Experiences with MundoCore

Erwin Aitenbichler Jussi Kangasharju Max Mühlhäuser
Telecooperation Department
Darmstadt University of Technology, Germany
{erwin,jussi,max}@tk.informatik.tu-darmstadt.de

Abstract

Pervasive Computing environments require new communication and programming paradigms. In this paper we present our experiences from the implementation of MundoCore*, a pervasive communication middleware. MundoCore's service discovery allows us to discover all available services in the nearby environment, as our experimental results demonstrate. MundoCore offers different programming paradigms, such as DOOP and Publish/Subscribe. We show how a simple room-control application can be programmed with the different paradigms, and we highlight the advantages of the Publish/Subscribe paradigm.*

1 Introduction

Pervasive computing environments introduce new challenges for communication architectures. Highly dynamic environments require efficient mechanisms for discovering devices and services, as well as efficient methods for communication between them. Such environments also require support for the programmer, in the form of new programming abstractions for the new communication paradigms.

Consider a meeting, where there are several people, each with her own devices. Usually, the participants will need to interact with each other or the environment (e.g., exchange files, or print something to the local printer). The fundamental step to enable these interactions is the discovery of all available devices and services in the area.

In this paper we present our experiences in implementing MundoCore, a pervasive computing middleware. We explain our distributed test architecture used to verify the correct behavior of the middleware core and its services. MundoCore features a reliable discovery of all nearby devices and services, and provides the programmer with useful programming abstractions, e.g., Publish/Subscribe. We present an example of the implementation of a room-control application, both as a traditional, RPC-based implementation (as many programmers would likely do!), and also as a MundoCore Publish/Subscribe implementation which enables a cleaner and more feature-rich implementation.

The paper is organized as follows. In Section 2 we give a short introduction into our middleware, followed by a brief description of Related Work in Section 3. In Section 4 we describe our architecture to run distributed tests. Section 5 discusses the discovery problem and we evaluate discovery in MundoCore. In Section 6 we report experiences with users and conclude the paper in Section 7.

2 MundoCore

Based on a structured view of pervasive distributed systems – the Mundo [4] Reference Architecture – we have derived requirements for a communication middleware. These can be clustered into the three following blocks:

Adaptivity to networks and platforms: In an environment where networks will be formed spontaneously, hosts must be able to configure themselves automatically and adapt to changing conditions. For example, Bluetooth devices often require the user to perform a pairing procedure with manual interaction. This will not scale to thousands of sensors and computing devices expected to be embedded in smart environments. Furthermore, devices must be able to operate in isolation and cannot rely on an infrastructure to be present. Many pervasive computing devices have only very limited resources in terms of memory and processing power, hence it is vital that the middleware is modular and has a small memory footprint.

Multi-paradigm: The middleware should also be extensible in terms of programming abstractions, allowing programmers to choose the most appropriate abstraction for the task at hand. (See Section 6 for details.)

QoS: Especially in wireless networks, many application domains require end-to-end quality of service (QoS) guarantees. The fixed Internet today is mainly based on a best-effort strategy for all kinds of applications, because network bandwidth is available in excess. Wireless bandwidth is likely to remain a scarce resource in the near future for the following reasons. First, wireless bandwidth is growing at

a much slower speed than processor capacity. Second, high bandwidth implies high energy consumption or short radio range (i.e., small cells). Third, coverage and quality of wireless networks is highly heterogeneous.

3 Related Work

Reconfigurable middleware (e.g. [1,5,6,8]) can adapt its behavior to different environments and application requirements. Such middleware has additional interfaces allowing applications to change the strategies for concurrency, request demultiplexing, scheduling, marshaling and connection management. In addition, monitoring is supported to detect when adaptation should take place. Most of these systems are focused on powerful reconfiguration interfaces rather than on a fine-grained modular structure suitable for resource-poor devices. Furthermore, they do not address spontaneous networking. Notable exceptions are modular middleware approaches that specifically target Pervasive Computing applications, like BASE [1] and UIC [8].

JXTA [7] technology is a set of open protocols that enable any connected device on the network, ranging from cell phones and wireless PDAs to PCs and servers, to communicate and collaborate in a P2P manner. However, JXTA comes with its own programming abstractions, and current implementations lack ORB and QoS support.

4 Testing distributed applications

Implementations of abstract data types and smaller units of frameworks can be successfully verified with unit tests. However, to verify the correct behavior of a distributed computing middleware, it is also essential to run tests with distributed applications. To conduct such tests, we have implemented a distributed script interpreter. A typical setup is shown in Figure 1. Script server processes are manually started on different hosts in the network. The script servers and the master script interpreter are also based on MundoCore. This enables them to automatically discover each other on startup.

To run a test, the name of an XML script file is passed to the master interpreter. When the master interpreter encounters a `execRemote` instruction, it starts a new process on a remote script server and passes the text contained in the tag to the standard input of the remote process. The standard output of the remote process is passed back from the remote script server to the master interpreter and written to a log file. The shell return codes indicate if processes were successful running their sub-tests. The master interpreter considers a test run as successful, if all started sub-processes terminated with the return code 0.

If the test processes are also based on MundoCore, script server and test processes form two independent connection

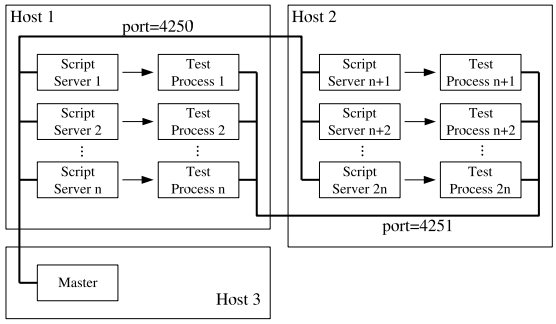

Figure 1. Scripting architecture

graphs, as shown in Figure 1. There are no direct network links between these two different kinds of programs. It is particularly important that test processes form their own, independent overlay network from scratch when testing automatic node and service discovery.

5 Discovery

In this section, we consider the so-called discovery problem. By this we mean the problem of finding all the available devices and services in the nearby environment. Our focus is on discovering all the nearby nodes in a reliable manner which will allow us also to discover all the services on those nodes. The *node discovery problem* is based on the following assumptions:

- We consider spontaneous networks without any centralized service registries or caches.
- If devices share a physical network connection, then all services offered by them should be discoverable.
- In case of wireless network connections with high packet loss, discovery should have an all-or-nothing semantic. Either a node is fully integrated into the group, or it does not get any access to remote services.

When programming ubiquitous computing applications, it is desirable to build on top of a middleware that provides reliable discovery of other network nodes and services. Our aim is to give certain guarantees to applications regarding quality of service discovery. The middleware should compensate for imperfect behavior of the network and operating system layers. Discovery protocols often use UDP packets sent via IP broadcast or multicast. Networks do not guarantee correct delivery of UDP packets - they might get lost. Loopback network drivers of certain operating systems virtually drop all broadcast packets. This makes it impossible for local processes to discover each other, if no real network connection is present.

The following discussion focuses more on the practical applicability of existing frameworks rather than on theoretical properties of algorithms and protocols. MundoCore has been specifically optimized to deal with the discovery problem outlined above.

5.1 Discovery in MundoCore

Discovery in MundoCore is based on three concepts. First, a node joining the network announces its presence with IP broadcast messages. Second, the rendezvous via the *primary port* ensures that all local processes are able to discover each other. Third, *neighbor messages* are used to propagate information about new nodes in the network.

One MundoCore process is always listening on the *primary port* which is a well known TCP port defined in the configuration file. All nodes that participate in the same overlay network are configured with the same primary port number. When a MundoCore process is started, it tries to allocate the primary port. This will fail if another MundoCore process is already running on the same machine. In that case, the process allocates any free port for listening and connects to the primary port on the local machine. If the connection to the primary port breaks, this means that the process that held the primary port has shut down. When this happens, the current process tries to allocate the primary port. Thus, if at least one MundoCore process is running on a machine, a process will listen on the primary port.

This discovery scheme is reliable and was necessary, because of the following issues with broadcast discovery:

- The behavior of UDP broadcasts over loopback network interfaces is undefined. Especially on Windows platforms, UDP broadcasts are often not forwarded to local processes. Thus, local processes cannot discover each other in absence of a real network interface.
- Limited scope of broadcasts. Broadcasts are usually used only for the local subnet or are often filtered out by gateways and firewalls. In that case, a node can explicitly connect to a remote MundoCore node if it knows the hostname or IP-address. Because a MundoCore process is always listening on the primary port, it can discover all nodes running on the remote machine.

MundoCore implements the following features to cope with the described problems:

Zero configuration: MundoCore features true zero-configuration in cases where all processes share a network connection. Port numbers are automatically assigned.

Reliable local discovery: Discovery of processes on the local machine can entirely rely on reliable (TCP) connections in MundoCore. This enables discovery to work properly even when the loopback interface driver does not properly forward broadcast packets.

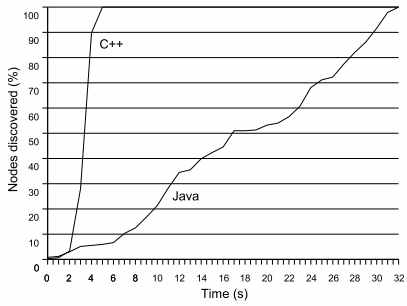

Figure 2. Performance comparison

Connections to hosts in other domains: Since MundoCore guarantees that a process is listening on the primary port if at least one process runs on a machine, it is possible to connect to remote machines in other domains by knowing the hostname or IP address. Neighbor messages then propagate information about this new link and can connect processes across multiple domains.

Buffering: Correct semantics of the Publish/Subscribe system at a higher layer is guaranteed even in presence of network delays or delays caused by high system load. This is implemented by means of buffering subscriptions and messages for a certain period of time, which is given by the maximum expected network delay plus some tolerance.

5.2 The Test Program

Each instance of the test program receives its number (rank) and the total number of processes as parameters. The program then advertises a communication endpoint using a name generated from its rank (“client1”...“client32”) and listens for incoming messages. Next, the program attempts to discover the endpoints of all other clients. Finally, it sends a message to each endpoint. On reception of a message, this event is written to a log file together with a timestamp. This shows how many communication endpoints could successfully be discovered and used. Furthermore, the timestamps offer a performance comparison.

The test program uses channel-based Publish/Subscribe. Each instance subscribes to the channel with the same name as its own and publishes a message to each other channel.

5.3 Performance Comparison

The tests were conducted on 4 machines running SuSE Linux 9.1 connected via GBit-Ethernet. On each machine, 8 processes were started. This gives a total of 32 processes and 992 discovery operations and message passes. Figure 2 shows a performance comparison of the Java and C++ versions of MundoCore. The graph shows time on the x-axis and the number of successful message passes on the y-axis.

Because the peer-to-peer framework JXTA also features automatic discovery and is available on the Web, we chose to also run comparison tests with JXTA. Our first results indicate, that the performance of MundoCore is considerably better and JXTA is only able to discover about 80% of the communication channels, i.e. pipes.

A major drawback of JXTA for Pervasive Computing applications is, that it lacks zero-configuration capabilities. It is basically designed for running only a single process per machine and brings up a GUI on the first start of each process where the user has to manually assign port numbers, a client name and create an administrator account.

6 Programming Abstractions

The two most relevant programming abstractions supported by MundoCore are Publish/Subscribe and distributed object-oriented programming.

In the Publish/Subscribe paradigm, Subscribers express their interest in an event and are notified of any event, generated by a publisher, which matches their registered interest. An event is asynchronously propagated to all subscribers that registered interest in that event. The strength of Publish/Subscribe is in the full decoupling in time, space and synchronization between publishers and subscribers [2].

Content-based Publish/Subscribe is especially useful for processing and distributing context information. It decouples the sensors which provide information from higher-level software which makes decisions about what to do with the information. Sensors generate events which they then publish on a channel. There can be one or more sensors publishing on a single channel. The best solution for each particular case depends on the details, such as how many sensors are involved, where they are placed, etc. Content-based filters allow services to define subscriptions that span multiple sensors as sources while filtering for a certain aspect, e.g., "Report all sightings of badge X in any room".

At the heart of distributed object computing middleware are Object Request Brokers (ORBs), which eliminate many tedious, error-prone, and non-portable aspects of developing and maintaining distributed applications using low-level network programming mechanisms. In particular, ORBs automate common network programming tasks such as object location, object activation, parameter marshaling/demarshaling, socket and request demultiplexing, fault recovery, and security. Thus, ORBs facilitate the development of flexible distributed applications and reusable services in heterogeneous distributed environments. [6]

6.1 Experiences with Users

We have observed that programmers starting to use MundoCore first try to solve problems with the programming abstractions they are used to from using other frameworks. In most cases, they make use of RPC-based designs, where the Publish/Subscribe paradigm would be better. In the following, we describe two different implementations of a room-control application. The first design was made by a MundoCore "newbie" based on RPC. The second design shows a better solution based on Publish/Subscribe.

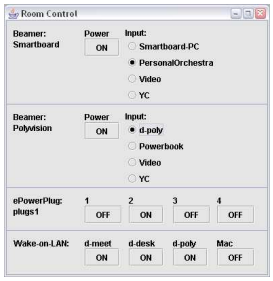

Figure 3. Room control applet

6.2 Room Control

Our meeting room is equipped with several wall displays and projectors. A number of computers are used to drive the displays and host application services. Our demo applications typically use an overlapping subset of devices in the room. Our aim is that we want to be able to start a certain demo with a single click. All involved devices should be automatically configured. To control devices in the room, services to control smart power plugs, projectors, and computers have been developed.

6.3 RPC-based Control Panel

The control panel application discovers the available device control services. For each discovered device, a visual component is displayed (Figure 3). The application allows to change device settings, like turning devices on and off. The first version of this program was a straightforward implementation based on RPC.

At program startup, the control panel application discovers available services. It then calls methods of the respective service to get the current state and updates the visual controls accordingly. When the user changes a setting, a method is called on the service. Because multiple control panel applications might be running, each program periodically polls the service to get the current state.

This implementation has two major drawbacks. First, the method calls to change settings on devices are blocking and report if the operation was successful in the return value. Some method calls can take between 3 and 20 seconds. For example, to turn a plug on a smart power plug on or off takes about 3 seconds. This is because the power plug is controlled via an embedded web server and the MundoCore

service has to build HTTP requests and parse HTML pages during interaction with the switch. These blocking method calls also lead to a blocking behavior of the user interface.

Second, this implementation uses polling to reflect changes made by other clients. A better way to make these updates would be to use Remote Listeners. However, in this case the service implementations would have to do the necessary Listener bookkeeping by themselves. The next section shows how this program can be implemented in a better way using the event-based paradigm.

6.4 Event-based Control Panel

Like the RPC-based application, the available device control services are first discovered. Then, a channel name is obtained from the service with an RPC call. Each service has a separate channel that it shares with all its clients. This channel serves as a back-channel that is used to deliver events from the service to its clients.

Clients send requests to a service by means of one-way RPC calls. Once a service receives such a request, it immediately emits a *confirm* message, indicating that it has started to perform the requested action. To emit this message, the service makes an one-way call to a client-side Stub which then generates the message and publishes it to the common back-channel. This reply message may also contain the estimated time it will take to complete the action. When the action is finished, the service emits a *done* message, indicating that the action has been performed.

The control panel clients listen for messages on the back-channel and use a server-side Stub for demultiplexing and dispatching. If a *confirm* message is received, the client produces a visual feedback showing the user that an action was started. For example, power buttons change their colors to dark green indicating that they are about to be turned on. If the client receives a *done* message, it updates the display to reflect the new state. For example, power buttons change their colors to light green.

By using Publish/Subscribe together with Stubs, we do not lose type-safety when switching to an event-based programming paradigm. The client-side and server-side Stubs are automatically generated by a proprietary precompiler.

7 Conclusion and Outlook

In this paper we described the distributed test architecture we are using to verify the correct behavior of our MundoCore middleware and applications built on top of it. Simple tests can be directly programmed in an XML-based scripting language. More complex tests are natively programmed in Java or C++ and the script interpreter is used to distribute, start and monitor the test processes.

However, there are still many open questions in the area of testing, monitoring, and debugging of distributed applications. Debugging of applications that just make use of synchronized RPC is not difficult, because the processes are tightly coupled and their program flows are synchronized. When communication processes are decoupled in time and space, as in pervasive computing, debugging requires additional tools. This issue has already been investigated in parallel computing, but debugging tools for parallel systems are tailored for communication libraries like MPI [3] and cannot be directly applied to Pervasive Computing. The most annoying question in debugging event-based applications is: “Why did I NOT receive event X?”.

We have also described a typical discovery problem and argued that middleware should provide guarantees to applications regarding the quality of service discovery. We showed that MundoCore provides reliable discovery and compared the performance of the Java and C++ implementations. We are planning to apply these tests also to other communication frameworks in our future work.

We further discussed our experiences on what programming abstractions are used by programmers in MundoCore. We found that RPC is often used in places, where the event-based paradigm would be better. To date, event-based programming is widely used in GUI toolkits and media processing libraries. We think that tutorials, code samples, and design patterns have to be provided with MundoCore to make programmers aware of the advantages of event-based programming in communications.

References

[1] C. Becker, G. Schiele, H. Gubbels, and K. Rothermel. BASE: A Micro-Broker-Based Middleware for Pervasive Computing. In *Proc. of IEEE PerCom'03*, pages 443–451, 2003.

[2] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[3] M. Forum. The Message Passing Interface Standard. http://www-unix.mcs.anl.gov/mpi/.

[4] A. Hartl, et al. Engineering Multimedia-Aware Personalized Ubiquitous Services. In *IEEE Symposium on Multimedia Software Engineering*, pages 344–351, 2002.

[5] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. Mobile Computing with the Rover Toolkit. *IEEE Transactions on Computers*, 46(3):337–352, 1997.

[6] F. Kon, et al. Monitoring, Security, and Dynamic Configuration with the dynamic TAO Reflective ORB. In *Middleware 2000*, 2000.

[7] Sun Microsystems. JXTA. http://java.sun.com/othertech/jxta/index.jsp.

[8] M. Roman, F. Kon, and R. Campbell. Reflective Middleware: From Your Desk to Your Hand. IEEE Distributed Systems Online Journal, 2001.