Experiences: Minimalism and Extensibility in BASE

Marcus Handte, Christian Becker, Gregor Schiele

Institute for Parallel and Distributed Systems (IPVS)
Universität Stuttgart, Germany
{marcus.handte|christian.becker|gregor.schiele}@informatik.uni-stuttgart.de

Abstract. In the vision of Ubiquitous Computing everyday objects become smart. Technically, this requires some sort of processing and communication technology. We have designed and implemented a middleware for spontaneous networking in Ubiquitous Computing environments. The major objectives were minimalism and extensibility in order to deploy the middleware on a variety of devices ranging from sensor nodes to classical general purpose computers. In this paper we will assess the taken approach based on two follow-up projects: the port of BASE to a small embedded system and the design and implementation of a component system on top of BASE. While the fundamental concepts and design principles of BASE have proven to be solid, both projects provided insights that led to minor conceptual and major technical changes.

1 Introduction

Ubiquitous Computing (UC) [10] envisions spontaneous interaction of computerized devices in order to achieve complex goals and support people's tasks. As in ordinary distributed system settings, interaction is achieved through the exchange of data and therefore is based on mechanisms that enable communication of computer systems. Support for communication in UC environments faces challenges that go beyond those of systems in static environments. Apart from the heterogeneity of devices which, to some degree, can also be found in ordinary distributed systems, UC is based on networks that form spontaneously and change dynamically. The mobility of devices makes it inevitable, that devices integrate in their ever-changing surrounding networks in order to utilize the functionality provided by them.

Resulting from the need to enable communication between heterogeneous computer systems in dynamic environments, a number of infrastructures have been proposed. These infrastructures are designed to provide an easy and efficient way of building and executing applications for ubiquitous computer systems. Depending on the degree of device mobility anticipated, they can be classified into two categories. The first category of infrastructures is based on the concept of smart environments. Prominent examples are Gaia [8], Aura [5] and iRos [7]. They provide means to integrate small, mobile devices into relatively heavy weight environments with the immense processing power and storage capacities of today's desktop systems. The second category of infrastructures is targeted at supporting mobile devices with limited resources without relying on the processing power or storage capacity of the envi-

ronment. Two representatives of this category are RCSM [11] and BASE [3], a middleware that supports spontaneous communication between devices. BASE has been designed to support a wide range of devices from sensor platforms to general purpose computers. Its micro-broker architecture allows the creation of a portable system with minimal hardware requirements, but it makes extension mechanisms inevitable in order to optimally utilize the capabilities of different devices.

In this paper we present our experiences with porting BASE to a JStamp processor [9], a Java-based embedded system supporting only the Java 2 Micro Edition [6] in the Connected Limited Device Configuration (CLDC). Further experiences where gained when we designed and implemented a component system for UC on top of BASE. Our experiences so far are promising. Both projects together enabled a first evaluation of minimalism and extensibility of BASE and led to optimizations regarding the internal mechanisms and external abstractions provided by this middleware.

The remainder of this paper is structured as follows. Next, we will present an overview of BASE's architecture. Section three briefly describes the projects that led to the experiences described in this paper. In the forth section we will discuss the problems that we have encountered, their solutions and lessons learned. Section five summarizes and concludes the paper.

2 BASE – A Micro-broker Based Middleware

In order to understand the approach taken during the design of BASE's architecture, it is necessary to explain the underlying requirements and design rationales. As a complete description would go beyond the scope of this paper we only present a brief overview before presenting the architecture. A more detailed discussion of the requirements can be found in [2] while BASE is described in [1], [3] in more detail.

2.1 Design Rationales

BASE was designed to fulfill three major requirements. First, application programmers should be provided with a *uniform programming interface* for accessing device capabilities, like a GPS receiver, and application objects, both, local and remote ones. This allows transparently switching functionality at runtime or more general, adapting to changes in the availability of functionality in a uniform way, e.g. by switching to a remote location service once the GPS receiver stops operating indoors. Therefore, in BASE, a service abstraction is provided to the application programmer to access device capabilities and application objects.

Second, the variety of different devices will likely lead to a number of different interoperability protocols with different communication models, e.g. events, remote procedure calls (RPC), etc. These should be *decoupled* by the middleware *from the application communication model*. This allows for example using an event-based interoperability protocol to deliver request/response messages of an RPC.

Last but not least, the middleware should be *minimal* and *tailorable*. This allows the installation on resource restricted devices, e.g. sensors, as well as using resources on more powerful devices, such as presentation systems or desktop computers.

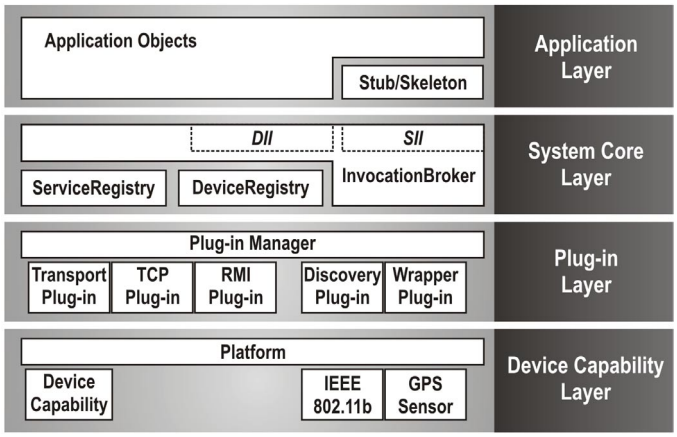

Figure 1. BASE Architecture

2.2 Architectural Overview

The architecture of BASE is depicted in figure 1. BASE offers application programmers a static (SII) and a dynamic invocation interface (DII). For the SII, stubs and skeletons are generated by a compiler and are used to map a method call to/from a so-called invocation object. If the DII is used, the application composes invocation objects directly. Invocation objects are Java objects, containing the *unmarshalled* invocation parts, like method name and parameters as well as further information on how to thread the invocation, e.g. which synchronization pattern should be used. While marshalling typically is a stub/skeleton responsibility, it was omitted on this layer and pushed down to the transport plug-ins to give the middleware maximum flexibility in choosing a suitable interoperability protocol at runtime.

In the system core layer, the invocation broker is responsible for delivering the invocation to either a local device capability or a remote service by choosing an appropriate plug-in. The invocation broker relies on information from the service registry (local services) and the device registry (currently reachable devices and the corresponding transport plug-ins) in order to dispatch an invocation. Since plug-ins can realize arbitrary protocols the invocation broker has to synchronize the invocation according to the application programming model and the underlying plug-in.

Plug-ins can be dynamically loaded and thus allow the extensibility of the middleware. The invocation broker follows the micro-kernel philosophy by only offering minimal functionality, i.e. how to find a service responsible for the invocation, dispatch it, and synchronize the invocation according to the application communication model. Thus, we call it a micro-broker.

Since all plug-ins, i.e. for device discovery, device capability, and transports rely on the same interface, i.e. handle invocations, applications can use the same programming interface (SII, DII) to access them. As stubs and skeletons do not provide any marshalling functionality, transport plug-ins have to ensure the marshalling of parameters and construction of interoperability protocol messages.

2.3 First Experiences

The first prototype of BASE was developed using an IBM J9 implementation of the Java 2 Micro Edition with the Connected Device Configuration (CDC). The CDC omits a variety of features from the Standard Edition, e.g. reflection, while others, such as object serialization, are present. Initial measurements [3] showed a reasonable small memory footprint of about 130 Kbytes but also that the initial marshalling resulted in two to three times overhead compared to Java RMI. This overhead mostly resulted from the naïve approach taken, i.e. serializing an invocation object with Java's object serialization.

3 Porting and using BASE

After the initial prototypical implementation that built upon the J2ME CDC platform, we started two projects related to BASE. One project ported BASE from its original platform the JStamp. The other project aimed at the development of a component system on top of BASE. The combined experiences created a picture that allowed an initial evaluation of both, the internal structure and the external abstractions.

3.1 Porting BASE

Although BASE is targeted at systems of all sizes we decided not to deal with all complexities that arise from the application of extremely restricted platforms during the development of the first prototype. Therefore, we did not build upon the most restricted platform defined by the J2ME specification. Instead we used the CDC, since it has a range of advanced features that allowed us to speed up the initial development. These features included for instance, JVM support for object serialization and dynamic class loading. The typical hardware that provides CDC sized runtime environments are high-end personal digital assistants or TV set-top boxes. Clearly, UC aims at devices that are even smaller. Therefore, we began to port BASE to the CLDC shortly after the first prototype was built successfully. The CLDC is targeted at devices including low-end personal digital assistants and embedded processors. Porting BASE required two tasks. First, we had to remove or reconstruct all convenient features that were solely available on CDC enabled systems. Second, we had to build platform specific transport and discovery plug-ins, since the JStamp processor did not support our existing IP-based transport and discovery plug-ins. Both tasks together gave us a chance to evaluate the internal structures when porting BASE to other platforms.

3.2 BASE as a Platform for Components

BASE aims at abstracting from platform specifics, but it leaves application programmers with only basic support, when dealing with fluctuating availability of local

and remote services. As these fluctuations are inherent in mobile ad hoc networks, code of stable applications is necessarily tangled with code that manages dependencies on functionality provided by services. Since this kind of tangled code raises the complexity of application development, we decided to automate dependency management by the middleware using a component abstraction. The resulting component system used BASE as means of communication. Since we did not want to change the main mechanism and abstractions provided by BASE during its development, the component system can be seen as an application built on top of BASE. Therefore, this project enabled us to evaluate BASE's external structures that are used during application development.

4 Experiences

Before we present the lessons learned from conducting the port and the development of a component system, we will describe the resulting modifications to BASE. The modifications can be divided into two classes depending on their effects. The first class has been foreseeable and did not have conceptual impact. The second class is more interesting as it affects the fundamental concepts of BASE.

4.1 Technical Modifications

The additional restrictions imposed by the CLDC led to technical issues that could be resolved in a straight forward manner. Most noteworthy we were facing the following difficulties:

Class loading: the initial version of BASE made use of dynamic class loading in order to locate and execute plug-ins and services at runtime. As dynamic class loading is very restricted by the CLDC, we had to reduce this flexibility. Instead of dynamic class loading we modified BASE to use linked classes. We simplified the resulting more complex configuration process by providing a graphical configuration tool that generates desired configurations.

Object serialization: the CLDC does not provide means for serialization of objects. Since plug-ins are responsible for the marshalling, the first prototype of BASE simply serialized the invocation object. As mentioned before, this resulted in an unnecessary overhead and additionally, it was not possible on the CLDC. Our solution to this problem is straight forward. Via a serialization interface the marshalling code can access the object's state and write/read it to/from an output/input stream. We will later describe a solution for a more flexible and performance oriented plug-in structure.

4.2 Conceptual Modifications

BASE's plug-in concept offers a rather coarse grained structure currently including marshalling, interoperability, discovery, and transport layer abstractions. As the JStamp did not support our existing transport and discovery plug-ins, we had to develop new plug-ins. Although developing plug-ins is a fairly simple undertaking, due

to their coarse grained structure, we were not able to reuse much of the existing code. Along with the marshalling performance mentioned earlier and current activities for QoS management, we have to conclude that the plug-in concept so far provides suitable abstractions to interface to the micro-broker but requires additional structuring into an interoperability framework. Optimized marshalling code for distinct interfaces, service discovery, as well as transport layer related issues, e.g. SSL encryption, can be integrated via interceptors offering a simple configuration and re-use of these elements in other plug-ins.

Apart from the technical modification described earlier, the inability to load classes dynamically also led to conceptual changes. Just like JINI [4] services, BASE services were designed to provide stubs for their clients. The automated delivery of stubs allows service-instance specific stubs and skeletons, but it relies on the ability to load classes dynamically. Porting BASE led to the conclusion that, due to its overall architecture, service-instance specific stubs and skeletons are an unnecessary feature. With respect to JINI services, loadable stubs are the only way to support flexible communication mechanisms. While BASE decouples stubs from the specifics of the transport and interoperability layers, JINI's stubs cut right through all communication layers. Therefore, JINI clients have to use the stub provided by the service. Otherwise they will not be able to create valid requests. The only functionality provided by BASE's stubs is the creation of Invocations. Encoding and transmission of data is handled by plug-ins. As a result, clients are able to include stubs for all services that they might use. The fact that BASE does not need service-instance specific stubs and skeletons results in a leaner ServiceRegistry.

4.3 Lessons Learned

From conducting both projects we learned a lot about the design decisions made during the initial development of BASE. A very obvious lesson that can be learned is that porting a Java-based system is not always as simple as some people claim. Although Java is usually considered to be a platform independent language, switching to a more restricted J2ME configuration can lead to costs that are comparable to the costs of porting platform dependent programs. Both, the lack of object serialization and dynamic class loading required the design of new mechanisms to achieve a similar level of convenience.

Apart from the platform related issues, the conceptual modifications provided two interesting insights. First, we learned that it is possible to use our plug-in concept to successfully build plug-ins for small devices. At the same time, we discovered that the granularity of the plug-in layer is not yet satisfactory. Therefore we have to conclude that the plug-in concept offers the required extensibility, but it needs a more sophisticated structure to increase reuse of existing code and to provide improved support for developers.

The second modification showed that the plug-in architecture allows removing service specific stubs and skeletons without loss of functionality. The extensibility provided by BASE's plug-in layer is sufficient to achieve at least the same degree of flexibility as systems like JINI.

The previously discussed lessons can be derived directly from modifications, but there are also lessons learned that result from keeping existing concepts. For example, one interesting feature of BASE that did not change during the projects is its reflection mechanism. In contrast to the Standard Edition, J2ME does not support reflection. However, in the presence of dynamic invocation creation and appropriate means to specify services and their interfaces via the service registry, a simple reflection mechanism is provided by BASE. It was an ongoing discussion in the team whether to aim for general reflection, i.e. storing signatures and class-relations in the service registry, or only providing interface names and the class-hierarchy information. So far, we have chosen the latter approach without experiencing any restrictions. Our component system provides more powerful concepts for interface description and exploration including non-functional parameters and hence we decided to keep BASE minimal.

Another and probably the most important lesson that we have learned is also a result of not changing anything. During the development of the component system, there was no need to modify BASE. All necessary additions were implemented in the application layer. Only two extensions were integrated directly into BASE. First, components managed by the component system use stubs and skeletons that inherit from the original stubs and skeletons provided by BASE. This enables a faster dispatch of messages since there is no additional indirection in the dispatch chain. Second, some of the functionality provided by the Registries is accessed directly in order to remove indirections that might have negative impact on the performance of the system. Note, that these design decisions are performance optimizations. We could have done everything in the application layer (although this would have led to a much slower system). This brings us to the conclusion that BASE provides suitable abstractions for implementing applications as well as high-level infrastructures.

The successful development of the component system also raised questions. Our preliminary evaluation indicates that the overhead caused by a carefully designed component system is reasonably small compared with the initial cost of using BASE. The current version of BASE requires 90KB. Through the usage of the component system, these requirements are increased by 30KB. Considering target systems like the JStamp that have at least 1MB of memory, we are currently considering whether it makes sense to completely abandon the service abstraction and use components instead. But at the moment it is too early to fully assess all consequences of such a move.

5 Conclusions

In this paper we have presented our experiences with conducting two projects that build upon BASE. While the internal structures have undergone technical and conceptual modifications, the external structures stayed remarkably stable. The conceptual modifications led to a follow up project, in which we began to design an improved plug-in layer to overcome the described deficiencies. Furthermore, we were able to successfully port BASE to a new set of target devices and to utilize it for a larger application. This success is encouraging and it shows that BASE is not only suited for

smaller devices, but also that it can be used as infrastructure for applications as well as for further high-level abstractions. We are highly confident that the minimalism of our micro-broker approach together with the extensibility of its plug-in architecture will prove to be adequate for UC environments.

BASE and the component system are freely available to research institutions and can be downloaded at http://www.3pc.info.

References

1. Becker, C., Schiele, G.: BASE: A Minimal yet Extensible Platform for Pervasive Computing, International Conference on Tales of the Disappearing Computer, Santorin, Greece, 2003
2. Becker, C., Schiele, G.: Middleware and Application Adaptation Requirements and their Support in Pervasive Computing. 3rd International Workshop on Distributed Auto-adaptive and Reconfigurable Systems (DARES) at ICDCS, pp. 98-103, May 19-22, Providence, USA, 2003
3. Christian Becker, Gregor Schiele, Holger Gubbels, Kurt Rothermel: BASE - A Micro-broker-based Middleware For Pervasive Computing. In Proceedings of the First IEEE International Conference on Pervasive Computing and Communication (PerCom), pp. 443-451, March 23-26, Fort Worth, USA, 2003
4. Edwards, W.K.: Core JINI. The SUN Microsystems Press Java Series, Prentice Hall, 1999
5. Garlan, D., Siewiorek, D., Smailagic, A., Steenkiste, P.: Project Aura: Towards Distraction-Free Pervasive Computing. IEEE Pervasive Computing, vol.1, no.2, pp.22-31, April-June 2002
6. Java Micro Edition home page, http://java.sun.com/j2me/
7. Johanson, B., Fox, A., Winograd, T.: The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. IEEE Pervasive Computing, vol.1, no.2, pp.67-74, April-June 2002
8. Román, M., Campbell, R.: Gaia: Enabling Active Spaces. In Proceedings of the 9th ACM SIGOPS European Workshop, Kolding, Denmark, pp. 229-234, September 2000
9. Systronix Inc home page, http://www.jstamp.com/
10. Weiser, M.: The Computer for the Twenty-First Century. Scientific American, vol.265, no.3, pp.94-104, September 1991
11. Yau, S.S., Karim, F., Wang, Y., Wang, B., Gupta, S.K.S.: Reconfigurable Context-Sensitive Middleware for Pervasive Computing. IEEE Pervasive Computing, vol.1, no.3, pp.33-40, 2002