MundoCore: A Light-weight Infrastructure for Pervasive Computing

Erwin Aitenbichler, Jussi Kangasharju, Max Mühlhäuser

Abstract

MundoCore is a communication middleware specifically designed for the requirements of pervasive computing. To address the high degree of heterogeneity of platforms and networking technologies, it is based on a microkernel design, supports dynamic reconfiguration, and provides a common set of APIs for different programming languages (Java, C++, Python) on a wide range of different devices. The architectural model addresses the need for proper language bindings, different communication abstractions, peer-to-peer overlays, different transport protocols, different invocation protocols, and automatic peer discovery.

1 Introduction

The original aim of distributed object computing middleware was to enable the cooperation of objects independent of devices, operating systems, and programming languages. Over time, personal computer and server platforms became more powerful and had no problems running large, monolithic, and not well-optimized middleware software. Pervasive computing introduces a wide spectrum of new computing platforms, vastly different in terms of size, mobility and usability. The lower end of this spectrum is marked by computers embedded into everyday objects and small sensor nodes that often have only very limited processing capabilities.

Pervasive computing does not build on single computing devices - its real power emerges from the cooperation of many devices. Hence, communication is a fundamental requirement. Because the communication requirements of devices are also very different, several different wireless networking technologies are in place. To address the increased heterogeneity of devices and networks, modular and reconfigurable middleware architectures gained interest.

1.1 Goals of MundoCore

MundoCore is the lowest middleware layer of our smart spaces platform. It is responsible for all communication-related aspects and was specifically designed for the needs of the services in the higher layers. The typical communication requirements in smart spaces are described in the following.

Pervasive computing applications running on mobile devices have to be adaptive on multiple levels, because it is unlikely that the surrounding infrastructure will be equally "smart" everywhere. Today we have a growing number of different wireless networking technologies, tailored for specific purposes. The combination of low-bandwidth networks with a large coverage area and hotspots with high-bandwidth connectivity leads to so-called wireless multitier networks [1]. In order to provide mobile users with a continuous network connection, it is necessary to provide seamless handovers between different networking technologies. IP supports, e.g., only simple roaming between WLAN cells, because this can be handled at the driver level. However, switching to a different network interface breaks open TCP connections.

Published in: E. Aitenbichler, et al., MundoCore: A light-weight infrastructure for pervasive computing, Pervasive and Mobile Computing (2007), doi:10.1016/j.pmcj.2007.04.002, published by Elsevier B.V.

An adaptive middleware must be adaptive at all levels. Roaming to a different network can also require different routing strategies. Single-hop routing is a good strategy for highly dynamic networks with not more than 30 nodes. For larger networks, the introduction of *super-peers* [29, p. 355] is useful and communication on the global scale may require concepts like Distributed Hash Tables [23, 26, 30]. Because of uneven conditioning, the services provided by the surrounding environment also change. When a user enters a smart space, the user's personal computing environment should be able to spontaneously interact with the computing devices embedded into the environment. When the user leaves the smart space, it may be necessary to switch from using remote services to local services, that might be less powerful.

MundoCore addresses the above adaptivity issues by providing a layer model with modular services, that can be replaced on demand and applications can extend the middleware core by providing additional transport, discovery, routing, and brokering services.

An effective API must provide functions to programmers suitable for their specific application scenarios. Distributed Object Computing is a widely-accepted and easy to use programming model. In context-aware and other information-driven systems, publish/subscribe is a good abstraction for distributing events, because it supports multicasting and decouples data producers from data consumers. Having the right abstractions offered by the middleware leads to reduced application development time and reduced size of application code.

TCP/IP with its notion of streams was designed for transferring files and documents over the Internet. However, it is a bad match for many message-based applications, like remote invocations, sensor events, or multimedia data. Applications should be able to tailor the communication stack specifically for their needs. For example, sensor data in augmented reality applications needs to be distributed with low latency, while losing single samples is not critical. This leads to a different choice of transport protocol (i.e., UDP instead of TCP). If low latency and reliability are required, then the application might want to add a forward error correction handler to the protocol stack.

In strictly layered communication architectures, a protocol layer is only allowed to interact with the next lower and the next higher layer. However, in many cases, this approach is not flexible enough and we already observe a number of "layering violations" in current systems (see section 4). Also, if a layer is adaptive, its adaptive behavior should not be completely hidden inside the layer. Instead, the information that an adaptation took place should be propagated to all interested external components. Finally, applications should be able to influence the amount of control they want to have over adaptations. This requires means to communicate the changes in context observed in lower protocol levels up to the application level.

1.2 Our Contribution

In the following, we describe a modular middleware system that addresses the issues described above. Our contribution is threefold.

First, the modular architecture supports a wide range of devices reaching from small sensors up to servers. The connector abstraction allows to fully decouple components and allows us to separate the service functionality itself from concerns like service interconnection, service orchestration, or error recovery into separate services. To implement this decoupling, we made an extension to the Java language to provide output interfaces.

Second, the layer model organizes communication services into groups, according to the addressing schemes and message structure used. With this model, we aim to harmonize the worlds of DOOP, Publish/Subscribe, Peer-to-Peer and multimedia communication. This gives a number of interesting synergies, e.g., programming language objects can be externalized and filtered by the content-based publish/subscribe system or oneway calls can be used to multicast using the publish/subscribe system. Also, in the lower levels, different systems share much common functionality. For example, an object request broker can rely on a publish/subscribe system for object location and naming.

Third, with the concept of protocol heaps, applications can tailor the protocol stack according to their needs. Adaptivity and custom protocol stacks are often conflicting goals. We describe a method to support both.

This paper is organized as follows. Section 2 describes the middleware architecture, section 3 the layer model, and section 4 the protocol heap model. Section 5 explains how to program with MundoCore with a few code examples. Section 6 presents evaluation results and section 7 an inspection tool for the middleware. In section 8 we discuss related work and conclude the paper in section 9.

2 Middleware Architecture

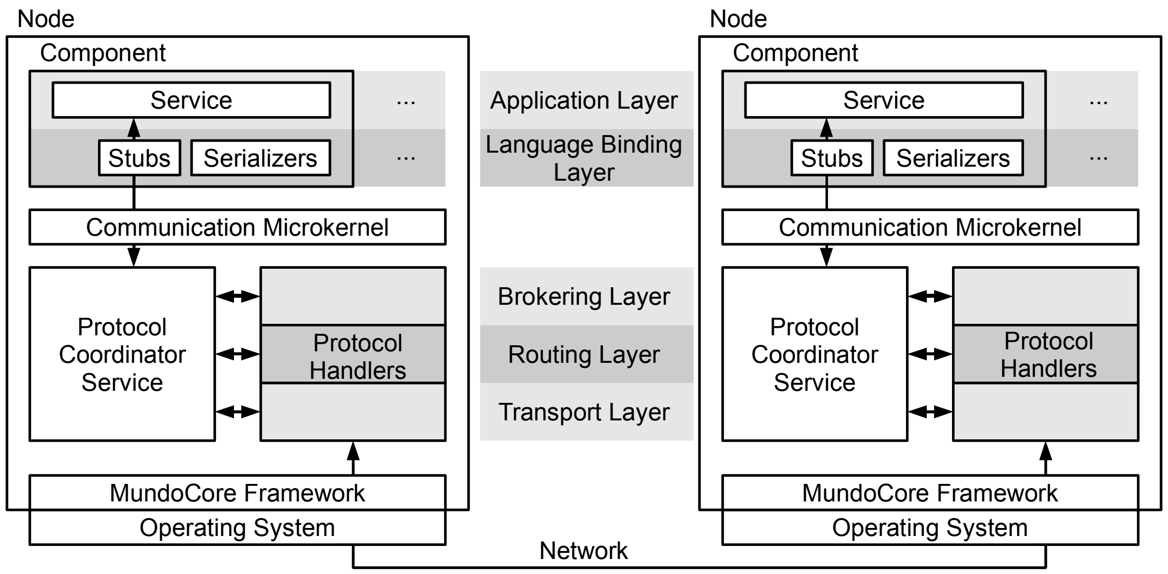

Figure 1: Overview of MundoCore Architecture

The architecture of MundoCore is shown in Figure 1. At the most basic level, the *MundoCore framework* provides utility classes and platform abstractions for memory management, process management (threads and synchronization), and I/O APIs. The microkernel and all services build on top of this framework. The central element is the *communication microkernel*, which handles all communication between services within the same *node*. Communication with remote nodes involves the *protocol coordinator* and *protocol handler* services, which are described in the next section.

2.1 Communication Microkernel

First, we describe the intra-node communication. The kernel permits services to communicate with each other based on a channel-based publish/subscribe interaction scheme. Services interested in certain messages subscribe to the corresponding channel and receive notifications when new messages are published to that channel by other services. Services that want to emit messages to a channel have to advertise first. The basic communication operations are:

$$\begin{array}{rl} S = & \textbf{subscribe}(\mathit{channel}) \\ & \textbf{unsubscribe}(S) \\ \text{callback:} & \textbf{received}(\mathit{message}) \\ P = & \textbf{advertise}(\mathit{channel}) \\ & \textbf{unadvertise}(P) \\ & \textbf{send}(P, \mathit{message}) \end{array}$$

The most important building blocks of the MundoCore middleware and of applications are *services*. Services are "coarse-grained" application or middleware components with well-defined interfaces that typically consist of multiple objects. Applications in a smart environment typically span multiple devices and involve dozens of services distributed among these devices. Services are interconnected by *channels*, which end in *input* or *output ports* of services.

2.2 Services in MundoCore

A service implements a certain application or middleware functionality. At the basic level, there is no distinction between middleware and application services. The middleware itself is modular and can be extended in many ways. Services have the following properties.

A service is a container for objects that are closely related to each other. It is the distribution, migration, and persistence unit.

- **Distribution:** References to local and remote objects are explicitly distinguished by different types of reference variables. This distinction explicitly expresses that the interaction with remote objects is inherently different. Local object references express tight coupling between objects. Remote object references implement loose coupling. Remote method calls have different error semantics and are by orders of magnitude slower.
- **Migration:** Services are not bound to a particular physical location, they can be migrated from one address space to another. The communication abstractions provided by MundoCore enable services to maintain all communication links to other services even if they are moved to other physical locations.
- **Persistence:** Services implement methods to write their configuration or state to persistent storage and to instantiate the service again from the data in the persistent store.

Services also serve as a synchronization concept to deal with the concurrency that is natural to distributed systems. Services implement the Active Object Pattern [12] to decouple operation invocation from operation execution.

2.3 Connector Abstraction

Object-oriented programming languages provide the abstraction of objects and systems are composed out of many interacting objects. While the interfaces of objects are clearly defined on the "input side", mainstream object-oriented languages do not provide explicit support to express the connections of an objects to other objects. Instead, connections are implicit, either as object references on the heap or implemented by applying design patterns such as Proxy, Adapter or Observer [12]. As a consequence, connection code is not clearly separated and becomes part of the application code. This has several drawbacks:

- The Observer pattern is often used to connect an event source with multiple event listeners, e.g. in GUI toolkits. This pattern requires the event source to maintain a list of listeners and to provide methods for registration and deregistration. Hence, the necessary listenerlist-bookkeeping code is replicated over and over again.
- The dispatch code in event sources is replicated as well. This issue cannot be simply solved by subclassing, because if general parameter types are used, the system loses type safety, or additional methods have to be implemented for each event type.
- In distributed systems, the dispatch code also has to take exception handling and synchronization aspects into account, which adds a considerable additional overhead to each dispatch method.

Because of this code replication and the missing separation of connector and application code, it is difficult to reuse and evolve connection mechanisms. In MundoCore, services have well-defined *input* and *output* interfaces. A class can define its input interface with the Java-standard *interface* keyword and its output interface with the *emits* keyword, which we have added. Services can emit events or perform anonymous request/reply calls by using the special variable `emit`. The outputs and inputs of objects can be connected with the `connect` operation, which allows to connect objects directly or via channels:

connect(*eventSource*, *eventListener*)
connect(*eventSource*, *channel*) and **connect**(*channel*, *eventListener*)

Subscription management and event dispatching is implemented by the publish/subscribe system. Method invocations are marshaled to message structures by stubs, which are automatically generated by a precompiler (see section 2.6). The examples in section 5 show how our connector abstraction is used in programs.

2.4 Component Model

Especially in pervasive computing, a common operating system, the same middleware implementation, or programming language on all devices in the system cannot be assumed. As a consequence, we do not try to keep the principle of data abstraction across component boundaries. The serialized forms of requests, replies, and events are explicitly described along with the interfaces. Thus, MundoCore's programming model is programming language independent and interoperability is established on the protocol level. It has also been observed in the one.world project [14] that data-centric data models have significant advantages over programmatic data models (see also section 8).

Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system [33]. Because we want to achieve independence between components, independence from platforms and programming languages, and support for late composition, many programming abstractions known from object-oriented languages cannot be used, e.g., type declarations, preprocessor macros, templates, or (Smalltalk) code blocks. Other abstractions, such as methods, classes, modules, or entire applications, can form components, as long as they have the necessary interfaces to be composable.

MundoCore services can be packaged to form such self-contained software components. The component model is concerned with service interdependencies, dynamic loading of services, and the description of services. One or multiple services can be packaged into a MundoCore Component.

A MundoCore Component consists of

- the implementations of services in binary form,
- the implementations of stubs, serializers, and data objects that are exchanged via the service interfaces,
- interface and data object descriptions (in WSDL), and
- the description of dependencies with base libraries.

Components are packed into Java Archives (JARs) and can be loaded and unloaded from a MundoCore node at run time. Data objects have an arbitrary number of public fields and constant value declarations, but no methods. They are typically used to implement event notification data structures in a portable way. Since MundoCore uses the WSDL standard to describe interfaces, services can be easily made accessible as Web Services via simple proxies.

2.5 Static and Dynamic Configurations

It is possible to create static or dynamic configurations for deployment. In a static configuration, certain middleware and application *services* are packaged together with the middleware core into a single distribution package (JAR). In this case, the services are selected at design time and the configuration process is similar to configuring a Linux kernel. When the underlying Java platform supports custom class loaders and has sufficient resources (RAM), then MundoCore also supports dynamic configurations. In dynamic configurations, optional services are packaged as *components* that can be loaded by a process on demand.

The loading, unloading, and administration of dynamic components is controlled by a service manager service running on each node. The service manager monitors a deployment directory in the filesystem. If a new Java archive file is copied into this directory, it gets automatically loaded. An XML configuration file in the archive describes which service instances should be created and how they should be configured. Components can also be unloaded by simply deleting the corresponding archive file in the deployment directory. The service manager then automatically shuts down all service instances created by this component. In addition, the service manager provides interfaces for the remote instantiation, configuration, and administration of services.

Hybrid configurations that combine the benefits of both approaches are also possible. Services that are permanently used by an application can be linked statically, while services that depend on certain contexts are built as independently loadable components. Such contexts are, e.g., available memory, available network connectivity, or available location tracking systems in the environment.

2.6 MundoCore Precompiler

The MundoCore precompiler generates the necessary glue code between services. It reads interface descriptions from Java 1.5 conforming metadata tags embedded in Java source files, from special comment tags embedded in C++ header files, or from WSDL files. Based on these interface descriptions, it can generate interface declarations for Java, class declarations for C++, WSDL files, as well as object serializers, client and server stubs for remote method calls for Java and C++. In addition, it implements a C-style preprocessor for Java source files and some Java language extensions. The preprocessor is very useful for writing portable Java programs, since the Connected Limited Device Configuration (CLDC) profile that is typically used on cellphones, is significantly different from other Java profiles. Without the preprocessor, a separate source tree for the CLDC version would be required.

MundoCore uses some proprietary extensions to WSDL to implement *data objects* (see section 2.4). The language-specific implementations of data objects and their serializers are automatically generated by the precompiler.

3 Conceptual Layers of the Communication Architecture

A *node* hosts an arbitrary number of services implementing the application-specific functionality, together with the basic services. A node typically corresponds to a single operating system process.

When the kernel receives a subscribe or advertise request, it in turn generates corresponding notifications. Thus, services may subscribe to subscription notifications in a way similar to subscribing for normal messages. Message brokers use this concept to get information about new *communication endpoints*, i.e., publisher and subscriber objects. The concept of subscribing for subscriptions can also be used for on-demand automatic instantiation of services and to efficiently implement multi-stage information flows.

This section presents the communication architecture of MundoCore, which comprises all services involved in the transport of a message from one application-layer communication endpoint to the other. After the description of the general architecture with its components and the interfaces between components, details about the implementation in MundoCore are presented.

This architectural layer model classifies communication services according to the *addressing schemes* used and *message semantics* understood. Transport services in the lower layers require routes as address information, while message brokers in the higher layers may support indirect addressing via channels. In terms of semantics, low-level services treat messages as binary data blocks, while application-level services use programming-language native objects (Table 1).

Layer	**Addressing**	**Message Content**
Language Binding	Any	Programming language native objects
Brokering	Any	Passive objects
Routing	Node-Id	Passive objects
Transport	Route-Id	Binary data

Table 1: Addressing and Message Content in Different Layers

The layer model only serves as a reference model and does not define a strictly layered system. How communication stacks can be configured is described in section 4. The services in the different layers have the following responsibilities, starting with the lowest layer: Transport services provide connections to adjacent nodes. The overlay routing layer routes a message to the node with the specified node-id. The abstraction provided by this layer makes routing independent of underlying transports. Messages that are indirectly addressed to their targets are handled by the broker services in this layer. The language binding layer offers convenient programming interfaces to access services in the lower layers.

3.1 Transport Layer

Services in the transport layer are responsible for transporting messages to adjacent nodes. Frequently used transport services in our projects include IP, (bluetooth virtual) serial connections, and infrared links. The information how to reach a specific destination peer is shared between routing services and transport services by means of Route objects. Route is an abstract class that contains the node-id of the destination node. Each transport service provides specialized versions of Route that contain additional service-specific information. For example, an IPRoute adds the IP address of the destination. Concrete routes are discovered by a transport service itself or are provided by higher level services.

The interface of transport services provides methods to *manage connections* and to *transport messages.*

- **Connection management:** The methods provided allow to open the necessary *connections* required to reach the destination specified by a route.
- **Message transport:** These methods allow to send a message to a remote peer using the specified route.

Transport services emit the following events:

- **Discovery:** The service signals when a new neighbor node is discovered. It creates a new route object containing the node-id of the remote node and service-specific address information. Once a routing service receives this event, it adds the route to its routing table. The transport service also generates an event when it loses connection to a remote peer.
- **Message delivery:** The service signals when it receives a message from a remote peer.

Beside message transport, the second important functionality of a transport service is the automatic discovery of peers. Node discovery could be handled by separate services as well. However, in many cases discovery is closely related to message transport and often based on the same set of APIs. For example, the IPTransportService uses TCP or UDP packets for message transport, and its discovery mechanism uses TCP, UDP unicast, and UDP broadcast packets.

In the following, the problem of *node discovery* is considered. Because users should be enabled to interact with smart environments in a natural way, the computing devices involved must be aware of each other. Thus, devices must be able to find all available devices and services in the nearby environment in a reliable way. It should be noted that node discovery is not necessarily the same as *service discovery.* Service discovery is a concern of a higher layer and involves the description of services, caching of service information, and complex queries. However, if node discovery at the transport layer allows to discover all nearby nodes in a reliable manner, a higher-level service can easily discover all services on those nodes. Services that provide node discovery should be designed according to the following principles:

Autoconfiguration: Transport services should not require any node-specific or dynamic configuration information. Instead, a single, common, and static configuration should describe how nodes can find each other. Most current systems lack zero-configuration capabilities.

Reliability: Node 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. In the unlikely case that packet loss in the network is extremely high, a node joining the network will not be able to discover any other node. Conversely, if a node successfully joins the network, all nodes in the network see the entire network.

Discovery in MundoCore is based on the following three concepts:

- A node joining the network announces its presence with IP broadcasts.
- Because the loopback network drivers of some operating systems virtually drop all broadcast packets, the discovery of local processes cannot rely on broadcasts. The rendezvous

via the *primary port* ensures reliable discovery in this case. The primary port is a well known TCP port defined in the configuration file. MundoCore ensures that if at least one process is running on a machine, a process will listen on the primary port.

- *Neighbor messages* are used to propagate information about new nodes in the network. This permits to open 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.

Thus, MundoCore features true zero-configuration in most cases. There is no need to provide any node-specific, dynamic configuration information. For example, any number of processes can be started on a single host without having to configure port numbers manually.

3.2 Routing Layer

Routing services deliver packets to remote nodes that are directly addressed by their node-ids. A routing service uses one or more transport services in the next lower layer for the actual message transport. For a given message and address, it picks a suitable transport service and route according to a metric. Route objects can be created by services at lower layers:

- Many transport services support the automatic discovery of neighbor nodes. Each time a new neighbor is found, discovery creates a new route object and notifies the routing services. For example, the IP transport service uses UDP broadcast packets to discover other nodes in the same subnet.

Route objects can also be created by services at higher layers:

- Routes can be provided by a service registry. Only when a client wishes to use a specific service, the underlying communication links are opened. This technique is useful for power-saving.
- In many cases, routes between different domains are statically configured by the administrator. These routes interconnect super-peers. Usually, also firewalling rules are defined on such gateway nodes.
- Bluetooth requires the explicit formation of links, for power-saving reasons. Routing services can request network access services to open new routes.
- The sender of a message can specify the full route for source-routing.

MundoCore supports both structured and unstructured overlays. The type of overlay usually has to be decided from case to case based on the requirements of an application. A large number of nodes joining and leaving a structured overlay - often referred to as *churn* - concurrently puts particular stress on the overall stability of the system, reducing routing efficiency, incurring additional management traffic, or even resulting in partitioned or defective systems [29, p. 93]. While structured overlays offer better scalability, unstructured overlays are more reliable, because they are less affected by churn. However, the routing table sizes grow proportional to the size of the network.

Currently, MundoCore implements routing services tailored for the following three different network structures:

- **Single-hop:** In a single-hop network, each node communicates with each other node directly. This network structure is usually the best choice if the network is highly dynamic, like in spontaneous collaborations, and the number of nodes does not exceed about 30 clients.

- **Hierarchical:** A larger number of clients can be supported by introducing *super-nodes*. Super-nodes should be highly available and rather static in terms of location. For example, a smart environment has embedded computing resources and can easily provide such static super-nodes. The routing algorithm between super-nodes can be a variant of distance vector routing. Because a generic network topology is allowed, the routing algorithm must avoid routing messages along cycles. In order to do so, the classic reverse path forwarding technique is used. In most cases, this algorithm is not used to route messages through the network. It is rather used to route subscribe and advertise requests between channel- or content-based publish/subscribe brokers. The publish/subscribe layer then builds a multicast tree for the distribution of messages.

- **Structured:** Especially on the publish/subscribe layer, unstructured networks only scale up to a certain degree, because the size of routing tables grows with the size of the network. In structured overlays, each data item, etc. is assigned a location where it is stored in the overlay. Thus, structured overlays are more suitable for large-scale deployments. Mobile clients use single-hop communication to the nearest super-peer. The super-peers in the same site use distance vector routing. A site gateway connects the network to a global, structured network.

3.3 Brokering Layer

The routing layer described in the last section provides the functionality to route messages to a node by specifying its address. The publish/subscribe layer adds the functionality for indirect addressing and multicast. The implementation of MundoCore provides services for channel-based and content-based addressing. Because addresses are an abstract concept, the architecture allows for plugging in additional services that handle these new address types.

Since the basic programming paradigm is already channel-based publish/subscribe, message brokers on this level forward messages to remote nodes. A channel-based broker simply matches publishers and subscribers by comparing the channel names provided with their advertisements and subscriptions. As a consequence, such brokers are relatively simple and can also run efficiently on resource-constrained devices. The channel-based addressing scheme is sufficient for many applications.

If more flexibility is required, then the content-based addressing scheme can be used. In this scheme, publishers do not address consumers directly in any way. Instead, they provide messages that are fully transparent to the routing service and subscribers may incorporate any part of a message into their filter expressions. Compared to channel-based routing, this approach offers more flexibility and extensibility, however filtering is more complex and has higher requirements in terms of computing resources. Filters are specified with subscribe and advertise operations. A subscription filter specifies the interests of the client. It will only receive messages that match this filter. An advertisement filter expresses which messages a client intends to emit. A client may only emit messages that match the filter specified in the advertisement. The filter model in MundoCore builds on the notion of conjunctive filters. This is a powerful model, however, its expressiveness is intentionally limited in some aspects to allow for filter merging in a distributed network of brokers. This is an important prerequisite to build scalable publish/subscribe systems. A detailed description of the data and filter model can be found in [2].

Brokering services can further be distinguished by the type of overlay network they assume:

Routing in unstructured overlays: In a single-hop network, each node communicates with each other node directly. To match the subscriptions of a node with all advertisements, it has to contact all its neighbors. This causes a lot of traffic when a new node joins the network. However, departing nodes do not affect the communication of other nodes in the network. This mode of operation is usually the best choice if the network is highly dynamic, like in spontaneous collaborations. The efficiency and scalability of larger networks can be enhanced by introducing super-nodes. A super-node acts as a central rendezvous point for subscriptions and advertisements for a group of peers.

Routing in structured overlays: Structured overlays offer a better scalability, because the subscription tables are distributed among nodes in the network. This approach scales well but requires a rather static network. While routing is very efficient, the maintenance cost for the overlay is high. If a node departs, it may affect the communication of seemingly unrelated remote nodes. To circumvent loss of global knowledge, information must be replicated in the overlay. The MundoCore services for structured overlays build on top of FreePastry, a publicly available implementation of Pastry [26].

3.4 Language Binding Layer

The interfaces between lower layers are characterized by the message data structure and a simple message handler interface. The language binding layer now aims to provide a broader and more easy-to-use programming interface for the application layer.

With *externalization*, MundoCore supports the transformation of objects native to the programming language into structured messages. The methods that perform the actual data conversions are automatically generated by the MundoCore Precompiler. In contrast to most other frameworks, in MundoCore this mapping is a two-step process. Externalization converts an *active object structure* with arbitrary user objects to a *passive structure*, that only contains a number of base types. Serialization then converts the passive object structure to e.g., an XML document or into a binary form.

Remote Method Calls allow to transparently call methods on remote services via stubs.

4 Protocol Heaps

Protocol layering has served well as an organizing principle for the strict end-to-end model of the original Internet infrastructure. However, today firewalls, NAT boxes, proxies, caches, QoS, multicast, overlay routing, and tunneling require many "layering violations" [8, 22]. This suggests that strict layering is not a good abstraction, since some interactions between layers have to happen implicitly and hidden - as explained below.

The network stacks in modern operating systems are designed according to the ISO/OSI 7-Layer Model. Since most applications use TCP/IP for communication, the lower four layers implement a functionality that is commonly used. In contrast to that, common implementations for layers 5 (session) and 6 (presentation) never made it into operating systems, because the requirements of applications are too diverse. As a consequence, the original 5-Layer TCP Model [35] is still often used by the networking community, or the OSI-Layers 5 to 7 are all considered to be in the application layer [20].

Even in the lower layers, there is constant pressure for "layer violations" and to insert new functionality between existing layers. For example, MultiProtocol Label Switching was inserted at "layer 2.5", IPsec at "layer 3.5", and Transport-Layer Security at "layer 4.5" [8].

The same applies to the Bluetooth stack. Because wireless networks are based on a shared medium, multicast would often be available "for free". However, the Bluetooth stack only provides point-to-point links in the lower levels. Furthermore, emulations of higher-level service discovery performance in Bluetooth scatternets show that cross-layer optimizations can lead to 1-2 orders of magnitude better performance [22].

The *Interceptor* design pattern is commonly used in modular implementations of CORBA. It allows applications to extend a framework transparently by registering 'out-of-band' services with the framework via predefined interfaces [27, p. 109]. For a designated set of events processed by the framework, it specifies and exposes an interceptor callback interface. Applications can derive concrete interceptors from this interface to process occurrences of these events in an application-specific manner.

The interceptor concept has several drawbacks. Interceptors have no or only little control of how messages flow through the middleware services. In addition, an interceptor callback cannot split up a single control path into multiple, e.g., to split up a message into multiple fragments. Different interception points specify different, incompatible interfaces. This makes it difficult to write general handlers that can be registered at different interception points.

iBus [5] is an IP-based communication middleware with a configurable protocol stack. The bottom of each stack ends in a definable TCP or UDP socket and connections are distinguished by the respective IP port numbers. The order in which protocol handlers are applied is strictly defined by the stack. Protocol handlers are limited to relatively simple operations, like transformation of the payload, fragmentation, or flow control. A handler cannot redirect packets, implement overlay routing, or dynamically select among handlers in the next lower layer.

4.1 Basic Concepts of Non-Layered Architectures

An idealized *Role-based Architecture* determines the processing order based on dynamic precedence information carried in packets as well as static precedence associated with the protocol handlers in the nodes [8]. Giving up protocol layering has a number of immediate consequences. Layering provides modularity, a structure for metadata, encapsulation, and an ordering principle for processing packets.

Modularity is an important concept to decompose complex systems into better manageable, smaller pieces. Each protocol layer adds to the services provided by the lower layers in such a manner that the highest layer provides a full set of services to manage communications and run distributed applications. The principle of information hiding ensures independence of layers by defining services provided by each layer to the next higher layer without defining how the services are to be performed. This permits changes in a layer without affecting other layers. Because modularity is an indispensable tool for system design, alternate non-layered architectures must address this issue adequately.

The **structure of metadata** in layered architectures logically forms a stack. When a layer receives a packet from the next higher layer, it adds its protocol header and passes the packet on to the next lower layer. Conversely, if a packet is received from the next lower layer, the protocol header is removed and processed, and the remaining part of the packet is passed on to the next higher layer. Without strict layering, it must be possible to access metadata in random order, not only in FIFO order. Thus, the "stack" is replaced by a container holding variable-sized blocks of metadata, which may be accessed, modified, inserted, or removed in any order.

Encapsulation is the addition of control information to the payload data by a communications protocol. Encapsulation takes place at each layer of the OSI reference model. Non-layered architectures require a different organizational principle for data and metadata in a packet. Encapsulation only needs to be performed if it is necessary. In most cases, metadata about the data being encapsulated does not need to be encapsulated itself, as would be the case in a layered architecture. The payload of a packet only needs to be encapsulated, if it has to be, e.g., fragmented or transformed.

The **processing order** is fixed in layered architectures. A packet either travels downwards or upwards through the stack, strictly from one layer to the next layer. In non-layered systems, the processing order is no longer defined implicitly. In principle, protocol handlers may operate on messages in any order, or even simultaneously. However, in most cases an appropriate partial ordering among specific handlers is required.

Because reasoning on dynamic properties is complex and ambiguous, MundoCore relies solely on static properties.

4.2 Messages

A *message* is a tuple $(type, chunk_1, ..., chunk_n)$ where *type* is the MIME type [17] and $chunk_1$ ... $chunk_n$ are the chunks of the message. Each chunk is a tuple $(name, repr, content)$, where *name* is the name of the chunk, *repr* is the representation of the content, and *content* is the content. The representation is one of the following:

- An *active object graph* contains objects native to the programming language.
- A *passive object tree* only contains basic data types, lists and maps. This data structure is comparable to a DOM tree [16]. However, it is typed and makes a clear distinction between lists and maps.
- A *binary data chunk* contains a variable-sized array of bytes.

Chunks are uniquely identified by their *name* combined with the *type*, i.e., the same chunk may exist in several different representations at the same time. Applications typically create messages from programming language native objects. This active object graph is first externalized into a passive object tree. Then, this passive structure is serialized into XML or a binary format which can be sent over a transport connection. Transformations between representations are only made when they are necessary. As long as a message is passed around within the local node, there is no need to perform any transformation on the message object.

Because the parts of a message structure can be accessed in random order, information from higher "layers" during sending, and information from lower "layers" during reception of messages remains accessible. For example, consider a transport service receiving a message over WLAN. This service is able to store the RSSI information associated with the received data packets in the message and pass it along all the way to the application this way.

4.3 Common Type System

MundoCore defines a set of basic data types that allow constructing structured and typed messages. This basic type system is common to all implementations of MundoCore (Java, C++, and Python). The basic scalar data types were chosen as a subset of the types defined in XML Schema. This enables a one-to-one mapping from messages containing only basic types to XML/SOAP documents and vice versa.

The container types *array* and *hash* allow for nesting and creating structured messages. These basic container types were chosen because this concept has proven successful, e.g., in the programming languages Perl and Python. Languages like Java or C++ only have scalar basic types. The language PHP only supports a hash type and arrays have to be implemented as hashes. This leads to significant performance issues. XML is tree-structured and does not distinguish between arrays and hashes.

4.4 Message Processing

MundoCore supports configurable protocol stacks that allow to tailor message processing specifically for applications' needs. For example, programmers can choose in favor of reliability, efficiency, or security. Custom protocol stacks can be created by combining protocol modules. Depending on an application's needs, it is possible to create stacks ranging from unreliable multicast transfer up to reliable, compressed and encrypted transfers. Such a protocol stack can be specified separately for each channel. The requirements for data transmission can be very different:

- If audio or media data streams are transmitted and played back immediately, it is important that data is transmitted with a low latency. In this case it is important to use a simple stack that does not unnecessarily delay the transmission. The loss of single data packets can be usually tolerated.
- If software or application data is transmitted, it is important that it is transmitted correctly. The time for transmission is of secondary importance. In this case it is important to select a stack that supports retransmission of lost or faulty data packets.
- If data packets exceed the maximum transferable unit (MTU) of the underlying transport, data packets must be fragmented beforehand, and then have to be reassembled on the receiver's side. For example, in media streaming applications, the size of a frame often exceeds the MTU of IP.
- If confidential data is transferred over an insecure network, it is necessary to encrypt this data. In many cases, data is also compressed before it is encrypted. On the receiver's side, the packet then has to be decrypted and decompressed accordingly.

4.5 Message Handlers

Protocol stacks can be defined at the granularity of single channels. A stack is described as a list of message handlers and channel-specific options. Message handlers implement the interface IMessageHandler with the operations up and down. The operation down is called by the framework when a message is passed to the handler from a higher layer and the operation up is called when a message is passed up from a lower layer. After the handler has finished processing, it emits the message using the IMessageHandler signal interface. An external *protocol coordinator* component is responsible for identifying the next handler and forwards the message to it. In addition, a handler may emit additional events, e.g., to indicate that it has discovered a new peer.

4.6 Downward Processing

A custom protocol stack is described as a list of *message handlers*. When a packet is transmitted, this stack is processed *downwards* starting from the application layer down to the transport layer.

The message handlers are applied on the message simply in the order they are specified. A stack is primarily made up of a number of routing and transport services, for example:

- A ChannelBroker implements a channel-based publish/subscribe message broker. There may be multiple implementations that build on different overlay network topologies.
- A RoutingService provides the functionality to forward a message to a remote node specified by its node address. It keeps track of available transport services on the next lower layer and picks a suitable transport for the message based on some metric. There may be multiple implementations that use different routing algorithms, e.g., single-hop, classic distance vector, or ad-hoc on-demand distance vector.
- An IPTransportService provides message transport over IP connections. There may be different implementations on this level that are based, e.g., on serial links or shared memory.

For improved flexibility and to support implicit adaptations, *proxy handlers* may be specified in the stack instead of specific handlers. A proxy handler picks the most suitable handler from a set of handlers. For example, specifying the proxy AnyTransportService permits MundoCore to use any available transport service. The concrete handler used is determined either by the proxy itself or the proxy tries one handler after the other until a handler accepts the message together with the destination address.

Secondary handlers may be inserted into the protocol stack based on an application's requirements. Such handlers include:

- The Activation handler transforms active object graphs into passive object trees and vice versa. This handler is in the stack if object serialization or remote method calls are used. In a typical stack, this is the first handler.
- The BinSerialization or XMLSerialization handlers transform a passive object tree into a binary or textual representation. In a typical stack, this is the second handler. The following handlers then operate on binary data.
- The Fragment handler allows fragmenting big messages into a number of smaller messages and to reassemble the fragments in the correct order to reproduce the original message. For example, this handler is useful if the underlying transport is UDP-based and the message size may exceed the maximum allowed size of UDP packets.
- The ZIP handler compresses the contents of the message.
- Encryption handlers allow encrypting and decrypting a message. Symmetric and asymmetric encryption methods are provided, e.g., AES and RSA. Usually, an application first generates a symmetric session key and then exchanges this key via a channel that uses asymmetric encryption. Then, the actual data transfer is performed over a second channel using symmetric encryption.

Applications can provide their own, additional message handlers, if necessary. Handlers may add metadata chunks with service-specific address and routing information or transform the payload data. If the content type of the payload is changed, the handler changes the MIME-type of the message accordingly. For example, a compression handler applied to a message of type text/xml changes the type to application/zip. The original content type is stored in a private metadata chunk so that it can be restored later on the receiver's side.

4.7 Upward Processing

When a packet is received, the protocol stack should be processed *upwards* starting from the transport layer up to the application layer. In contrast to downward processing, the transport service cannot simply forward an incoming message to the next higher handler, because each channel may have its own stack and the stack which matches the message is not known at this stage. Hence, upward processing is done in two phases:

1. In the first phase, the name of the destination channel is not known. The responsible message handler is determined by the MIME-type of the message. The mapping between MIME-types and handlers is unique, i.e., only a single handler can be registered for a specific type. If multiple handlers can process the same type, then a proxy handler must be used that picks and delegates to a concrete handler. The same handler can only be invoked once in a row in order to avoid infinite loops.

2. The second phase starts as soon as the name of the destination channel is known. As a consequence, the protocol stack associated with the channel is also known. The first broker service in the stack is now located and then the remainder of the stack is processed up to the application level.

4.8 Custom Protocol Stacks

A couple of examples for custom protocol stacks and their relation to the layer model are described in the following.

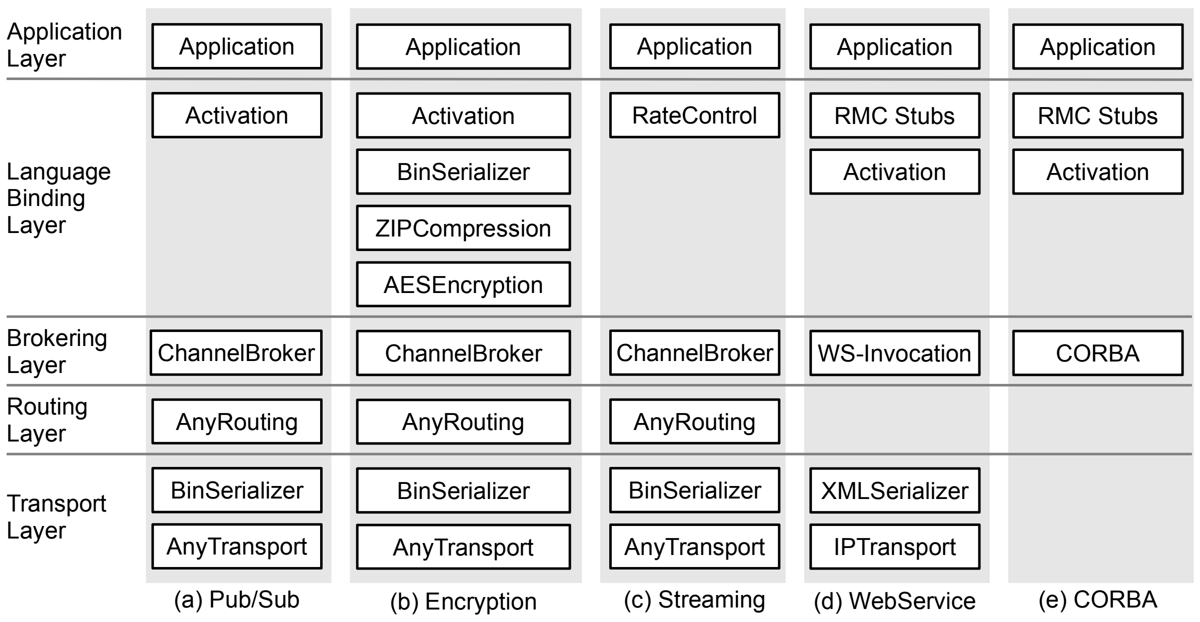

Figure 2: Examples for custom protocol stacks

The default stack provides publish/subscribe communication with channel-based addressing, object activation, and permits the middleware to automatically pick suitable routing and transport services (Figure 2a). Single channels can be encrypted by adding an encryption handler below activation. Because the encryption handler operates on binary data, the message must be serialized before. In addition, a compression handler was added to this stack as well (Figure 2b). For streaming media data, a rate control handler is typically used in a high layer to provide end-to-end Quality of Service (Figure 2c). Remote method calls can be performed by adding stubs in the language binding layer. Client and server stubs exchange invocation messages using the MundoCore RMC protocol. Invocation messages can also be translated to alternate inter-ORB protocols by adding the appropriate broker services, e.g. XML/SOAP Web Services (Figure 2d) or CORBA-IIOP (Figure 2e).

5 Programming with MundoCore

MundoCore has been implemented in Java, C++, and Python. Services are based on the MundoCore framework, which provides a platform abstraction and skeletons to build services. The C++ version supports the platforms Win32, Windows CE, Linux, uClinux, and MacOS/X. It provides wrappers for memory management, process management (threads and synchronization), and I/O (files, sockets, serial port, audio devices) APIs. The advantage of C++ is that it is easy to access hardware, low-level operating system APIs and program efficient audio and video processing services. Hence, MundoCore C++ is primarily used for low-level services and for services where performance is critical. Most higher-level services are programmed in Java, because of the higher productivity. The following discussion will be limited to the Java version.

In principle, the Java Virtual Machine and the Java class libraries are designed to hide the peculiarities of the underlying platform from applications. However, the three different Java profiles (J2SE, J2ME/CDC, and J2ME/CLDC) supported by MundoCore have significant differences, e.g., missing language features (no static access to class objects with .class on CLDC) or missing basic interfaces (no Cloneable on CLDC). Hence, our platform abstraction provides some of the missing interfaces. Missing language features have to be handled by using the preprocessor. A minimum configuration of the Java version is only 46 KB (JAR size) small and therefore easily fits on a mobile phone.

In the following subsections, we will explain the API of MundoCore by means of simple example programs. The basic API is similar to other publish/subscribe systems, such as JMS [31], or Elvin [28]. The novel elements in MundoCore are that messages can be accessed at different levels in different representations, the concept of filter objects, the seamless integration of remote method calls with the publish/subscribe system, and the connector abstraction enabling better decoupling for remote calls.

5.1 Publish/Subscribe in MundoCore

In the following, a simple chat program is considered. The publish/subscribe paradigm is well-suited for such group communication applications. Because the publish/subscribe system handles subscription management and distribution, it is only necessary to write a chat client application. A server is not needed. An arbitrary number of clients can be started in the network. The program publishes to and subscribes to a common channel, named chat.

```
class ChatClient extends Service implements IReceiver {
  private Subscriber subscriber;
  private Publisher publisher;

  // overrides Service.init
  public void init() {
    Channel ch=getSession().getChannel("lan", "chat");
    subscriber=getSession().subscribe(ch, this);
```

```
      publisher=getSession().publish(ch);
    }
    // implements IReceiver.received
    public void received(Message msg, MessageContext ctx) {
      System.out.println(msg.getMap().getString("text"));
    }
    public void send(String text) {
      TypedMap map=new TypedMap();
      map.putString("text", text);
      publisher.send(new Message(map));
    }
  }
```

To receive messages, the service *subscribes* to the common channel and gets a Subscriber object in return. The second parameter of subscribe specifies the object implementing the IReceiver interface. The received method of this interface is called when a message is received. Subscribers receive messages through *sessions*. Sessions decouple message delivery from message handling. Messages delivered by the kernel are stored in a queue and then dispatched to the handler by a separate thread. This concept enhances concurrency and simplifies synchronized access to objects that reside in their own threads of control. In order to send messages to the common channel, the service has to *advertise* first and gets a Publisher object in return. The send method of this object can then be used to publish messages.

5.1.1 Content-based Subscriptions

The following example shows how to define a content-based subscription with a TypedMapFilter structure representing a conjunctive filter. Only messages whose text field contains the substring IMPORTANT pass the filter.

```
TypedMapFilter filter=new TypedMapFilter();
filter.putString("text", AttributeFilter.CONTAINS, "IMPORTANT");
subscriber=ContentSubscription.subscribe(session, filter);
```

The same filter can also be specified using XQuery:

```
XQuery xq=new XQuery();
xq.parse("for $o in $msg where contains($o/text, 'IMPORTANT')");
subscriber=ContentSubscription.subscribe(session, xq.getMapFilter());
```

Because the filter model is limited to conjunctive filters and a limited set of operators, only a subset of the XQuery language is implemented.

These examples illustrated how messages objects can be directly constructed and accessed in service code. The next section shows how arbitrary Java objects can be directly used in messages.

5.2 Object Externalization

The metadata tag @mcSerialize instructs the precompiler to generate the necessary externalization code for the class ChatMessage:

```
@mcSerialize
public class ChatMessage {
  public String text;
}
```

Objects of this class can now be directly used as arguments for the send method:

```
publisher.send(new ChatMessage(text));
```

Receivers can obtain the ChatMessage object using the getObject method of Message.

```
public void received(Message msg, ...) {
  ... (ChatMessage)msg.getObject();
}
```

5.2.1 Filter Objects

The precompiler also supports the automatic generation of filter classes. Filters can be generated for all serializable classes that also specify the @mcFilter attribute:

```
@mcFilter
@mcSerialize
public class ChatMessage {
  public String text;
}
```

This filter class can now be used as follows to make a subscription:

```
ChatMessageFilter filter=new ChatMessageFilter();
filter.text="IMPORTANT";
filter._op_text=filter.CONTAINS;
subscriber=ContentSubscription.subscribe(session, filter);
```

5.3 Remote Method Calls

This section discusses an implementation of the chat program based on Remote Method Calls. Remote method calls rely on additional client and server stub classes that are generated by the precompiler. The @mcRemote tag indicates, that the compiler should generate client and server stub classes for the following class or interface.

```
@mcRemote
class ChatService extends Service {
  @mcMethod
  public void chatMessage(String text) {
    // process message
  }
}
```

On the “server side”, the object is connected to the channel chat-rmc with the following operation:

```
Signal.connect(getSession().subscribe("lan", "chat−rmc"), this);
```

On the “client side”, a distributed object (i.e., a client stub) is created an connected to the same channel:

```
doChat=new DoChatService();
Signal.connect(doChat, getSession().publish("lan", "chat−rmc"));
```

A client can now call the chatMessage method on the distributed object to send a message. Methods in the client stub accept an additional parameter that permits to select between synchronous, asynchronous and oneway calls. In this case the choice is *oneway*, because it is not important that the call blocks and if more than one peer is present, then the call might yield multiple results. (The current implementation would only uses the first result and drop other responses.)

```
doChat.chatMessage(text, ClientStub.ONEWAY);
```

5.3.1 Output Interfaces

To send messages to remote parties, the previous example used a remote reference variable. A drawback of this solution is that the proxy object is created in client code and the outgoing connections are not explicitly defined in the interface of the service.

```
@mcRemote
class IChat {
  public void chatMessage(String text);
}

class ChatService extends Service implements IChat emits IChat {
  public void chatMessage(String text) {
    // handle message
  }
}
```

Two local chat services this and otherInstance can be interconnected with the following connect statements:

```
Signal.connect(this, otherInstance);
Signal.connect(otherInstance, this);
```

Interfaces can also be connected to subscribers and output interfaces can be connected to publishers:

```
Signal.connect(getSession().subscribe("lan", "chat-rmc"), this);
Signal.connect(this, getSession().publish("lan", "chat-rmc"));
```

All chat examples shown so far explicitly connect the input and output ports of the service in the init method, which is called at service initialization time. When the last concept with explicit input and output interfaces is used, a greater degree of decoupling can be achieved. The service can be connected from outside, by an external coordinator component and does not make subscriptions or advertisements by itself.

6 Evaluation

The first series of tests compares the performance of MundoCore with other systems (Figure 3). The tests were conducted on two Athlon 64 systems with 2,0 GHz running Windows XP, interconnected by a 1 GBit/s network.

Publish/Subscribe (Test A): This test measures the roundtrip time from a client to a server process. Because brokers are implemented as external processes in SwiftMQ (A1), it has additional communication costs. This is the main reason why MundoCore (A2) performs better in this test. Overall, the results show that the publish/subscribe system of MundoCore provides a good performance compared to an industry-strength product.

Publish/Subscribe with XML (Test B): This test is similar to the previous one, but uses XML for message encoding. MundoCore (B2) was compared with JXTA 2.3.1 [34] (B1). Since JXTA generally uses XML to encode messages, MundoCore was configured to use the XML/SOAP protocol for better comparability. In JXTA, publish/subscribe was emulated by using pipes. The performance is quite similar, because both make use of the same XML parser from the Apache Xerces project and spend most of the time with XML processing. However, while communication over JXTA pipes is relatively efficient, the discovery of the pipes beforehand is very inefficient, as later tests will show.

XML Remote Procedure Calls (Test C): Sun's Java Web Service Developer Pack 2.0 (C1) is compared with MundoCore Java (C2) and MundoCore C++ (C3). Again, the Java-based

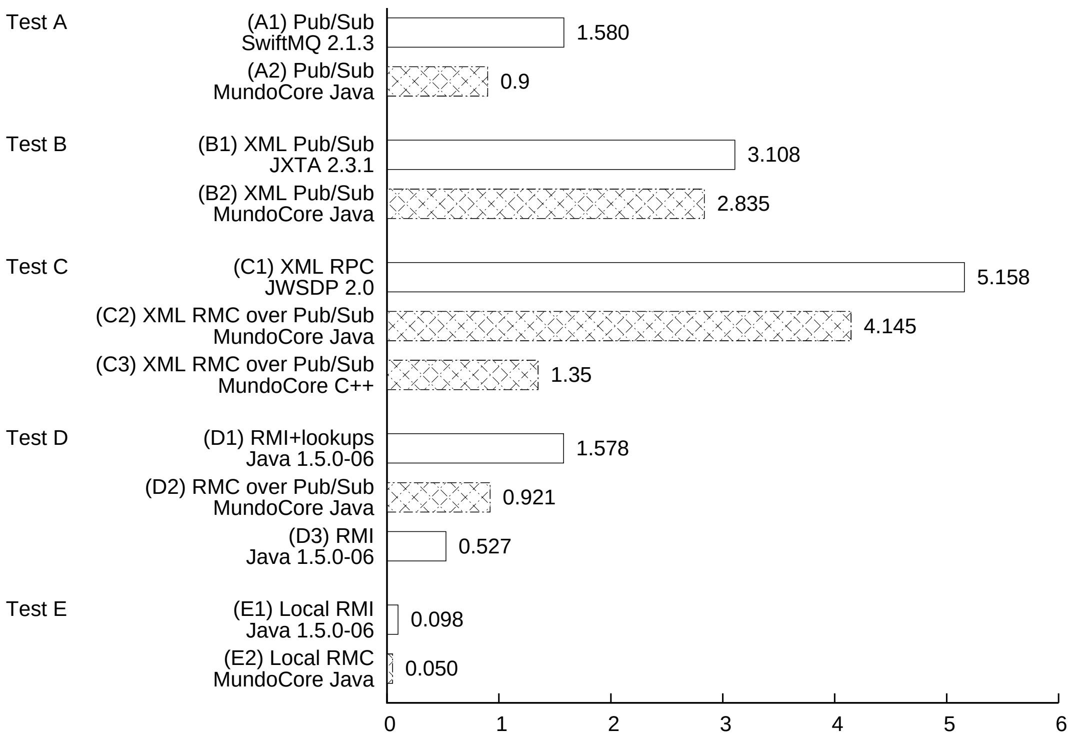

Figure 3: Comparison with Other Systems (Response Time)

programs are based on the same XML parser. MundoCore Java is slightly faster due to more efficient object adapters and stubs. The C++ program is faster by a factor of three, because it does not use a VM and because of a custom-built, speed-optimized XML parser.

Remote Method Calls (Test D): MundoCore RMCs are typically used in conjunction with publish/subscribe to get the benefit of execution transparency. The performance of a simple Java RMI (D3) was compared with an RMC (D2) and RMI with naming lookups to emulate the same transparency (D1). RMI is faster than MundoCore in this test, because RMI uses VM-level optimizations, has a much simpler, linear message flow, and performs less context switches.

Local Calls (Test E): When client and server reside within the same VM, MundoCore (E2) performs calls twice as fast as RMI (E1). MundoCore applies protocol handlers only if necessary, while RMI requests and replies always have to pass through the whole protocol stack.

Overall, the results confirm that the concept of protocol heaps does not have a negative impact on the performance, compared to the simpler layered architectures. Figure 4 shows the test results from several embedded platforms.

e830 (Test F): A Toshiba e830 PDA with an Intel PXA272 CPU at 624 MHz running Windows CE 4.2. The comparison shows a C++ server process using the binary protocol (F1), XML protocol (F2), and a Java (J2ME/CDC) server process running on the NSIcom CrE-ME 4.00 VM using the binary protocol (F3), and the XML protocol (F4). The advantage of C++ over Java increases on embedded platforms, because efficient Just-In-Time compilation would require a considerable amount of additional memory, which is often not available.

Triton (Test G): The Ka-Ro Triton 2 is a complete embedded system on a DIMM module. It is based on an Intel PXA255 CPU at 400 MHz and runs Linux. The figure shows the performance

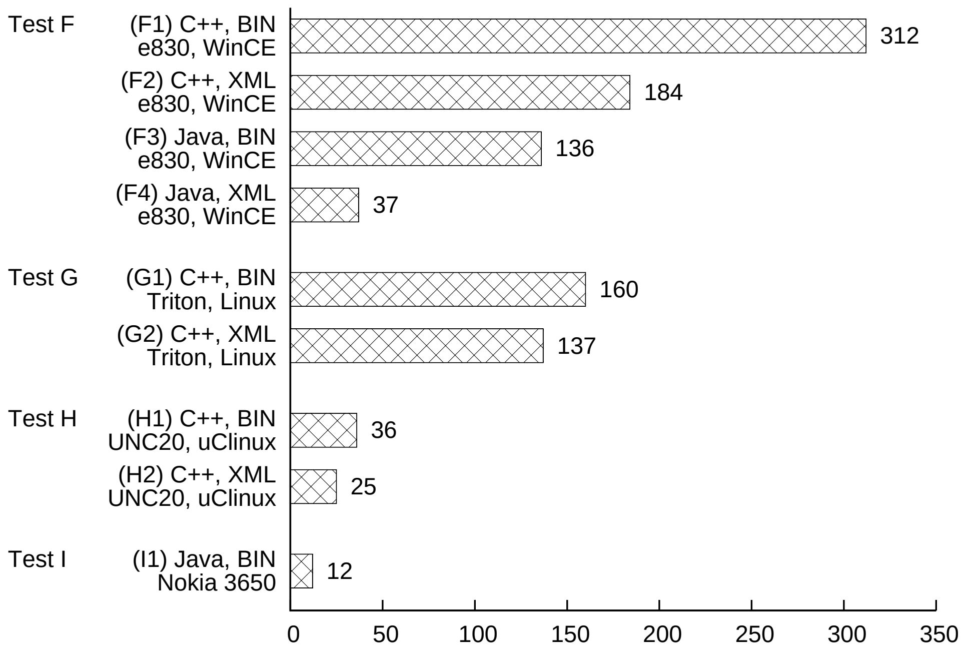

Figure 4: Performance on Embedded Systems (Throughput)

of a C++ server process using the binary (G1) and XML (G2) protocol.

UNC20 (Test H): The FS-Forth UNC20 is a complete system on a small DIL module. It is based on an NetSilicon 7520 CPU at 55 MHz and runs uClinux.

Nokia 3650 (Test I): A Java J2ME/CLDC server process running on a Nokia 3650 cellphone using the binary protocol (F1).

6.1 Discovery

The performance of peer discovery was evaluated in MundoCore and JXTA 2.3.1. Each instance of the test program advertises a communication endpoint and listens for incoming messages. Next, the program attempts to discover the endpoints of all other instances and 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. In addition, the timestamps are used for performance comparison. The MundoCore 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. JXTA does not use a publish/subscribe abstraction. However, the task of discovering pipes in JXTA is equivalent. The JXTA test program creates a unicast pipe and advertises it once using the discovery service. To discover the other pipes, the discovery service is called with a polling interval of 1 or 5 seconds.

The tests were conducted on four Athlon 64 PCs with 2.0 GHz running Linux, interconnected with GBit-Ethernet. On each machine, eight processes were started. This gives a total of 32 processes and 992 discovery operations and message passes.

As the results in Figure 5 show, the performance of MundoCore is considerably better and JXTA is only able to discover about 80% of the communication endpoints. The behavior of JXTA is due to its group concept which does not guarantee that all peers can see each other.

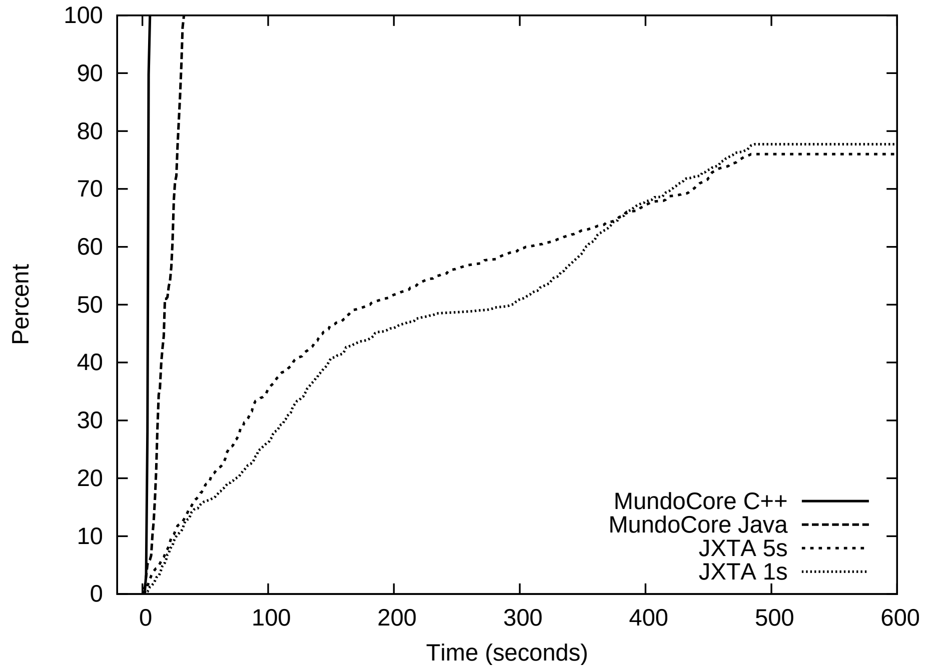

Figure 5: Discovery Performance of MundoCore and JXTA

However, in pervasive computing it is often required to discover resources within a defined, limited scope in a reliable manner. Thus, JXTA is not suitable for applications that require a good *quality of peer discovery.*

7 Monitoring

The MundoCore Inspect tool is a SWING-based application to monitor the running system. This application connects to the debug service of an arbitrary remote node and permits to view information about the running services. This tool allows to

- list the current routing table of a node and the states of the corresponding connections
- list the imports and exports tables of the publish/subscribe brokers. The imports table contains a list of all channels for which local subscriptions exist. The exports table lists channels along with all adjacent nodes interested in receiving messages from this channel.
- inspect messages
- inject messages into the publish/subscribe system
- call methods on a service with a generic RMC client
- deploy, start, and stop services
- view service configuration information and reconfigure services

8 Related Work

Conventional middleware systems like CORBA [4], Java RMI [15], or DCOM [13] were developed to provide unified access to remote objects independent of hardware architectures, operating systems and programming languages. They assume a mostly stable network environment and treat service unavailability as an error. Because conventional middleware is based on a monolithic design and aims to provide as many services as possible, it is resource intensive and not suitable for small devices.

Resource-poor devices are addressed by shrinked versions of conventional middleware systems, e.g. Minimum CORBA [21]. However, providing only a functional subset often leads to a different programming model. Because Minimum CORBA lacks dynamic invocation, extended methods not specified in the shrinked interfaces become inaccessible. Furthermore, the CORBA IIOP and Java JRMP protocols are not self-describing, i.e., an interface repository must be accessed at run time in order to understand the structure of invocation messages and to perform dynamic invocations. In MundoCore, invocation messages contain all necessary type information. In contrast to CORBA and RMI, dynamic invocation and dynamic stubs are considered to be the basic functionalities and are made available on all platforms. This way, even MundoCore services running on limited Java profiles can access all available services and methods in the distributed system.

Communication frameworks designed for small devices often also lack essential features like object serialization and remote method invocations. This otherwise common functionality then has to be built into the application code. The connector abstraction in MundoCore and the platform-dependent, automatic generation of connector code by the precompiler enable a much more consistent API across different platforms.

Reconfigurable middleware (e.g. [6, 18, 19, 25]) can adapt its behavior to different environments and application requirements. Such middleware has additional interfaces allowing applications to change the strategies for e.g. 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 [6] and UIC [25]. These two systems will be described in more detail in the following.

The development of UIC (Universally Interoperable Core) [25] is related to the *Gaia* [24] component-based operating system designed for active spaces. UIC defines a skeleton based on abstract components, which encapsulate the standard functionality aspects common to most object request brokers. Concrete dynamically loadable components specialize these abstract components to implement the properties required for particular middleware platforms, devices, and networks. UIC must be specialized to meet the particular application's requirements. A particular instance is denoted as a personality after the specialization.

BASE [6] is a micro-broker-based middleware developed at the University of Stuttgart. The micro-broker is the central component of the architecture that implements only a minimal functionality. It is responsible for dispatching *invocation objects* between local services, or to a transport plug-in, which transports the invocation to a remote broker. A minimum configuration of the middleware can be run on resource-constrained devices. Resource-rich environments can augment the functionality by adding plug-ins to the system. The prototype of BASE is implemented in Java and relies only on the functionality provided by the Java J2ME/CLDC profile. In contrast to that, we have five different MundoCore libraries (for Java 1.1, 1.2, 1.3, 1.4, and 1.5). Each library makes optimal use of the underlying Java framework, which results in a higher quality of service by the middleware. For example, MundoCore's node discovery works better on Java 1.4 and 1.5, because of improved Java networking APIs. The MundoCore precompiler allows us to generate these different versions from a single source code.

Compared to reconfigurable middleware systems, the concept of protocol heaps is a unique feature of MundoCore. Other systems either do not support a fine-grained control of message transport, or they do not support adaptivity. While UIC and BASE allow to build adaptive systems, reconfigurations always have a global impact on a node. A commonly used concept are Interceptors [27]. However, they offer no or only little control of how messages flow through the middleware services. In iBus [5], the order in which protocol handlers are applied is strictly defined by the stack and each stack ends in a specific TCP or UDP port. This leaves no room for adaptivity. MundoCore permits applications to customize the message transport on a much more fine-grained level by defining service-specific communication stacks. The concept of protocol heaps is described in [8]. However, the authors do not provide implementation details or efficient algorithms for determining the message flow. A similar approach to the connector abstraction in MundoCore is described in [3]. MundoCore supports both adaptivity and application-specific protocol stacks.

one.world [14] represents all data as tuples, which define a common data model, including a type system and expresses all communication through asynchronous events. Tuples are self-describing, such that applications can dynamically inspect their structure and contents. MundoCore uses a similar approach with its message structure. However, one.world tuples build on Java classes as types, resulting in a programmatic data model which suffers from interoperability problems. MundoCore clearly distinguishes between active messages (a programmatic data model) and passive messages (a data-centric model) and clearly defines the transformations between these two representations.

Distributed event-based systems, such as JMS [31], Elvin [28], Rebeca [11], or Siena [9], are mainly designed for the Internet and do not perform well in mobile settings. More recent work addresses temporary disconnections [32], durable subscriptions [7], roaming [36], and extending such systems using interception [10]. Since publish/subscribe is an important foundation of MundoCore, it has many commonalities with distributed event-based systems. However, MundoCore permits a more flexible configuration of the communication stack and offers a seamless integration of the distributed object-oriented programming paradigm.

Most other systems also lack zero-configuration capabilities. For example, JXTA [34] requires dynamic configuration data and user interaction. 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. Pastry [26] requires per-node configuration information as well. Each instance must be assigned a local port number, and the hostname and port number of a boot peer must be specified.

9 Conclusion

MundoCore is a novel communication middleware specifically designed to meet the requirements of pervasive computing. In a nutshell, MundoCore has two distinctive features: i) it scales from servers and PCs all the way down to resource restricted embedded hardware and sensors, featuring a very small footprint and thereby high efficiency - without compromising the capabilities, i.e., application potentials by any restrictions; ii) it provides a distinctive harmonization of all the important communication abstractions needed in advanced distributed smart environments:

DOOP (distributed object oriented programming), publish/subscribe, peer-to-peer, and multimedia communication, i.e., streaming.

These distinctive 'outside' features are realized by way of a considerable number of 'inside' concepts. On the highest level, these concepts can be summarized as follows. The first feature - small footprint - is mainly supported through a dynamically reconfigurable architecture with 'loadable' services, the 'connector' abstraction for component separation, and the data structures and algorithms that realize the 'protocol heap' approach; the autoconfiguration flexibility comes at very low cost, namely complexity O(1). The second feature - harmonization of programming abstractions - is based on both the layer model and a concise way of building these abstraction atop the core publish/subscribe concept, combined with small but important extensions to the host programming languages.

The evaluation as described in the present paper provides a glance at the efficiency and sophistication of MundoCore. In addition to that, the number of applications, prototypes, and services built atop MundoCore to date - a total of about 50 - represent vital proofs of the quality of both the approach and its realization.

The monitoring approach described is also just one example from a coordinated set of development tools. These tools - for design, test, performance evaluation, etc. - complement standard software development aids in cases where the development of pervasive computing applications requires special care. For instance, integrated modeling / test tools support visual design and testing of intelligent environments, both in 2D (e.g., visualizing application components on a floor plan) and in 3D (visualizing applications that require a geometric model of the environment). All in one, MundoCore and the development tools, together with a few off-the-shelf tools, represent a full-featured distributed programming environment.

MundoCore has been released under an open-source license and can be downloaded from the homepage of our research group.

References

[1] Behnaam Aazhang and Joseph R. Cavallaro. Multitier Wireless Communications. *Wireless Personal Communications*, 17(2–3):323–330, June 2001.

[2] Erwin Aitenbichler. *System Support for Ubiquitous Computing.* PhD thesis, Darmstadt University of Technology, 2006.

[3] Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Language Support for Connector Abstractions. In *Proceedings of ECOOP 2003*, volume 2743 of *LNCS*, pages 74–102, 2003.

[4] M. Aleksy, Axel Korthaus, and Martin Schader. *Implementing Distributed Systems with Java and CORBA*. Springer, Berlin, 2005.

[5] M. Altherr, M. Erzberger, and S. Maffeis. iBus - A Software Bus Middleware for the Java Platform. In *International Workshop on Reliable Middleware Systems*, pages 43–53, 1999.

[6] Christian Becker, Gregor Schiele, Holger Gubbels, and Kurt Rothermel. BASE - A Micro-Broker-Based Middleware for Pervasive Computing. In *First IEEE International Conference on Pervasive Computing and Communications (PerCom'03)*, pages 443–451. IEEE Computer Society, 2003.

[7] Sumeer Bhola, Yuanyuan Zhao, and Joshua Auerbach. Scalably Supporting Durable Subscriptions in a Publish/Subscribe System. In *International Conference on Dependable Systems and Networks (DSN)*, pages 57–66. IEEE Computer Society, June 2003.

[8] Robert Braden, Ted Faber, and Mark Handley. From Protocol Stack to Protocol Heap - Role-Based Architectures. *ACM SIGCOMM Computer Communications Review*, 33(1):17–22, January 2003.

[9] Antonio Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks.* PhD thesis, Politecnico di Milano, Milano, Italy, 1998.

[10] Edward Curry, Desmond Chambers, and Gerard Lyons. Extending Message-Oriented Middleware using Interception. In *3rd International Workshop on Distributed Event-Based Systems (DEBS'04)*, 2004.

[11] Ludger Fiege, Gero Mühl, and Felix C. Gärtner. A Modular Approach to Build Structured Event-based Systems. In *ACM Symposium on Applied Computing (SAC)*, pages 385–392. ACM Press, 2002.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley Professional, 1995.

[13] Richard Grimes. *Professional DCOM Programming.* Wrox Press, 1999.

[14] Robert Grimm. One.world: Experiences with a Pervasive Computing Architecture. *Pervasive*, 3(3):22–30, July 2004.

[15] William Grosso. *Java RMI. Designing and Building Distributed Applications.* O'Reilly Media, 2001.

[16] Arnaud Le Hors, Philippe Le Hgaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document Object Model (DOM) Level 2 Core Specification. `http://www.w3.org/TR/DOM-Level-2-Core/`, November 2000.

[17] IANA. MIME Media Types. `http://www.iana.org/assignments/media-types/`, August 2005.

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

[19] Fabio Kon, Manuel Roman, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhaes, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamic TAO Reflective ORB. In *Middleware 2000: IFIP/ACM International Conference on Distributed Systems Platforms*, volume 1795 of *LNCS*, pages 121–134. Springer, 2000.

[20] Charles M. Kozierok. The TCP/IP Guide. `http://www.tcpipguide.com/free/t_TCPIPArchitectureandtheTCPIPModel.htm`, 2001.

[21] OMG - The Object Management Group Inc. The Minimum CORBA Specification. `http://www.omg.org/cgi-bin/doc?formal/02-08-01`, August 2002.

[22] Bhaskaran Raman, Pravin Bhagwat, and Srinivasan Seshan. Arguments for Cross-Layer Optimizations in Bluetooth Scatternets. In *Symposium on Applications and the Internet (SAINT01)*, pages 176–184. IEEE Computer Society, January 2001.

[23] Sylvia Ratnasamy, Paul Francis, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *SIGCOMM 2001*, pages 161–172. ACM Press, 2001.

[24] Manuel Roman, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. A Middleware Infrastructure for Active Spaces. *Pervasive*, 1(4):74–83, October 2002.

[25] Manuel Roman, Fabio Kon, and Roy H. Campbell. Reflective Middleware: From Your Desk to Your Hand. *IEEE Distributed Systems Online Journal, Special Issue on Reflective Middleware*, 2001.

[26] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms*, volume 2218 of *LNCS*, pages 329–350. Springer, November 2001.

[27] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Vol.2 : Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000.

[28] Bill Segall and David Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of AUUG97 (Online Proceedings)*, Brisbane, Australia, 1997.

[29] Ralf Steinmetz and Klaus Wehrle. *Peer-to-Peer Systems and Applications*. Springer, 2005.

[30] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[31] Sun Microsystems. Java Message Service. `http://java.sun.com/products/jms`, 2006.

[32] Peter Sutton, Rhys Arkins, and Bill Segall. Supporting Disconnectedness - Transparent Information Delivery for Mobile and Invisible Computing. In *First International Symposium on Cluster Computing and the Grid*, pages 277–285. IEEE Computer Society, May 2001.

[33] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 1997.

[34] Bernard Traversat, Ahkil Arora, Mohamed Abdelaziz, Mike Duigou, Carl Haywood, Jean-Christophe Hugly, Eric Pouyoul, and Bill Yeager. Project JXTA 2.0 Super-Peer Virtual Network. `http://java.sun.com/othertech/jxta/`, May 2003.

[35] Ronald E. Wyllys and Philip Doty. Notes on the 5-Layer and 7-Layer Models of Interconnection. `http://www.gslis.utexas.edu/~l38613dw/readings/NotesOnInterconnection.html`, 2000.

[36] Andreas Zeidler. *A Distributed Publish/Subscribe Notification Service for Pervasive Environments*. PhD thesis, Department of Computer Science, Darmstadt University of Technology, November 2004.