BASE - A Micro-broker-based Middleware For Pervasive Computing

Christian Becker, Gregor Schiele, Holger Gubbels, and Kurt Rothermel

University of Stuttgart
Institute of Parallel and Distributed Systems (IPVS)
Breitwiesenstr. 20-22
70565 Stuttgart, Germany

{becker, schiele, rothermel}@informatik.uni-stuttgart.de
gubbelhr@rupert.informatik.uni-stuttgart.de

Abstract

Pervasive computing environments add a multitude of additional devices to our current computing landscapes. Specialized embedded systems provide sensor information about the real world or offer a distinct functionality, e.g. presentation on a "smart wall". Spontaneous networking leads to constantly changing availability of services. This requires middleware support to ease application development. Additionally, we argue that an extensible middleware platform covering small embedded systems to full-fledged desktop computers is needed. Such a middleware should provide easy-to-use abstractions to access remote services and device-specific capabilities. We present a micro-broker-based approach which meets these requirements by allowing uniform access to device capabilities and services through proxies and the integration of different interoperability protocols. A minimum configuration of the middleware can be executed on embedded systems. Resource-rich execution environments are supported by the extensibility of the middleware.

1. Introduction

Existing middleware platforms are characterized by their precautions to overcome heterogeneity of computer systems with respect to the hardware platforms and programming languages. However, the computer systems on which applications are executed are mostly homogeneous according to their processing and storage capabilities. The vision of ubiquitous or pervasive computing [31] creates a world populated not only by computers as we know them today but also with sensors and smart "everyday items". The heterogeneity added by these smart things is characterized by an additional property: the embedded systems integrated in the environment are typically tailored to distinct purposes. Hence, not only processing and storage capabilities differ widely but local device capabilities, such as different sensor types for temperature, pressure or positioning, are also device-specific. Communication between the different end-systems can take place over different kinds of network interfaces, such as infrared communication or radio links, e.g. Bluetooth or IEEE 802.11, and additionally via different interoperability protocols, such as IIOP, RMI, or simple event-based protocols.

The availability of resources, remote ones as well as local ones, can change over time, due to network connectivity as well as sensor-specific properties, e.g. it is unlikely that a GPS-based positioning system will work indoors.

In order to provide application programmers with support for conquering the additional complexity in pervasive computing environments, we have developed a micro-broker-based middleware. Our middleware will serve as a foundation for applications as well as component systems, hence the name BASE. Key features of BASE are the uniform access to remote services and device-specific capabilities, the decoupling of the application communication model and the underlying interoperability protocols, and its dynamic extensibility supporting the range of devices from sensors to full-fledged computers.

The paper is structured as follows. Next, we will motivate the requirements for such a middleware and introduce an example scenario. Existing approaches are classified and discussed in the related work section before we will sketch the overall design rationale of our approach BASE. Some implementation details of BASE and an evaluation will be presented before we close the paper with a conclusion and outlook on future work.

2. Requirements

In order to clarify our system model and derive our requirements, we want to sketch a small scenario. In a future "pervasive computing world", a building, e.g. an office, contains a huge number of highly specialized and therefore very heterogeneous computing devices. While some of them are stationary, e.g. placed in a room, others are carried by users, e.g. as wearable computers. Devices range from small embedded sensors to classic stand-alone computers. Clearly, the resources and capabilities of such devices differ widely, due to cost and size restrictions. Note, that the capabilities of a mobile device can also change dynamically. As an example, a GPS-sensor will stop functioning when entering a building. To summarize, a pervasive computing environment consists of a multitude of heterogeneous devices, both stationary and mobile, with different and dynamically changing capabilities and specific ways to access them.

One essential device capability is the ability to communicate and interact with other devices. This is achieved by forming spontaneous networks with changing members due to the communication range. Following [12] we prefer to use the term 'spontaneous' instead of 'ad-hoc' as ad-hoc tends to be restricted to specific lower level functionality like routing. The network interfaces used are highly heterogeneous ranging from infrared communication over radio links to wired connections. Interoperability protocols are tailored to specific requirements as well, e.g. a sensor does not need to implement a complex interoperability protocol but can simply emit its data periodically as events. To summarize, devices interact by forming spontaneous networks using different network interfaces and interoperability protocols. Membership in these networks is temporary and network related properties like communication cost and bandwidth change dynamically.

Distributed applications in this scenario are structured into application objects, or services, interacting with each other. Services in turn use device capabilities or further services, which are provided by either the local device, or by remote interaction with other devices. From the application's point of view, one of the main challenges is to use services and capabilities with changing availability. As we have seen, this is true for local, e.g. GPS, as well as remote cases, e.g. due to reachability. In addition, even a service that is both functional and reachable can become unavailable. Take for example a presentation system integrated into a video projector. If the user leaves the room, the presentation system becomes unavailable, because the user cannot see its output anymore.

Existing middleware platforms typically address portability of applications via standardized interfaces for remote service interaction, e.g. via stub and skeleton objects, and interoperability of applications across different middleware platforms via interoperability protocols. We derive three additional requirements:

1. **Uniform programming interface:** while classical middleware addresses uniform access to remote services the additional heterogeneity of specialized device capabilities requires similar abstractions, e.g. proxy objects, in order to access different device capabilites in a uniform way independent of the underlying platform.
2. **Flexible protocol support:** the service model of a middleware, e.g. remote procedure call or events, is typically reflected in its underlying interoperability protocol, e.g. using request/response messages or emitting event messages. The devices and systems in the above-mentioned scenario would need the integration of a variety of such service models which are reflected by their correspondent interoperability models. A decoupling of the service model from the interoperability model used by the middleware can help to bridge these interoperability domains. Additionally, this allows different communication paths for the incoming and outgoing messages. As an example think about two devices communicating via infrared in order to save energy. If the infrared link breaks due to obstacles or distance and a wireless radio link still exists, communication can continue. This can be either achieved by providing one interoperability protocol over different network interfaces or by the abstraction of different interoperability protocols which allows flexible usage of existing technologies.
3. **Tailorable:** To be useable on all kinds of devices found in future scenarios, the middleware has to be tailorable to the device at hand, a sensor device as well as a mainframe. The core functionality should be small enough to be executed on a sensor platform, but easily extensible to use the capabilities of resource richer devices.

Nowadays middleware platforms already provide high abstractions for programming distributed systems. Some platforms are already targeted to the above mentioned scenarios. The next section will discuss related work before we will present our approach.

3. Related Work

3.1. Conventional Middleware Systems

Device heterogeneity is not a unique characteristic of pervasive computing, but can be found in conventional systems, too. Different middleware systems like CORBA [19], Java RMI [8] or DCOM [6] have been developed to pro-

vide a homogeneous access to remote entities independent of e.g. operating systems or hardware architectures. Typically, these middleware systems try to provide as much functionality as possible, which leads to very complex and resource consuming systems, that are not suitable for small devices. Approaches to solve this problem exist and are discussed below. Conventional middleware systems are designed for mostly stable network environments, in which service unavailability is a rare event and can be treated as an error.

3.2. Dynamically Reconfigurable Middleware

Extending conventional middleware systems to dynamically reconfigurable middleware systems (e.g. [2]-[4], [13], [24], [25]) enables such middleware to adapt its behavior at runtime to different environments and application requirements, e.g. how marshalling is done. Still, different communication models or different protocols for outgoing and incoming messages are typically not supported. As one exception, the Rover toolkit [9] provides this functionality for its queued RPC (QRPC) concept, layered on top of different transport protocols. However, Rover only supports the QRPC and addresses potentially disconnected access to an infrastructure and not spontaneous networking.

A further difference from BASE is that most existing reconfigurable middleware systems concentrate on powerful reconfiguration interfaces and not on supporting small, resource-poor devices. A notable exception to this is UIC [25], which is discussed below.

3.3. Middleware for Resource-Poor Devices

The resource restrictions on mobile devices prohibit the application of a full-fledged middleware system. One way to address this is to restrict existing systems and provide only a functional subset (e.g. [18], [27], [28]) leading to different programming models or a subset of available interoperability protocols. Another option is to structure the middleware in multiple components, such that unnecessary functionality can be excluded from the middleware dynamically. One example is the Universally Interoperable Core (UIC) [25]. UIC is based on a micro-kernel that can be dynamically extended to interact with different existing middleware solutions. Still, the used protocol stack is determined before the start of the interaction and cannot be switched between request and reply as in BASE and abstractions are only provided for remote services.

3.4. Middleware for Pervasive Computing

Most pervasive computing middleware systems (e.g. [1], [5], [16], [22]) try to establish some kind of integrated, preinstalled technical infrastructure in a physical area, e.g. a room or building, often called an intelligent environment (IE), in which the user and his/her mobile devices are integrated on-the-fly when entering the area. The IE offers a huge variety of different capabilities and middleware services that can be used, once the device of the user is integrated.

As an example, the goal of the Gaia system [22] is to enhance physical spaces with computers to ActiveSpaces. Gaia provides an infrastructure to spontaneously connect devices offering or using services registered in Gaia. To integrate existing systems, like CORBA, interaction between application objects is done via the Unified Object Bus [23], which is layered on top of these systems. As essential system services, such as discovery and lookup, are provided by the Gaia infrastructure, mobile devices cannot cooperate autonomously without the infrastructure.

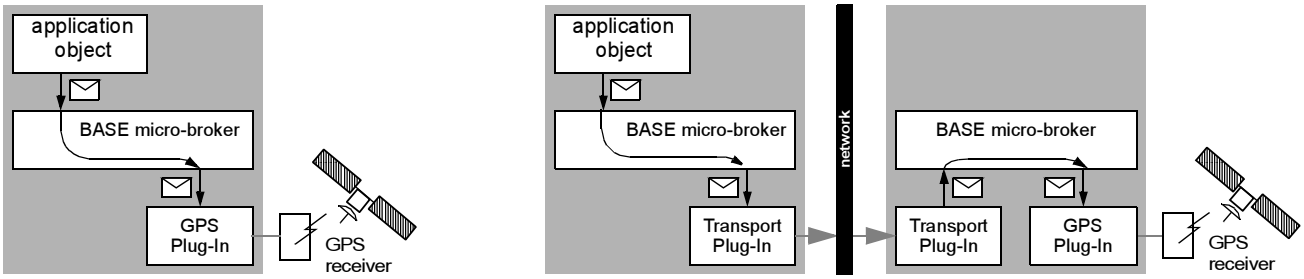
In contrast to this, we aim at supporting the cooperation of nearby devices, i.e. using only temporarily available hardware and software capabilities of nearby devices, independent of the presence of an external infrastructure. An infrastructure, such as an IE, may be included into a spontaneous network as temporarily available services, but the other way round - without the infrastructure - spontaneous networking requires additional support.

4. BASE

Before we describe the architecture and implementation of BASE, we first want to motivate our design rationale.

4.1. Design Rationale

One key idea behind BASE is the uniform abstraction of services as well as device capabilities via proxies as the application programming interface. Consequently, the middleware delivers requests to either device services in the middleware or transport protocols. Allowing different communication models with respect to the transactional pattern (request/response, event, synchronous, asynchronous, etc.) results in the middleware to provide the synchronization independent of the underlying protocols. Our approach is inspired by micro-kernels as they were introduced into the realm of operating systems (e.g. [21], [29]) and had some first applications in the middleware area as well (e.g. [20], [25]). Only minimal functionality, i.e. accepting and dispatching requests (so-called invocations), is located in the micro-broker. Interoperability protocols as well as object lifecycle management can be added as additional services, realized as plug-ins.



(a) accessing a local device capability

(b) accessing a remote device capability

Figure 1: Local and remote capability usage.

The micro-broker accepts requests represented as so-called invocation objects. In the following, we will refer to the invocation object when talking about an invocation. An invocation is composed of a source and a target address, an operation with parameters, and additional information concerning the handling of the invocation. The micro-broker dispatches the invocation to either a local service, a local device capability or a transport plug-in, which transports the invocation to a remote micro-broker. Transports which receive an invocation or a reply to a previous invocation – also represented by an invocation – submit them to the micro-broker to initiate the dispatching to the corresponding local service or device capability. Invocations can be either generated by proxies, representing a service or a device capability, or manually by the application programmer, e.g. like the request object in the dynamic invocation interface in CORBA [19]. Figure 1 depicts the micro-broker in a typical setting, where invocations are dispatched to (a) device capabilities and (b) transport plug-ins for the remote processing on other nodes. Remote service interaction follows the same pattern and is depicted in Figure 4.

Let us briefly argue why we have chosen this approach. Clearly, the requirement for uniform access of device capabilities as well as remote services can be easily established by our approach.

The micro-broker allows the flexible integration of new transport plug-ins and device capabilities by simply registering a new entity which accepts an invocation. This allows to provide access to all features available on resource-rich computer systems. The minimal functionality of the micro-broker itself allows the deployment of the middleware on resource-poor devices as well. To sum up, the uniform programming abstraction is provided by the service abstraction for remote service access and device capabilities. Together with the extensibility of the micro-broker this fullfils the first and third requirement that we have identified. The micro-broker allows in- and out-going messages over different transport protocols that can be dynamically loaded and configured through the invocation abstraction, which satisfies the second requirement. Although our implementation does not rely on reflection, the dynamic composable invocations along with the service registries provide means for reflection about services registered with the middleware.

The prototype of BASE is implemented in Java but relies only on features available in the Java Microedition. This allows the deployment on small Java-based embedded systems (e.g. [14]) or specialized Java processors (e.g. [11]). The proliferation of end-systems besides classical computers capable of executing Java, such as cell-phones or PDAs, and the aforementioned embedded systems make Java a suitable starting point providing a uniform abstraction for our middleware.

The benefit of our micro-broker approach compared to existing middleware platforms is the minimal footprint needed for a basic configuration which qualifies it for small embedded systems as well as the extensibility providing the means to use features of more sophisticated computers. The configurability that reflective middleware typically provides is also supported by BASE. A major difference to existing middleware platforms is the support of different communication models, such as RPC or events with different synchronization semantics, by the micro-broker, which allows these communication models over a variety of different interoperability protocols. Typically, the main communication model of a middleware is reflected in its interoperability protocols, e.g. CORBA's IIOP reflects the RPC by request/response messages. The BASE micro-broker only requires a transport plug-in to marshal and send an invocation. If responses are expected they may be received by any other transport plug-in.

4.2. BASE Architecture

Figure 2 depicts the overall architecture of BASE. Four layers are involved. The micro-broker is the central part of the system, consisting of the invocation broker and two registries for local services and devices which can currently be reached.

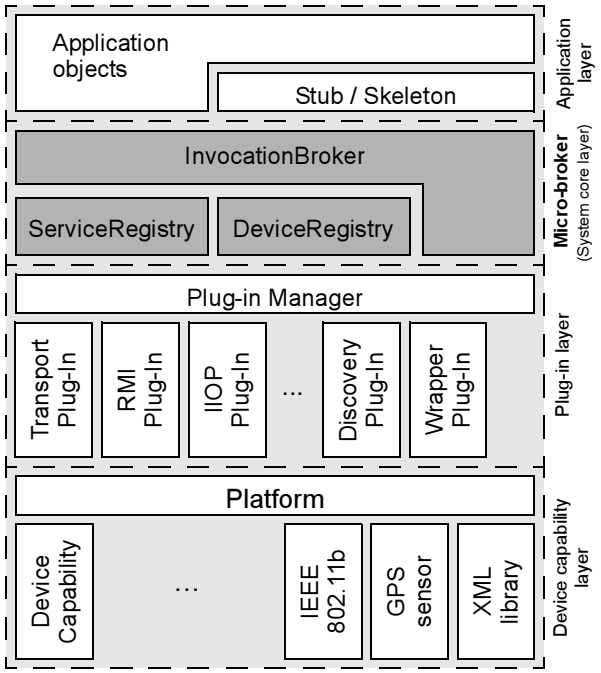


Figure 2: BASE architecture.

The micro-broker accepts invocations which are either manually assembled or generated by a stub-call. Additionally, an invocation can be used to access the registries for service lookups.

The plug-in layer maintains plug-ins which represent the entities capable of receiving invocations. Examples for plug-ins are transport protocols or encapsulations of device capabilities, such as sensor systems like positioning or temperature, or other services depending on the device, like input/output capabilities such as printing or video projection. Plug-ins typically involve interaction with the underlying operating system or directly with the hardware to offer access to a device capability or transport. The invocation broker accesses the plug-ins via invocations. Thus the underlying platform is encapsulated by the plug-ins. The device capability layer represents the device platform by its supported hardware and software.

In the remainder of this section the layers sketched above are discussed in more detail starting with invocations, the invocation broker, registries, stubs and skeletons, and the plug-in layer.

4.2.1. Invocation. Invocations are similar to dynamic invocation interface requests in CORBA. Figure 3 shows the elements of an invocation. Naturally, an invocation is represented as an object. Device and service IDs are used to denote a sender and receiver of an invocation. Services are given unique IDs that are local to a device. This ID is combined with a unique device ID to form a globally unique ID. The message IDs are needed for synchronization issues and are described in the paragraph discussing the invocation broker. A service context field allows the specification of additional parameters that indicate properties relevant to the processing of the invocation in the middleware such as synchronization issues or Quality of Service parameters. Basically, the context is a name-value list where parameters can be added freely. The payload contains the operations and parameters. In the case of event-based communication no receiver needs to be specified and the operation denotes the event-type on which applications can subscribe. The parameters then carry additional information of the event. In point-to-point communication the operations and parameters are interpreted as a remote method invocation.

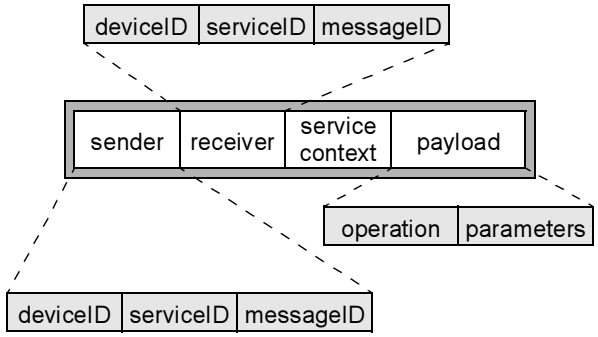


Figure 3: Invocation object structure.

4.2.2. Invocation Broker. Central to the system core, the invocation broker realizes the core functionality of the micro-broker. Invocations are accepted and dispatched. In order to separate the control flow between application and the processing of an invocation in a plug-in, a thread pool is maintained. Incoming calls are entered into the invocation table, assigned a message ID in order to identify parallel invocations of the same client. The context field contains, among other information, the communication model, i.e. synchronity and transactional pattern (request-response/ event) of the invocation. Depending on the communication model, the invocation broker blocks the incoming thread in case of a synchronous invocation. A new thread from the thread-pool is taken and the delivery of the invocation to the responsible plug-in (see below) is executed. After the plug-in has processed the invocation by either a local action, e.g. retrieving a sensor data, or a remote action, i.e. marshalling and sending the request to a remote peer, the thread returns and is added to the threadpool again. In case of a remote processing, an invocation may be sent back to the initial caller. The invocation broker receives the invocation from a plug-in for remote interaction, which may be different from the one that has processed the outgoing invocation, as shown in Figure 4.

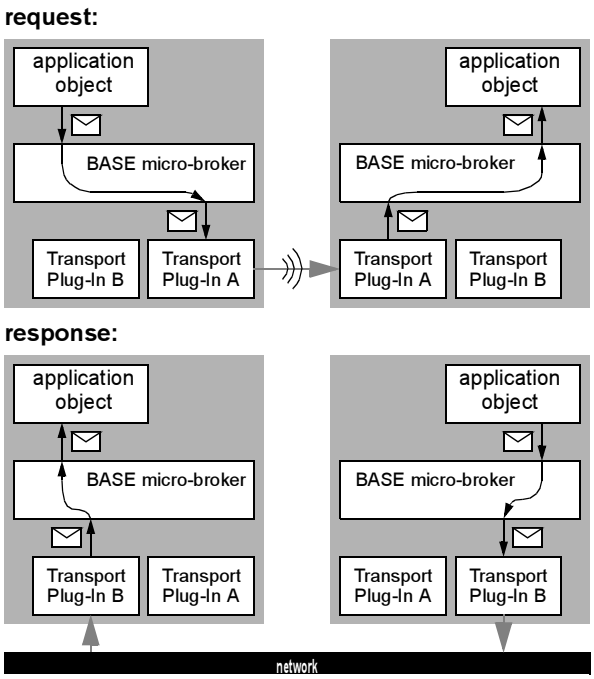


Figure 4: Request / response in BASE.

The invocation carries the target object and its message ID. If a message ID is contained in the receiver field of the invocation, this indicates that a caller is either blocked or awaiting an asynchronous delivery of the invocation. In case of a blocked call the waiting thread is freed and the invocation is provided as return. In the asynchronous case the invocation broker takes a thread from the thread-pool and calls up the application through a callback. In this case the message ID is used to designate the application callback registered at the invocation broker.

Notice that the explicit handling of synchronization depending on the communication model retrieved from the service context is a major design decision in BASE. This decouples the communication model from the underlying interoperability protocols. A request/response based communication model can be realized over two event-protocols as well as an event can be sent as a single request in an RPC-based interoperability protocol. An interaction can take place over different transport plug-ins for out-going and incoming invocations.

So far, BASE only supports a limited number of communication models, but an extension to different synchronization models, see e.g. [17], can easily be established with the underlying concept.

In order to determine the target of an invocation or to provide applications with service lookup two registries are maintained and described below.

4.2.3. Service and Device Registry . The service registry maintains all locally available services on a device. Services - as mentioned before - can be either application objects offering a service or device capabilities. Applications can query for available services by either specifying a name or the functional properties, i.e. the interface. Hence, a simple name and trading service is provided. Due to the nature of spontaneous networks, the availability of a lookup service cannot be assumed. The device registry maintains a list of all currently reachable devices and the transport plug-ins which provide the access to another device. If multiple transport plug-ins are possible for the same device, they are also entered into the list. This allows for a simple service lookup in the vicinity of a device. If a service request cannot be fullfilled locally, registries of nearby devices are queried and the result presented to the application.

The information of the device registry is also used by the invocation broker in order to determine which transport plug-in should be used. First, without any further information, any of the available transport plug-ins can be used. As long as there is a connection between two devices, i.e. the device is listed in the device registry and at least one transport plug-in is provided, invocations can be exchanged. Notice, that even if the transport plug-in by which a request invocation has been sent becomes unavailable replies can be received, if another transport plug-in exists. The service context sent with an invocation can be used to control the selection of specific transport plug-ins, e.g. in order to save energy or require a distinct bandwidth. We plan to extend this concept by strategies which will provide application-specific selection of transport plug-ins according to policies, e.g. energy awareness.

Although the current implementation of the service and device lookup is rather simple, the underlying concept is designed to be extensible allowing the integration of other lookup mechanisms, e.g. Jini [30] and UPnP [15].

4.2.4. Stubs and Skeletons. A common abstraction in middleware systems are local proxies for remote entities providing local access for application objects - stubs representing the remote service to clients and skeletons issuing local calls to services. In BASE, stubs and skeletons rely on the invocation abstraction. Stubs generate invocations upon method calls and skeletons generate local method calls upon a received invocation. Notice, that the generation of an invocation does not result in the marshalling of the parameters. This is a responsibility of the transport plug-ins. Invocations are used here to provide a common concept for interaction with the micro-broker. Applications can, however, omit the use of stubs and skeletons and compose and interpret invocations directly.

In contrast to systems like Jini [30], where stub and skeleton can include a service specific protocol stack this is not provided in BASE. Instead a service specific protocol

would be realized as a transport plug-in and thus become re-usable for other services as well.

4.2.5. Plug-In Manager. The plug-in layer is essential for the abstraction BASE presents to an application developer. Platform-specific capabilites, e.g. device capabilities and transports, are represented as plug-ins and become accessable to the application programmer as services. The plug-in manager allows the dynamic loading and integration of new plug-ins. Device capabilities are registered at the local service registry, and transport protocols at the invocation broker itself.

Plug-ins provide an abstraction of device-specific resources. Depending on the platform interface that allows the access of the device capability layer they can be portable among devices. Thus, an application on top of BASE will only interact via invocations, either dynamically constructed or generated by stubs, with device-specific capabilites.

Transport plug-ins are responsible for accepting an invocation, marshal it, and transmit it as a protocol data unit to a remote peer, which then constructs an invocation by demarshalling it. The simplest transport plug-in would use object serialization to marshal an invocation into a byte-buffer and send the buffer via a transport protocol, e.g. TCP/IP. Other transport plug-ins could rely on existing interoperability protocols and marshal and represent the invocation accordingly, e.g. map it to a request-message in IIOP and marshal the parameter by CDR, which allows interoperability with CORBA-based systems.

As long as the context of an invocation does not require a distinct transport plug-in, the invocation broker may use any transport plug-in to send an invocation to a remote device. The device registry maintains a list of all currently available transport plug-ins to a specific device. Hence, communication can take place as long as at least one transport plug-in allows the communication.

5. Implementation Status and Evaluation

This section will present the current status of our prototype implementation and discuss memory size and execution performance measurements.

5.1. Implementation Status

Our prototype has been implemented in Java to rely on its platform-independence. Although, for small devices C or C++ would seem to be a better choice at first, we found that Java allows us to run our middleware on a multitude of different devices, if the used Java features, like reflection, etc. are carefully restricted. A Tini minicomputer for example can execute only a subset of Java Version 1.1. Other devices, like smart phones or PDAs are limited to the Java Microedition [7].

So far, our prototype implements the basic concepts. Namely the invocation broker, the service and the device registry are implemented. The invocation broker handles different synchronization concepts and the service context is used to indicate the synchronization of RPC calls. For synchronous invocations, stub and skeleton support is implemented. Two transport plug-ins are realized so far, one based on the Java standard serialization mechanism on top of TCP/IP and a second based on Java RMI. Others are under way. The plug-in manager is implemented and allows the dynamic and static configuration of a BASE system.

5.2. Memory Size

The memory footprint of a minimal BASE configuration is crucial in order to allow the installation on small or embedded devices. We have measured the memory footprint of such a configuration, containing the micro-broker (invocation broker and registries) plus a TCP-based transport plug-in. The measurements where done using the IBM J9 implementation of the Java Microedition, more specifically the Java Microedition with the Connected Device Configuration and the Foundation Profile. First, in order to determine the memory footprint without additional dynamic memory consumption, i.e. BASE in idle mode, we use the Windows Task-Manager, as suggested in [32]. In this mode 132 KByte are used. During runtime, when invocations are exchanged, the system uses up to 420 KBytes, which was measured using the J-Sprint profiler [10].

5.3. Execution Performance Overhead

To measure the execution performance overhead introduced by the additional communication via the BASE micro-broker, we compared a BASE configuration sending invocations via a Java RMI transport plug-in with a pure Java RMI-based system. The measurements were conducted for a synchronous RPC communication by transmitting invocations for an operation testOperation, that takes a single string input parameter and returns immediately. The string size was either 0 or 1000 characters. This was done for local as well as remote invocations. The results are shown in Figure 5 and Figure 6. Each value given is the average of 12750 measurements. Measurement was done in 50 rounds with $roundnumber \times 10$ invocations per round, leading to a total number of $\sum_{i=1..50} 10 \times i = 12750$ measurements.

5.3.1. Local Invocations. In the local case, BASE is clear-

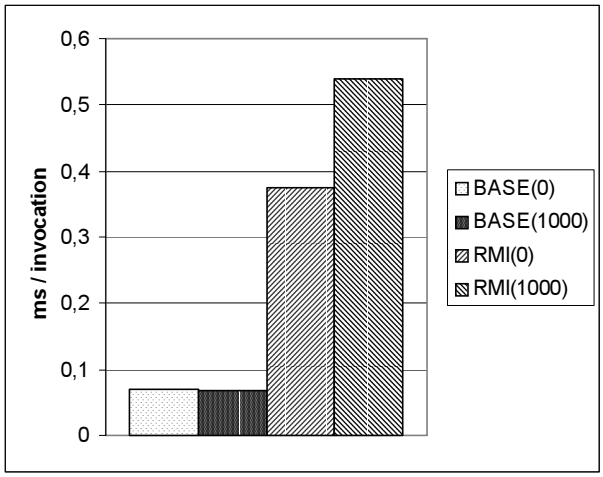


Figure 5: Local communication performance.

ly faster than RMI. This is due to the fact, that RMI in this case uses the loop-back interface including the RMI and TCP protocol stack while the BASE micro-broker forwards the call directly to the service skeleton and does not use the RMI-based transport plug-in at all.

5.3.2. Remote Invocations. In the remote case, BASE introduces an additional performance overhead of about 20%. Taking into account the creation of invocations from the stub objects and their interpretation by the skeletons, this seems acceptable. However, the absolute end-to-end latency measured for BASE is about 4 ms per remote invocation with a string size of 1000, which is rather long. Therefore, we did some additional measurements to compare this to the end-to-end latency of pure RMI, i.e. calling the remote operation directly through RMI without marshalling the invocation object. The pure RMI call only needed about 0,95 ms or 25% of the time BASE needed. This is due to the fact that we have used the standard Java object serialization mechanism in our prototypical RMI plug-in to marshal the invocation object. Note, that this is not a problem of the micro-broker itself, but of the current RMI plug-in implementation. Currently, other transport plug-ins are under development to overcome this performance bottleneck.

6. Conclusion and Future Work

We have presented the concept and design of BASE, a flexible middleware supporting the additional requirements of pervasive computing environments. Based on a micro-broker design, BASE allows minimal installations on embedded devices or specialized platforms as well as the integration of features available on resource-rich devices, such as personal computers. Application programmers

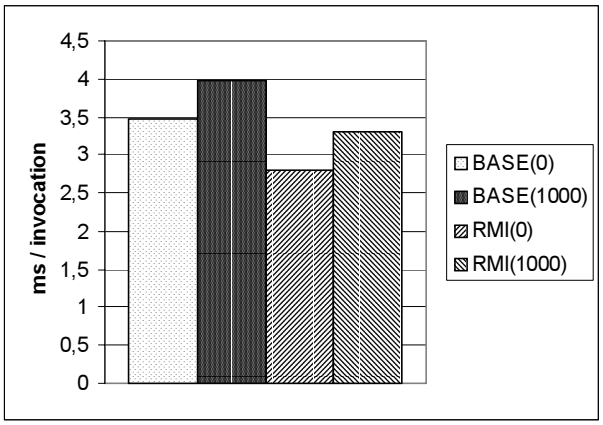


Figure 6: Remote communication performance.

can rely on a uniform abstraction to access remote and local services as well as device-specific capabilities. Thus BASE supports the portability of applications across heterogeneous devices. The middleware shields applications from the multitude of different communication technologies and interoperability protocols by separating the communication model of the application and the interoperability protocols used. This allows the usage of nearly arbitrary interoperability protocols.

The current implementation status of BASE is promising. Currently we are adding further support for different interoperability protocols and port BASE to some specialized devices. Further experience will be gained from doing protypical implementations of pervasive computing applications in our lab.

Using BASE as a middleware already will ease the design and implementation of applications. In further research directions we want to design a component system based on BASE that will support the adaptation of applications due to their execution environment. BASE will be extended by mechanisms to enforce adaptation strategies in the component framework, such as migration or service selection strategies. The extensibility of the micro-broker approach seems to be a good BASE here.

References

[1] M. Addlesee, R. Curwen, S. Hodges, J. Newman, P. Steggles, A. Ward, A. Hopper, "Implementing a Sentient Computing System", *IEEE Computer Magazine*, vol. 34, no. 8, pp. 50-56, August 2001

[2] C. Becker and K. Geihs, "Generic QoS-Support for CORBA", *Proceedings of the 5th IEEE Symposium on Computers and Communications* (ISCC'2000) Antibes, France, 2000

[3] G.S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R.

Moreira, N. Parlavantzas, and K. Saikoski, "The Design and Implementation of Open ORB version 2". *IEEE Distributed Systems Online Journal*, vol. 2, no. 6, 2001

[4] G. S. Blair, G. Coulson, P. Robin, M. Papathomas, "An Architecture for Next Generation Middleware", *Proceedings of Middleware 2000*, Lake District, UK, 2000

[5] M.H. Coen, B. Phillips, N. Warshawsky, L. Wiesman, S. Peters, P. Finin, "Meeting the Computational Needs of Intelligent Environments: The Metaglue System", *Proceedings of the 1st International Workshop Managing Interactions in Smart Environments* (MANSE'99), Dublin, Ireland, pp. 201-212, December 1999

[6] G. Eddon, H. Eddon, *Inside Distributed Com*, Microsoft Press, February 1998

[7] Java Micro Edition Homepage, http://java.sun.com/j2me/

[8] Java Remote Method Invocation Specification. Revision 1.8, Sun Microsystems, available online: http://java.sun.com/j2se/1.4/docs/guide/rmi/index.html, 2002

[9] A.D. Joseph, J.A. Tauber, and M.F. Kaashoek, "Mobile Computing with the Rover Toolkit", *IEEE Transactions on Computers: Special issue on Mobile Computing*, vol. 46, no. 3, pp. 337-352, March 1997

[10] J-Sprint Homepage, http://www.j-sprint.com/

[11] JStamp Homepage, http://jstamp.systronix.com/index.htm

[12] T. Kindberg, A. Fox, "System Software for Ubiquitous Computing", *IEEE pervasive computing*, vol. 1, no. 1, pp. 70-81, January-March 2002

[13] T. Ledoux, "OpenCorba: A Reflective Open Broker", *Proceedings of the 2nd International Conference on Reflection* (Reflection'99), pp. 197-214, Saint-Malo, France, 1999

[14] D. Loomis, *The TINI(tm) Specification and Developer's Guide*, Addison-Wesley, June 2001

[15] Microsoft Corporation, *Universal Plug and Play Device Architecture, Version 1.0*, http://www.upnp.org/download/UPnPDA10_20000613.htm, June, 2000

[16] M. Mozer, "The Neural Network House: An Environment that Adapts to its Inhabitants", *AAAI Spring Symposium*, Stanford, pp. 110-114, March 1998

[17] Object Management Group, *CORBA Messaging*, report orbos/98-05-06, 1998

[18] Object Management Group, *Minimum CORBA Specification*, Revision 1.0. August 2002

[19] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 3.0. July 2002

[20] A. Puder, K. Roemer, *MICO: An Open Source CORBA Implementation*, 3rd edition, Morgan Kaufmann Publishers, March 2000

[21] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, M. Jones, "Mach: A System Software kernel", *Proceedings of the 34th Computer Society International Conference* (COMPCON 89), San Francisco, CA, February 1989

[22] M. Román and R.H. Campbell, "GAIA: Enabling Active Spaces", *Proceedings of the 9th ACM SIGOPS European Workshop*, pp. 229-234, Kolding, Denmark, September 2000

[23] M. Román and R.H. Campbell, "Unified Object Bus: Providing Support for Dynamic Management of Heterogeneous Components", *Technical Report UIUCDCS-R-2001-2222 UILU-ENG-2001-1729*, Universiy of Illinois at Urbana-Champaign, 2001

[24] M. Román, F. Kon and R.H. Campbell, "Design and Implementation of Runtime Reflection in Communication Middleware: The DynamicTAO Case", *Proceedings of the 1999 ICDCS Workshop on Electronic Commerce and Web-Based Applications*, pp. 122-127, Los Alamitos, CA, 1999

[25] M. Román, F. Kon, and R.H. Campbell, "Reflective Middleware: From Your Desk to Your Hand", *IEEE Distributed Systems Online Journal, Special Issue on Reflective Middleware*, July 2001

[26] M. Román, D. Mickunas, F. Kon and R.H. Campbell, "LegORB and Ubiquitous CORBA", *IFIP/ACM Middleware'2000 Workshop on Reflective Middleware*, NY, April 2000

[27] M. Román, A. Singhai, D. Carvalho, C. Hess, and R.H. Campbell, "Integrating PDAs into Distributed Systems: 2K and PalmORB", *International Symposium on Handheld and Ubiquitous Computing (HUC'99)*, Karlsruhe, Germany, September 1999

[28] D.C. Schmidt, *Minimum TAO*. http://www.cs.wustl.edu/~schmidt/ACE_wrappers/docs/minimumTAO.html

[29] A.S. Tanenbaum, M.F. Kaashoek, R. van Renesse, and H. Bal, "The Amoeba Distributed Operating System-A Status Report", *Computer Communications*, vol. 14, no. 6, pp. 324-335, July/August 1991

[30] J. Waldo, "The Jini Architecture for network-centric computing", *Communications of the ACM*, vol. 42, no. 7, pp. 76-82, July 1999

[31] M. Weiser, "The computer for the 21st century", *Scientific American*, vol. 265, no. 3, pp. 94-104, September 1991

[32] S. Wilson, J. Kesselman, *Java Platform Performance: Strategies and Tactics*, Addison-Wesley, May 2000