as Java EJB [13], CORBA CCM [11]) are often tied to a remote protocol, an inhibiting factor with the wide spectrum of protocols for the increasingly extensive pervasive applications in this realm.

For mobile P2P applications, however, the classical designs of component models and architectures either suffer from extensive resource demands (memory, communication bandwidth, CPU) or dependencies on the operating system, protocol, or middleware (e.g. .NET, CORBA ORBs). In addition, any infrastructure must not significantly diminish the ability of applications to address the increasing functionality and complexity demands; otherwise, its adoption would be

jeopardized. Hence lightweight component models are needed with containers able to execute on resource-constrained platforms (PDAs) to enable reusability, the dynamic distribution and deployment, location transparency - irrespective of dynamic changes in the peer topology and combination, platform and middleware independence, standardized component definitions, hot-swapping, and optimal tailoring of service configurations. Therefore, a method for node-transparent and transport-transparent component interaction could significantly reduce the development time and costs of distributed component-based applications in our context.

The emerging trends of embedding information technology like tiny computers, sensors and actuators into objects and spaces like homes, offices, cars, trains, stations, airports, etc. will lead to future daily life situations where people and environments are mediated by various invisible computers, interaction devices, sensors and technology rich objects of everyday use. Due to the most recent technological developments, smart environment scenarios appear possible, in which almost every object in our everyday environment will be equipped with embedded processors, wireless communication facilities and embedded software to perform and control a multitude of tasks and functions. Many of these objects will be able to communicate and interact with the background infrastructure (e.g. the Internet), but also with each other [5]. Terms like “context-aware” smart spaces have appeared in the literature to refer to such technology-rich environments, which intelligently monitor the objects of a real world (like persons, things, places), and interact with them in a situative, pro-active, autonomous, sovereign, responsible and user-authorized way [6].

Among the most important issues in such computer mediated daily lives is the provision of context-awareness, and the integration of physical and virtual spaces to reduce interaction complexity and to customize the behavior of the environment to the users preferences, habits or even emotions. In smart “home” spaces with a dramatically growing number embedded devices therefore, middleware has to be designed and implemented in order to cope with a large number of different home appliances – all of them with very limited computational, memory and communication resources, to coordinate their concerted interaction, the seamless movement of a user in those spaces, and the personalization of the control depending on the users situation. We refer to middleware built for the purpose of controlling home environments “roomware”. Challenged by the goal of a seamless configuration and control of home appliances according to each user’s preference and situation – for which traditional approaches based on Jini and HAVi have turned out to be merely successful – we have designed and implemented a framework that strictly follows (i) the Peer-to-peer interaction concept, that (ii) adopts standard protocols as much as possible to ensure the interoperability among components and commercial products, that can be (iii) configured wirelessly from anywhere, by any user, at any

place. P2Pcomp has been built to support the implementation of roomware services in smart spaces [2] [3]. Opposed to centralized approaches in smart space middleware, P2Pcomp has been rigorously designed as a P2P framework, and implemented on top of JXTA. Comparable home environment networking approaches are [20], [21] and [23].

In this paper we motivate and present our component framework P2Pcomp, designed and implemented at the confluence of open standards compliance (OSGi) and the restrictions of limited resource platforms (PDAs and mobile appliances). P2Pcomp aims to ease and support the development of pervasive computing applications based on spontaneous interaction of mobile peers.

A central motivation for P2Pcomp was infrastructural support for context awareness in mobile P2P applications [8]-[10]. Thus the design goals for P2Pcomp were concerned with (i) supporting the description, gathering, transforming, interpretation and dissemination of context information within ad-hoc, highly dynamic and frequently changing computing environments, (ii) dynamically discovering, inspecting, composing and aggregating software components in order to identify, control and extend context, as well as overcome context barriers (like time, position, user preference, etc.), and (iii) allow for dynamic interactions among software components in a scalable fashion while satisfying special requirements such as fidelity, QoS, fault-tolerance, reliability, safety and security, etc.

The rest of this paper is organized as follows: in Section 2 we introduce the basic concepts of P2Pcomp, relate those to comparable concepts in the service-oriented container OSGi, and describe why our solution was necessary. Conceptual details of P2Pcomp for ports and containers, together with implementation and syntactical issues are presented in Section 3. Section 4 – in the frame of an application scenario – gives empirical evidence for P2Pcomp being truly lightweight. Our work is compared with other approaches in the literature in Section 5, and conclusions are drawn in Section 6.

## II. GENERAL CONCEPT

For rapid application development of distributed applications in this domain, we can identify two key elements: P2P as a communication paradigm and component-based programming for code reuse. For P2P coordination, the language-independent JXTA framework has established itself as a quasi-standard, but provides no component model. As a component model, the OSGi specification provides a component model geared for resource-constrained devices but lacks support for distributed components.

In our work, we build upon these two technologies and combine them to simplify the development of distributed, component-based applications. In OSGi terminology, a container will be used for managing components (Fig. 1); this

includes installing, starting, stopping and removing components as well as checking dependencies between components. In addition to these basic features of an OSGi-conformant container, it should also communicate with other containers and offer installed components a simple way of communicating with components instantiated in remote containers. In OSGi terminology, a component offers services to other components and is packaged as a bundle. Interaction between bundles is only possible via defined services.

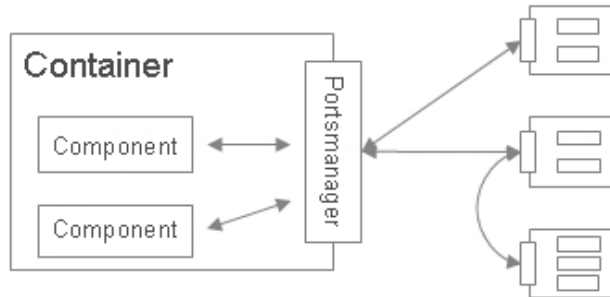


Fig. 1. P2PComp containers/components

With plain OSGi containers, components have to implement communication channels to remote components themselves; the container can only return references to other local components instantiated inside the same container. Thus, the present paper introduces the ports concept: a port is one endpoint of a communication channel and can be used by components to communicate with others. From the component view, only the port is visible, the underlying communication channel is not; this encapsulates, e.g., the protocol or protocol APIs from the component. When ports are used as a general concept of connecting to a service offered by another component, local and remote services can be accessed similarly. The container offers ports as a unified interface to inter-component communication for local as well as remote components, relieving component developers from the task of managing communication with remote components (cf. Portsmanager in Fig. 1).

### III. APPROACH

In the sequel, after introducing the main features of Oscar OSGi, we will present our P2Pcomp ports concept and introduce provide ports as a means to offer services to other components, and uses ports as points of connection for components to access those services.

#### A. Oscar OSG

As an OSGi implementation, the open source package Oscar [18] was used. It is compliant to the OSGi specification and implements most major functionality of OSGi 1.0. Its aim is to provide a fully compliant OSGi 2.0 framework and some of the major elements are already implemented, specifically the:

- Package Admin service
- System Bundle

- Service Tracker
- Service properties and selection algorithm
- Filter class and related framework methods

Although this aim has not yet been completely achieved and some minor compliance issues still have to be resolved, it has many advantages for the development of our ports concepts and for the deployment in resource-constrained systems:

- very lightweight – can easily be embedded in applications
- can fetch bundles (components) from a remote host
- offers an optional shell for interactive commands
- already has some (syntactical) parts of our ports concept (see below for details)
- supports dependencies between bundles
- each bundle is loaded in its own class loader (important for security)
- under an open source license (GPL)

Our code implementing the ports concept is independent of the specific OSGi framework implementation. Although Oscar supports dynamic class loading, it was apparently not designed to support remote services the way it is implemented by our PortsManager, since classes which are exported by a bundle may not be loaded by any other object but by Oscar itself. To override this, and enable the PortsManager to load and instantiate the exported classes, a new Interface PortsManager.ExportedClassFetcher has been created. The interface is implemented by a very small wrapper class for Oscar. While the functionality of the class is small, it was deliberately split into a class and interface; thus, the presented ports concept is usable with any OSGi container implementing this interface (possibly via a wrapper class as it has been done for Oscar).

While the OSGi framework is a good solution to run services within a container, operation is restricted to a single local node since there is no direct support for interoperability with other containers running on remote nodes. Each component that wishes to interact with other nodes must implement the network functionality and the invocation of remote services (see Fig. 2).

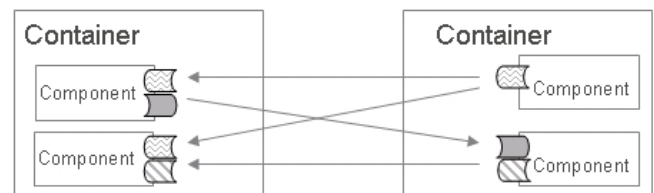


Fig. 2. Remote component interaction in OSGi

#### B. The Ports Concept

For making the implementation of interdependent components as simple as possible, a ports concept is

introduced as an abstract, flexible, protocol-independent, and high-level communication mechanism (see Fig. 3). The main design goal is that the communication should be completely transparent to the actual components; whether it is communication with local or remote components or OSGi-independent Web Services should not be known inside the component. This concept has the additional advantage that dependencies are not hard-coded in the components, but can be defined by the component deployer or at runtime to support very late binding.

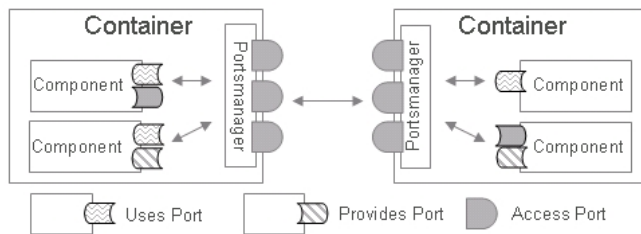


Fig. 3. P2Pcomp ports concept

### C. Provides Ports

A component may have zero or more provides-ports (see Fig. 3). A provides-port is a "service" that is "provided" to other components or to the framework and is defined in terms of a Java interface. When offering a provides-port, a component simply implements a Java interface and "exports" it via an entry in the deployment descriptor. From the component view, it is then up to the container to add this "service" to its internal registry and to advertise it other containers via P2P mechanisms. The container is also responsible for calling the interface methods on behalf of the "service users" when they are unable (or not configured) to call them directly.

In the case that a component is providing ports to two or more other "user" components, there is no prescribed scheduling behavior for the order in which the external invocations are served. It is up to the component implementation to determine this. Each component should supply a "data sheet" that defines any special runtime execution behavior that is required for its correct execution.

### D. Uses Ports

A uses-port can be viewed as a connection point on the surface of the component where the framework can attach (connect) references to provides-ports provided by other components or the framework (see Fig. 3). Viewed from the inside of the component, a uses-port is simply the Java interface the component needs to use. The component makes calls on uses-port references to "use" the "provided" services. A component may have zero or more uses-ports. These ports are named in the code, but the XML descriptor for the component provides a mapping to the actual name used in the system, which can vary from the name used at the time of the component implementation. This supports "very late binding"

of components by the deployer.

### E. Access Ports

An access-port is a connection point at the boundary of a container and is used for connections to other containers (see Fig. 3). It can use any available communication technology, e.g. JXTA, WSDL-based Web Services, SOAP, custom XML over UDP or TCP/IP, RMI, etc. to link local with remote provides- and uses-ports. For components, access ports are invisible because they only use provides- and uses-ports to communicate with other components.

### F. Implementation

The goal of our ports concept is that an invocation of the service implementation on a remote container is, for the programmer of the components, as simple as in the case of local invocation and completely transparent with regard to the location of the service implementation. Even syntactically, the invocation of a remote service should be equal to calling a (local) implementation of the interface.

To accomplish this, a component called PortsManager has been developed as an implementation of the ports concept and is packaged as an OSGi bundle. All components may fetch services via the PortsManager component. If a requested service is not locally available, the PortsManager component interacts with the respective PortsManager on other containers, thus enabling transparent interaction between services, regardless if they are remote or local.

An additional component, the P2PService, is an implementation of access ports for P2Pcomp, implementing JXTA and alternatively a special transport using XML messages over UDP broadcasts and TCP connections. The PortsManager component uses this simple interface for sending messages to other containers and is notified of incoming messages and of devices (peers) entering and leaving spatial proximity (i.e. remote containers becoming available or unavailable). The PortsManager component can use arbitrary implementations of access ports (e.g. for interacting with Web Services) as long as this simple interface is implemented.

The PortsManager component has a number of features which make it appealing for mobile application development:

#### Service fetching:

Local and remote service references can be queried via the PortsManager, which will in turn query the services from those OSGi containers that manage the requested service and forward them to the caller. In addition to the service interface, a filter string resembling an LDAP search filter according to RFC 1960 can be used for fetching a service. Additionally, a specific service reference for a single service implementation can be fetched if hot-swapping (see below) is undesirable for a specific application.

*Service ranking:*

According to OSGi, every service may be given a certain rank which describes its quality, importance, etc. depending on the services context. A services rank can be set within the bundles activator class and usually stays the same while a bundle is in the “active” state. If there is more than one matching service available, the PortsManager decides upon each service’s rank which to load first. Should a service become unavailable for some reason and the service has not been fetched by service reference, then the PortsManager automatically tries to locate the next highest ranked service.

*Hot swapping:*

If the matching service which was used during service fetching disappeared because it was either locally or remotely uninstalled or the specific remote peer is no longer reachable, the PortsManager will automatically try to regain a matching service. The service reacquisition order is the same as if it is fetched initially, i.e. depending on the service’s rank. This behavior enables the PortsManager to allow exchange of equivalent stateless services during run-time, i.e. perform “hot swapping”. To detect service transitions (i.e. new availability of a service, removal of a service or change of service properties), the PortsManager implements the OSGi ServiceListener interface. This extends the standard OSGi local functionality to remote service change notifications.

*Synchronous remote invocations:*

If a service reference returned to a calling component points to a remote device, then the invocation of methods on this service will be done remotely. Input parameters will be transparently forwarded over the network, the remote component method will be invoked and the return value will be transferred back while the client is blocked. Thus, the syntax and semantic of calling a method on a service that has been fetched via the PortsManager are, from the caller’s point of view, equal to calling a method of a local Java object.

*Asynchronous remote invocations:*

For P2P interactions, asynchronous object-oriented invocations provide enhanced application development vs. lower-level messaging. The PortsManager component offers the `asyncInvoke` method (Fig. 4), which takes the service reference, the method name and its parameters as input arguments and returns a token for retrieving the remote method’s result value when the remote method has terminated. The method of the remote component is then invoked asynchronously without blocking the caller – the status of the method can be queried using the returned token or the caller can register to receive an event when it terminates.

```
Object[] args = new Object[1];
args[0] = "content";
AsyncInvokeToken token =
    portsManager.asyncInvoke(
        prsntService, "show", args);
...
token.getResult();
```

Fig. 4. An asynchronous invocation

*G. Method call syntax with PortsManager*

Provides-ports are Java interfaces that are implemented by the components and registered with the container by listing them in the deployment descriptor. When requesting a service via the PortsManager, the requesting component connects its uses port to the provides port of the service. The PortsManager component is responsible for returning the correct Java object when the uses-port is requested by a component; it is a stub object (i.e. a generic dynamic proxy) which either calls the respective methods of the locally available service implementation object or translates the Java method calls to messages, sends them to a remote container, waits for remote execution and then returns the value contained in the received message.

To dynamically generate stub objects that implement the required Java interface for arbitrary services, a Java Dynamic Proxy [19] (available since JDK 1.3) is used. To process incoming requests (e.g. Java RMI, SOAP, JXTA) and appropriately call interface implementations of local components, the container interprets received messages and calls the respective component (which must be known to the container’s registry) methods via standard Java reflection.

```
prsntSrvRef = context.getServiceReference(
    PresentationService.class.getName());
prsntService = (PresentationService)
    context.getService(prsntSrvRef);
if (prsntService != null)
    prsntService.show(content);
```

Fig. 5a. Retrieving a service reference and invoking a service: plain OSGi vs. PortsManager

Fig. 5a shows a standard OSGi container-local service invocation while Fig. 5b shows the same invocation with the use of our PortsManager.

```
portsManRef = context.getServiceReference(
    PortsManager.service.PortsManager.
    class.getName());
portsManager = (PortsManager)
    context.getService(portsManRef);
prsntService = (PresentationService)
    portsManager.getService(
    PresentationService.class.getName());
if (prsntService != null)
    prsntService.show(content);
```

Fig. 5b. Retrieving a service reference and invoking a service: PortsManager

As can be seen, in addition to first fetching the PortsManager (within a container-independent OSGi bundle), the only change is to retrieve the service reference via the PortsManager service instead of the OSGi BundleContext object. Since calls on a service reference are equivalent, existing components can be easily adapted to use the PortsManager. The overhead in code size for using the PortsManager is insignificant and the run-time overhead is marginal, because Java dynamic proxies are used and the hot swapping feature (which dynamically checks service availability) can be deactivated if necessary.

If even more transparency of the PortsManager is required, the BundleContext context instantiated by the container can be modified. For most OSGi containers it should be possible to modify the class factory so that it returns a wrapper as the BundleContext, which will directly use the PortsManager for normal components. This would allow unmodified, OSGi conformant components to use the PortsManager features.

Thus for smart spaces, a mobile user could retrieve the presentation service reference once and use it at any location where such a service is available without reconfiguring the presentation client. In an auditorium, a powerful PresentationService implementation with overhead projector and audio system might be available. When leaving the auditorium and presenting a few more details in a cafeteria, a normal notebook computer could offer a less powerful PresentationService implementation (with lower service ranking). The client application, running on the user's PDA, does not need to notice this service transition, because method calls on the service reference will be resolved dynamically when the initial service becomes unavailable. Hot-swapping combined with service ranking greatly supports users on the move by fully exploiting the possibilities of ad-hoc environments.

#### IV. A SMART SPACE SCENARIO

To point out the benefits of the ports concept and its means for service ranking and hot swapping for real world applications, we describe the following smart space scenario. This scenario consists of two major entities (peers): a mobile user with a wearable computer and micro-optical display and a smart room with a wall interface [Simon]. With his wearable acting as personal agent for communication, the user is

capable of receiving email enriched with video messages (video mail). While he is on the move, the wearable display incoming video mail on the user's micro-optical display (See Fig. 6).



Fig. 6. Video-mail perceived via a micro-optical display

Although this allows mobility, the display resolution and color depth are intrinsically limited. When the current user's situation allows using better means for displaying video mails, a switch to these means could be performed automatically and without explicit user interaction. The ports concept naturally allows such convenience.

Such an application would require two services: The video mail agent service is responsible for receiving video mails using arbitrary communication media, whereas the presenter service can be used to display arbitrary content to the user. As depicted in Fig. 7, the video mail agent service utilizes a presenter service for playback of video messages upon receipt.

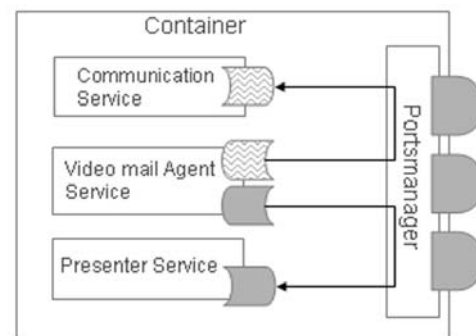


Fig. 7. Video mail agent service utilizing a local presenter service and a communication service

While on the move, the only instance of a presenter service available is the one using the directly connected, local micro-optical display. Both services are running on the wearable; thus, the video mail agent service could directly access the micro-optical presenter service by simply invoking appropriate methods. However, using the ports manager as a lightweight component middleware between services offers a significant advantage while having minimal impact on implementation efforts. Namely, the video mail agent service is then decoupled from the specific presenter service, allowing the presenter service to be exchanged on the fly -- the video mail agent service does not even have to be aware of any changes. As soon as the user enters the smart room, the

wearable peer detects the smart room peer due to spatial proximity. Therefore, the wearable immediately finds another instance of a presenter service, implemented by the smart room with its integrated data projector. Since the wall interface is of higher quality than the micro-optical display (i.e. it is more convenient), it is assigned a higher service rank. As a consequence, the ports manager will automatically select the superior presenter service for requests by the video mail agent service (hot swapping), as shown in Fig. 8.

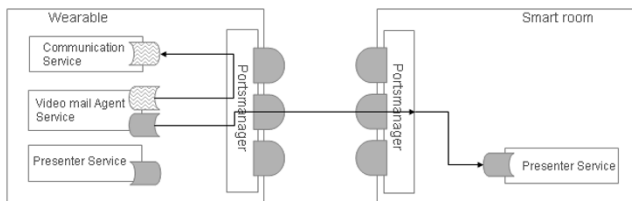


Fig. 8. Video mail agent service utilizing a remote presenter service and a communication service

While the user is located inside the smart room, all received video mails will automatically be played on the wall interface using the data projector (see Fig. 9). When the additional presenter service becomes unavailable because the user leaves the smart room, the ports manager will automatically fall back to the locally available micro-optical presenter service, thus further received video mails will be played on the micro-optical display.



Fig. 9. Video-mail perceived on the wall interface in the smart room

To summarize, the particular benefits of the ports concept for challenges posed by smart space applications or alike scenarios are that it

- is lightweight and thus suitable for a wide range of devices
- does not require any central infrastructure component, but is fully decentralized and designed for ad-hoc situations which are typically encountered in smart space environments. The lack of a central infrastructure allows the ports concept

to be used in smart environments without expensive infrastructure installations.

- allows easy and transparent invocation of local and remote services independently of their location and implementation
- provides ideal support for changing environments through automatic hot swapping of services. This feature can significantly improve fault-tolerance of the overall system in loosely coupled, peer-to-peer environments
- supports automatic selection of the best service available, based upon a simple ranking mechanism
- simplifies dynamic installation of applications upon spatial proximity

## V. EVALUATION

### A. Performance and Scalability

A performance case study for the P2Pcomp implementation has been conducted in order to demonstrate feasibility and scalability of P2Pcomp for different devices (Table 1). To test method invocation overhead with a few parameters, echoInt service is used (int result = echoInt(int a,int b)). The echoString service (String result = echoString(String data)) tests the parameter marshalling code and scalability regarding varying parameter sizes of the Portsmanager by passing in and returning a string using sizes varying from 10 to 105 bytes. Both services actually do nothing except returning the input parameters, since we do not want to measure the performance of the services itself but the performance of the invocation, passing in and returning different parameter sizes. Both services have been invoked in the following settings: a) without component indirection (monolithic), b) invoking the service via Oscar, c) using Portsmanager to access the service on the local device d) using Portsmanager to access the service on a remote device. Table 2-4 show the test results for settings a), b) and c). The values specified represent the average duration for invoking the corresponding service.

Since the overhead for method invocation on remote devices heavily depends on the used transport technology, setting d) has been conducted using TCP with XML messages (Table 5) and then with JXTA (Table 6) with 100Mb/s Ethernet and 11 Mb/s WLAN. Measurements using JXTA on the IPAQ were not possible.

TABLE 1. USED DEVICES

Device	CPU	RAM	OS	Java VM
Notebook	P3, 850 MHZ	256 MB	WinXP	Sun 1.4.1
Server	P4, 2.4 GHZ	1.0 GB	Linux 2.4.22	Blackdown 1.4.1
IPAQ	StrongArm206 MHZ	64 MB	Familiar Linux 0.7.1	Blackdown 1.3.1

TABLE 2. AVERAGE CALL DURATION, SETTING A)

in $\mu$ sec	Notebook	Server	IPAQ
echoInt	0.04	0.036	8.725



str(10 <sup>2</sup> )	0.04	0.032	7.203
str(10 <sup>4</sup> )	0.05	0.033	7.480
str(10 <sup>5</sup> )	0.06	0.038	10.372

TABLE 3. AVERAGE CALL DURATION, SETTING B)

In $\mu$ sec	Notebook	Server	IPAQ
echoInt	0.05	0.036	8.718
str(10 <sup>2</sup> )	0.04	0.031	7.217
str(10 <sup>4</sup> )	0.05	0.031	7.692
str(10 <sup>5</sup> )	0.06	0.044	10.146

TABLE 4. AVERAGE CALL DURATION, SETTING C)

in $\mu$ sec	Notebook	Server	IPAQ
echoInt	1.41	0.62	428.36
str(10 <sup>2</sup> )	0.8	0.43	247.30
str(10 <sup>4</sup> )	0.8	0.43	252.44
str(10 <sup>5</sup> )	1.0	0.45	275.94

Tables 1-4 show that the invocation of the echoString service is faster than the invocation of the echoInt service if the string is small enough. The reason for this is that the Java dynamic proxy code is faster for small strings than for integer variables. The invocation in setting c) is slower than in setting a) and b) since the calls are running “through” the Portsmanager and the Java dynamic proxy code.

TABLE 5. AVG. CALL DURATION FOR TCP, SETTING D)

in millisecc	Notebook→ Server/Ethernet	Notebook→ Server/Wlan	IPAQ → Server/Wlan
echoInt	8.51	15.72	242.89
str(10 <sup>2</sup> )	6.51	15.02	268.37
str(10 <sup>4</sup> )	31.85	78.82	3,073.75
str(10 <sup>5</sup> )	533.57	916.61	28,622.40

Table 5 and 6 shows the performance of our implementation for invoking remote methods using various parameter sizes. The measured values show that invocation duration is comparable to other means of remote method invocations.

The TCP transport used for the measurements in table 5 could be improved to speed up remote method invocations, for example by sending raw data instead of XML messages, leading to shorter invocation duration. The results also show that our implementation scales well regarding the size of the input parameter at least for local invocations. When invoking methods remotely, scalability heavily depends on the transport technology’s parameters (throughput, latency, frame size ...).

TABLE 6. AVG. CALL DURATION FOR JXTA, SETTING D)

in millisecc	Notebook→ Server (100Mbit, Ethernet)	Notebook→ Server (11Mbit, Wlan)	IPAQ → Server/Wlan
echoInt	39.56	47.07	n.a.
str(10 <sup>2</sup> )	30.15	41.36	n.a.
str(10 <sup>4</sup> )	62.99	119.17	n.a.
str(10 <sup>5</sup> )	626.40	1,111.60	n.a.

## VI. COMPARISON WITH RELATED WORK

Expeerience [21] is a middleware layer over JXTA that

addresses issues with JXTA with regard to intermittent connections in adhoc environments. It supports code mobility and service migration, including state to the extent of support for mobile agent systems. Expeerience does not, however, address the component models issue with JXTA nor protocol exchangeability as P2Pcomp does.

With regard to the combination of OSGi and JXTA, the advantages of extending OSGi with JXTA for Virtual Home Environments are described in [20]. It does not address a distributed component model, protocol exchangeability, and QoS for adhoc environments.

OSGi component-related work includes Beanome [1], a lightweight component model and framework on top of OSGi to support complex applications. While Beanome includes component descriptors, factories, a registry, and introspection capabilities, it does not address various issues that P2Pcomp does, such as remote communication, “transparent” asynchronous and synchronous remote invocation, OSGi peer discovery, protocol independence, dynamic binding, dependability, etc.

In the area of component communication, [7] presents a lightweight XML-based middleware based on a ports concept. While addressing protocol exchangeability with various transport channels and integration via XSLT-based connectors, it uses a generative approach that may limit runtime flexibility vis-à-vis P2Pcomp and does not address containers and component lifecycles.

The JavaPorts framework [22] aims to simplify multi-threaded distributed P2P applications with a component model. While it uses a location-independent ports concept and supports asynchronicity, it appears to be primarily focused on parallel computing and does not address the issues of mobile adhoc environments.

SEESCOA [24] supports dynamic reconfiguration and evolution of components in embedded systems by leveraging ports to reroute messages between components. However, the intent is not aimed at supporting P2P application interactions and protocol independence. Lightweight component models for embedded systems are discussed in [25], including the use of a ports concept. For our context, however, these models incorporate restrictions such as domain-specific languages while footprint and performance measurements are unavailable.

In general, embedded systems development is a different paradigm class from that of mobile P2P applications. While both are resource constrained, mobile P2P applications often are primarily focused on (i) distributed, heterogeneous and spontaneous communication, (ii) dynamic extensibility, and (iii) user interaction capabilities.

While agent systems appear to have a similar focus or be appropriate for this environment, the code development and runtime paradigm usually requires additional skills versus a component developer. P2Pcomp does not require code to be designed as an agent or to be mobile, but allows the container to manage the interactions between components in such an environment. Components can be passive, whereas agents are



usually designed to be active.

P2Pcomp has been built to support the implementation of roomware services in smart spaces [2] [3]. Due to the most recent technological developments, smart environment scenarios appear possible, in which almost every object in our everyday environment will be equipped with embedded processors, wireless communication facilities and embedded software to perform and control a multitude of tasks and functions. Many of these objects will be able to communicate and interact with the background infrastructure (e.g. the Internet), but also with each other [5]. Terms like “context-aware” smart spaces have appeared in the literature to refer to such technology-rich environments, which intelligently monitor the objects of a real world (like persons, things, places), and interact with them in a situative, pro-active, autonomous, sovereign, responsible and user-authorized way [6]. Opposed to centralized approaches in smart space middleware, P2Pcomp has been rigorously designed as a P2P framework, and implemented on top of JXTA. Comparable home environment networking approaches are [20], [21] and [23].

Smart space scenarios (such as roomware, smart homes) along with their underlying mobile, resource-constrained P2P applications will continue to challenge software architecture on various points, including QoS, performance, footprint, flexibility, extensibility, manageability, heterogeneity, updateability, testability, scalability, availability, etc.

## VII. CONCLUSION

In this paper, we have discussed the new challenges posed by mobile, completely decentralized, ad-hoc P2P applications to the design of component-based distributed software systems. As the devices representing peers in such applications are usually heterogeneous and resource-constrained, there is the need for an appropriate, lightweight component model. With our OSGi-compliant P2P framework P2Pcomp, we have integrated a minimal set of component model concepts (containers and ports for component interaction, location-independence, protocol-independence, dynamic deployment and binding of components, lifecycle management, packaging and distribution, etc.) on a very small software footprint. Our framework, P2Pcomp, thus represents an operational runtime environment that is both conceptually and physically lightweight, addresses the unique development needs in this context, and enables flexible and highly tailorable component-based, distributed, mobile P2P applications.

## ACKNOWLEDGMENTS

We wish to express our thanks to Thomas Köckerbauer and Michael Kumar for their implementation support of the Ports concept.

## REFERENCES

- [1] Cervantes, H., and Hall, R. Beanome: A Component Model for the OSGi Framework, in *Software Infrastructures for Component-Based Applications on Consumer Devices*, Lausanne, September 2002.
- [2] Streit, N. A. et al.: Roomware: Towards the Next Generation of Human-Computer: Interaction based on an Integrated Design of Real and Virtual Worlds. In J. A. Carroll (Ed.): *Human-Computer Interaction in the New Millenium*, Addison Wesley, pp. 551-- 576. 2001.
- [3] Phillips, P., Friday, A. and Cheverst, K.: Understanding Existing Smart Environments: A brief classification, In *Workshop on Ubiquitous Computing in Domestic Environments*, pages 1--4., Sept. 2001. Lancaster University.
- [4] Open Management Group: Unified Modeling Language version 1.4 specification, “formal/01-09-67”, 2001.
- [5] Mills. K.: *AirJava: Networking for Smart Spaces*, National Institute of Standards and Technology, US.
- [6] Fox, A., Johanson, B., Winograd, T., and Hanrahan, P.: Integrating Information Appliances Into an Interactive Workspace, *IEEE Computer Graphics & Applications* (special issue: “Off the Desktop”), May/June 2000.
- [7] Löwe, W., and M.L. Noga: “A Lightweight XML-based Middleware Architecture”, 20th IASTED International Multi-Conference Applied Informatics (AI), 2002.
- [8] Dey, A. K.: Understanding and Using Context. *Personal and Ubiquitous Computing*, Special Issue on Situated Interaction and Ubiquitous Computing, 5(1), 2001.
- [9] Ferscha, A.: Contextware: Bridging Virtual and Physical Worlds. *Reliable Software Technologies, AE 2002*. Springer Verlag, LNCS 2361, pp 51-64, 2002.
- [10] Ferscha, A.: Collaboration and Coordination in Pervasive Computing Environments. *Proceedings of the 12th International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE 2003)*, IEEE Computer Society Press, pp 3-9, 2003.
- [11] Corba Specification. OMG, <http://www.omg.org/technology/documents/formal/corbaiiop.html>.
- [12] Component Object Model. Microsoft, <http://www.microsoft.com/com/>.
- [13] Enterprise JavaBeans Specification. SUN Microsyst. <http://java.sun.com/products/ejb/docs.html>.
- [14] Simple Object Access Protocol (SOAP), <http://www.w3.org/2000/xp/Group/>.
- [15] Project JXTA. <http://www.jxta.org>.
- [16] Java Message Service (JMS). SUN Microsystems, <http://java.sun.com/products/jms>.
- [17] Open Services Gateway Initiative (OSGi), <http://www.osgi.org>.
- [18] Open Service Container Architecture (Oscar), <http://oscar-osgi.sourceforge.net/>.
- [19] Java 2 Platform, Standard Edition, <http://java.sun.com/j2se/>.
- [20] Loeser, C., W. Mueller, F. Berger, and H. Eikerling: “Peer-to-Peer Networks for Virtual Home Environments”, *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS’03)*, 2003.
- [21] Bisignano, M., A. Calvagna, G. Di Modica, and O. Tomarchio: “Expeerience: a JXTA middleware for mobile ad-hoc networks”, *Proceedings of the Third International Conference on Peer-to-Peer Computing*, 2003, pp. 214-5.
- [22] Manolakos, E.S., Galatopoulos, D. and Funk, A.: “Component-Based Peer-to-Peer Distributed Processing. Heterogeneous Networks Using JavaPorts”, *IEEE International Symposium on Network Computing and Appl.*, 2001, pp. 234-7.
- [23] Alda, S.: “Adaptability in Component-Based Peer-to-Peer Applications”, *2nd International Conference on Peer-to-Peer Computing (P2P’02)*, 2002.
- [24] Vandewoude, Y. and Berbers, Y.: “Run-time Evolution for Embedded Component-Oriented Systems”, *Int. Conf. on Software Maintenance (ICSM’02)*, 2002.
- [25] H. John Reekie and Edward A. Lee: “Lightweight Component Models for Embedded Systems”. Technical report, Electronics Research Laboratory, University of California at Berkeley, UCB ERL M02/30, October, 2002.
- [26] Clemens Szyperski. *Component Software*. Addison-Wesley, 1998.

**Alois Ferscha** received his Mag. degree in 1984, and a PhD degree in business informatics in 1990, both from the University of Vienna, Austria.

From 1986 through 2000 he was with the Department of Applied Computer Science at the University of Vienna at the levels of assistant and associate professor. In 2000 he joined the University of Linz as full professor where he is now head of the Institut für Pervasive Computing. He has served on the committees of several conferences like Pervasive Computing, UMBICOMP, WWW, PADS, DIS-RT, SIGMETRICS, MASCOTS, TOOLS, PNPM, ICS, etc. and was the program chair for the PERVASIVE 2004 conference.

**Manfred Hechinger** received bakk. techn. in computer science from the Johannes Kepler University Linz, Austria in 2004.

He is currently working on his diploma thesis in the area of Peer-to-Peer Systems and employed in a research project on fully distributed Peer-to-Peer systems at the Johannes Kepler University Linz, Austria. Major interests are distributed computing, embedded systems and computer networks.

**Rene Mayrhofer** received his Dipl.-Ing. degree in computer science from the Johannes Kepler Universität Linz, Austria in 2002

He is currently working on his PhD on context prediction, also at Johannes Kepler Universität Linz. In 2002, he joined the Institut für Pervasive Computing as an assistant His research interests include context awareness, embedded systems, artificial intelligence with specialization on spiking neural networks and security. He has served on the committee of PERVASIVE 2004 and is a contributing developer of the Debian GNU/Linux project.

**Roy Oberhauser** received his B.S. in electronic engineering from California Polytechnic State University San Luis Obispo in 1990, and an M.S. in computer science from National Technological University, USA in 1999.

He is currently a senior research scientist for Siemens AG in the Software Architecture department, Software and Engineering, Corporate Technology in Munich, Germany. He has worked in the computer industry since 1990, including Zitel Corp., Network Peripherals, and Hewlett-Packard, and is named on four U.S. patents.

Mr. Oberhauser is on the program committee of ECOWS, NODE, and GSEM. His current research interests include software engineering, distributed computing, service-oriented architectures, web services, peer-to-peer, grid, domain-specific languages, platforms, and a plethora of related technologies.