The BASE Plug-in Architecture - Composable Communication Support for Pervasive Systems

M. Handte, S. Wagner
Networked Embedded Systems Group,
University Duisburg-Essen, Germany
{first.last}@uni-due.de

G. Schiele, C. Becker
Information Systems II,
University Mannheim, Germany
{first.last}@uni-mannheim.de

P. J. Marrón
Networked Embedded Systems Group,
University Duisburg-Essen, Germany
{pjmarron}@uni-due.de

ABSTRACT

Pervasive computing is based upon spontaneously networked devices that are invisibly integrated into everyday objects. Due to the integration and mobility of devices, pervasive systems can be highly heterogeneous and dynamic. To enable cost-effective application development despite such harsh conditions, we have developed the BASE middleware. In its original implementation, BASE relied on monolithic plug-ins to provide tailored support for different communication abstractions, protocols and technologies. In this paper, we describe and evaluate a major architectural redesign that introduces modular plug-ins. The key benefits of the new architecture are increased flexibility and code reuse due to the efficient runtime composition of plug-ins.

1. INTRODUCTION

Pervasive computing envisions context-aware applications that provide seamless and distraction-free support for user tasks. To achieve this, pervasive applications leverage the distinct capabilities of multiple devices that are invisibly integrated into all kinds of everyday objects. Due to their integration, devices are heterogeneous ranging from specialized resource-poor embedded systems to powerful servers. In addition, due to device mobility and failures, pervasive systems may exhibit a high degree of dynamics. As a result, enabling cost-effective application development requires middleware that takes care of the low-level issues of interaction.

In the past, researchers have developed a diverse set of specialized middleware systems for different pervasive computing scenarios. Due to the variety of system parameters and application requirements, they can differ drastically with respect to the used communication abstractions (e.g. events vs. RPC), protocols (e.g. encrypted vs. unencrypted) and technologies (e.g. Bluetooth vs. IP-based). Usually, it is not possible to contrast and rate these middleware systems as their suitability for an application depends on the target scenario. As alternative to specialization, it is possible to opt for configurability as underlying middleware design principle. So instead of supporting a fixed set of abstractions, protocols and technologies, the middleware provides a configurable and extensible set which improves the overall applicability of the middleware core and API.

Configurability has been a key design rationale behind the BASE [4] middleware ever since the development of the first prototype in 2002. There, we started out with a minimal yet extensible middleware core. To support communication, the core can be configured with monolithic plug-ins. Although being generic, a monolithic approach suffers two main drawbacks. First, it requires the developer to implement a complete communication stack in every single plug-in. As a result, even the replacement of a part of the stack is a complicated undertaking. Secondly, as each plug-in implements a static stack, integrating optional orthogonal aspects such as routing or compression causes an exponential increase in the number of plug-ins to support all combinations.

In this paper, we present a major redesign of the BASE plug-in architecture. As a solution to the aforementioned issues, the redesign introduces plug-in layers that split monolithic plug-ins in a way that facilitates composition. An automatic composition process assembles communication stacks from fine-grained plug-ins. To control the process, plug-ins and applications can specify composition requirements. The evaluation results indicate that the overall approach is capable of providing the targeted benefits at an acceptable cost.

The remainder of this paper is structured as follows. Next, we briefly describe the overall architecture of BASE. In Section 3, we provide a detailed discussion of the redesigned plug-in architecture. In Section 4, we present use-cases to describe the benefits of the architecture. We discuss an experimental evaluation of the overheads and we identify and quantify optimization potentials. In Section 5, we describe related work and in Section 6, we conclude the paper.

2. BASE ARCHITECTURE

In this section, we describe the architecture of BASE. For brevity, we introduce the main building blocks and outline their interaction. A detailed description can be found in [4] and [8]. Figure 1 depicts the building blocks and groups them into the *application*, the *micro-broker* and the *plug-in layer*.

2.1 Micro-broker Layer

The core of the system is formed by the micro-broker. In the style of micro-kernel based operating systems, the

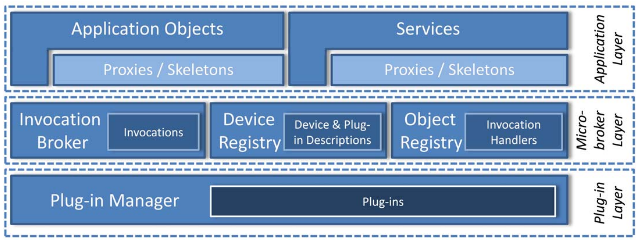


Figure 1: BASE Architecture

micro-broker implements a minimal and generic middleware core. This core consists of the invocation broker, the device registry and the object registry. The invocation broker is responsible for dispatching so-called invocation objects or simply invocations. Invocations represent an interaction between application objects such as a local or remote method invocation. To decouple the application from invocation processing, the broker implements different synchronization alternatives. Furthermore, it assigns globally unique invocation identifiers, allowing the parallel mediation. Finally, it keeps a table of ongoing invocations to support different execution semantics.

If an invocation should be dispatched to a local application object, the invocation broker retrieves the object's invocation handler from the object registry and forwards the invocation. When a handler is registered at the object registry, the registry associates a locally unique object identifier (OID) with the handler and stores the mapping for later retrieval. The range of OIDs is statically divided into generated and well-known OIDs which can be used to implement bootstrap services.

Remote invocations are forwarded to the plug-in manager which takes care of communication. To determine the appropriate plug-ins for transmission, the manager must compute a compatible set of plug-ins. This requires knowledge about the plug-ins available on the local and the remote system. This knowledge is maintained in the device registry in the form of plug-in descriptions and it is updated as part of the device discovery. In addition, the device registry stores a device description for each device which consists of a logical device identifier and a list of the well-known services registered at the corresponding device. Thus, applications and services can query the device registry for objects with well-known OIDs.

2.2 Application Layer

At the application layer, application objects and middleware services are interacting with each other using the micro-broker. In order to write an object that can receive an invocation, a programmer must implement an invocation handler that accepts invocations from the broker. The handler must then be registered at the object registry. Other objects that know the OID can then interact with it by manually creating invocations and passing them to the broker. However, BASE provides stub generators to ease this task.

In addition, BASE includes a number of optional services. One such service is the BASE service registry [4], which allows searching for services by name or type. Another example is the PCOM component container [3] which realizes a component system to support adaptive applications.

3. BASE PLUG-IN ARCHITECTURE

The two functions provided by the plug-in layer are device discovery and remote communication. The implementation of these functions is not only dependent on the targeted communication technology, but also on the targeted protocols and abstractions. Thereby, it is important to note that the communication technologies are usually determined by the hardware capabilities of a device. The suitability of a communication abstraction and protocol, however, is usually dependent on the interaction patterns of the application.

The BASE plug-in layer encapsulates these functionalities in plug-ins so that the middleware can be configured for a particular device with a set of applications. To configure BASE, a developer registers the desired plug-ins at the plug-in manager. The plug-in manager forms the generic core of the plug-in layer and it mediates the interaction with and between plug-ins. As a consequence, the redesign of the plug-in architecture primarily affects the plug-in manager. To motivate the design, we first discuss the goals. Thereafter, we present the redesigned plug-in manager and we organize plug-ins into layers. Finally, we show how the manager composes plug-ins and discuss how multi-layer plug-ins support improvements.

3.1 Design Goals

The design goals are easily derived from the functional goal of supporting composable communication and the constraints introduced by pervasive computing. Note that the design goals are not orthogonal, so we must strike an acceptable balance.

- *Flexibility*: Our main reason for the redesign is to modularize plug-ins so that their individual functionality can be composed dynamically at runtime. This eliminates the need to compose a communication stack statically which, in turn, enables the reuse of plug-ins in different stacks. To maximize this benefit, a key goal is support for a high degree of compositional flexibility.

- *Simplicity*: The architecture should be simple to use for both, application and plug-in developers. This avoids high learning effort and reduces programming failures. From the application developer's perspective, it should be possible to specify communication requirements with different levels of detail. From the plug-in developer's perspective, it should be possible to develop a plug-in without reasoning about others.

- *Efficiency*: To be applicable to a broad range of scenarios, the design should also be efficient. To resolve the resulting conflict with the other goals, we rely on configurability. To avoid a pre-defined and static balance of trade-offs, we enable developers to balance them during plug-in development and device configuration.

3.2 Plug-in Manager

The generic core of the plug-in architecture is formed by the plug-in manager. As depicted in Figure 2, the plug-in manager mediates the interaction between the device registry and the invocation broker, and the plug-ins. From a high-level perspective the plug-ins can be classified in discovery and communication plug-ins. The plug-in manager provides methods to install and remove plug-ins at runtime and it keeps references to the installed plug-ins. When a

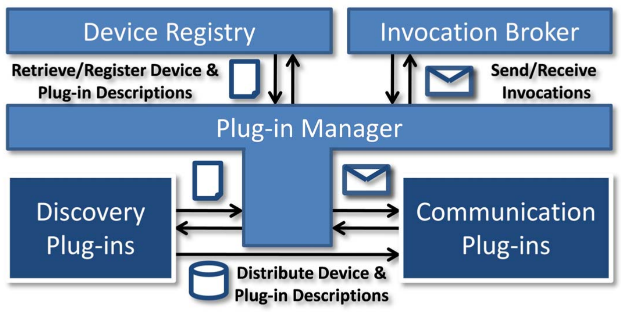


Figure 2: BASE Plug-in Manager

new plug-in is installed, the plug-in manager exposes a part of its functionality to the plug-in. To do so, it passes one or more references to its interfaces to the plug-in. The interfaces that are exposed depend on the type of the plug-in. After the references have been set, the plug-in manager calls a method on the plug-in to signal that it may start its operation. Thereby, a plug-in will typically allocate resources such as buffers in memory or sockets. After initialization, the plug-in manager retrieves the plug-in description. This description contains a type identifier to differentiate plug-ins and it may contain several key-value pairs that can be updated at runtime. The key-value pairs describe the characteristics that are required by other plug-ins of the same type. Examples are the address and the port of an IP plug-in or the public key of an encryption plug-in. After retrieving it, the plug-in manager registers the description at the device registry. When the plug-in is removed, the manager removes the description, stops the plugin and releases all references so that it can be collected as garbage.

At runtime, a discovery plug-in uses the plug-in manager to retrieve the local device and plug-in descriptions. Furthermore, it may retrieve the plug-ins that can be used to distribute the descriptions. Using these plug-ins, a discovery plug-in may implement an arbitrary distribution strategy. For example, it may distribute the descriptions proactively using flooding.

To enable remote communication, the plug-in manager mediates the interaction between the broker and the plug-ins. If an invocation shall be transmitted, the broker forwards the invocation to the plug-in manager. The plug-in manager then uses the plug-ins to perform the transmission. Alternatively, if the plug-ins cannot be used, e.g. because the plug-ins cannot perform the transmission, it informs the invocation broker. If an invocation is received by the plug-ins, they may forward it to the plug-in manager, which in turn delivers it to the invocation broker. The invocation broker may then dispatch the invocation which may result in further invocations, for example, to return the result of a remote method invocation.

3.3 Plug-in Layers

The previous discussion follows the original architecture of BASE. Yet, to support the runtime composition of communication stacks, we need to separate the stack into blocks. To do this, we can isolate communication technologies, protocols and abstractions as depicted in Figure 3.

Underneath the middleware, the operating system manages the available network interfaces and provides an API which may be specific for a certain technology that must be unified. The key problem here is to identify an interface that is lean enough to be implemented easily but still efficient. Due to the fact that discovery plug-ins need to distribute device and plug-in descriptions among all available devices and since many communication abstractions need to transmit larger amounts of data reliably, we decided to use a socket-style interface that supports unreliable packet-oriented communication with local broadcast semantic for discovery and connection-oriented communication for the transmission of invocations.

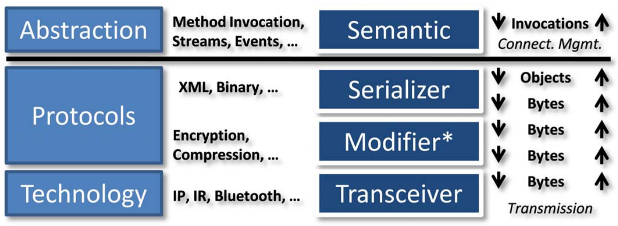


Figure 3: BASE Plug-in Layers

As a result, the protocols above need to operate on the byte-streams of the connection. Since BASE is built on top of an object-oriented programming language (Java), we can classify the protocols in ones that solely modify the byte-stream and ones that (de-) serialize objects. A simple example for the first is data compression that performs run-length encoding on chunks of input data. An example for the later would be an CDR based serializer. Obviously, it is possible to stack the protocols that solely manipulate byte-streams in an arbitrary order. However, there can only be one serializer protocol per stack. Above the technology and protocols, the communication abstraction determines how an invocation should be distributed. Thereby, it is inappropriate to associate the abstraction with a particular connection since abstractions such as publish-subscribe may require several. Consequently, developers of communication abstractions must be able to manage the connection establishment. So in contrast to protocols which are passively stacked on top of a technology, abstractions actively request the initialization of a stack.

In summary, this results in four different plug-in layers shown in Figure 3. At the lowest level, transceivers provide packet-oriented discovery and connection-oriented communication. Above the transceiver, an arbitrary number of modifiers implement protocols that modify the byte-streams. Above these protocols, a serializer is responsible for the serialization and deserialization of objects. Finally, at the highest layer, the semantic requests connections that are created by stacking transceivers, modifiers and serializers on top of each other.

3.4 Plug-in Composition

To achieve flexibility and usability simultaneously, it is necessary to automate the composition of plug-ins while allowing manual control whenever necessary. To allow varying degrees of control, we extend invocations with a configuration object. This object specifies the requirements on the communication stack that shall be composed. To avoid unnecessary restrictions on the flexibility of compositions, the internal structure of the configuration object corresponds to

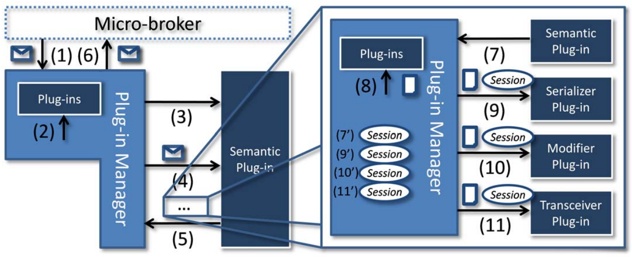


Figure 4: BASE Plug-in Negotiation

a list of requirements that is ordered according to the stack. This means that the head of the list specifies requirements on the semantic whereas the tail usually specifies requirements on the transceiver. The configuration object ensures that requirements are added according to the restrictions setup by the layers, e.g., it ensures that requirements on modifiers are not added above the serializer. In addition, the list structure also enables the developer to refrain from specifying requirements on a certain layer. However, as a minimum, the developer must specify the requirements on the semantic since they define the meaning of the invocation.

The requirements can be specified at various levels of details ranging from explicit requests for a particular type of plug-in to abstract requests for an implementation-specific characteristic. As a simple example for such a characteristic consider the bit-length of the encryption key. As the plug-in manager is supposed to be generic, it cannot understand the intrinsic meanings of all plug-in implementations. Furthermore, since the correctness of a plug-in may rely on the presence of plug-ins with certain characteristics at lower layers, BASE enables plug-ins to refine the requirements on plug-ins at lower layers.

To deal with these two issues, we separate the composition into a negotiation phase during which a set of plug-ins is computed and a connection phase during which a negotiated set of plug-ins is connected. Since we want to enable plug-ins to refine the requirements on lower layers, the negotiation phase must be performed top-down as depicted on the right side of Figure 4. The connection phase, on the contrary, should be performed in a bottom-up fashion as shown in Figure 5 to simplify the plug-in interface. In the following, we describe the details of these phases. For simplicity, we assume that the invocation represents a remote method invocation that returns a result. Other abstractions work analogously.

When an invocation must be transmitted, the micro-broker forwards it to the plug-in manger as shown in step (1) of Figure 4. Thereby, each invocation specifies its requirements on the stack by means of an attached configuration object. Using the invocation, the plug-in manager computes the candidate set of semantic plug-ins (2). If the requirements attached to the invocation are explicitly specifying a particular plug-in, the plug-in manager selects the plug-in. If the requirements are solely specifying implementation-specific characteristics, the plug-in manager selects all semantic plug-ins as targets. Then the plug-in manager asks each target plug-in whether it can fulfill the requirements specified by the configuration object (3). If the call fails, the next plug-in is selected. Otherwise, the plug-in manager aborts the process and forwards the invocation to the plug-in that acknowledged the requirements (4). If no semantic plug-in can be found, the plug-in manager signals this to the invocation broker which informs the application. After the semantic plug-in has received the invocation, it will transmit it to the remote system and after receiving the result, it forwards an invocation that represents the result to the plug-in manager (5). The plug-in manager in turn will forward the invocation to the broker (6) which will deliver the result, e.g. by releasing a blocked application thread.

3.4.1 Plug-in Negotiation

To transmit the invocation, the semantic plug-in can establish a connection. To do this, it must first negotiate a stack. This negotiation is shown on the right side of Figure 4. To start the negotiation (7), it passes three references to plug-in manager. The references encompass the system identifier of the target system, the configuration object and a reference to the requesting semantic plug-in. After receiving the negotiation request, the plug-in manager will compute a set of plug-ins that is available to both, the local and the target system. To do this, it queries the device registry for the plug-in descriptions of the remote device and it intersects the set with the plug-in descriptions of local plug-ins (8). Thereafter, the plug-in manager validates that the requesting semantic is also available on the remote device. If this is not the case, the negotiation fails immediately. Otherwise, the manager creates a session object for the request.

Similar to the requirements contained in a configuration object, session objects can be linked together as a list that represents a particular stack. However, a session object does not store abstract requirements but it stores a concrete plug-in identifier. Consequently, a list of session objects can be used to describe a stack for a particular system. In addition, the session objects can also store a set of plug-in specific parameters that are required to establish a connection. This set of parameters is divided into local parameters required on the client-side and remote parameters needed on the server-side. A simple example of a parameter is the end-point information of a transceiver. As explained later, the session object is transferred to the remote system and the parameters are automatically passed to the right plug-ins during connection initialization.

Once the session object has been created, the negotiation continues below the semantic layer. Using the list of requirements received from the semantic plug-in, the plug-in manager determines whether it is necessary to add a serializer to the stack. If a serializer is required, the plug-in manager will use the set of plug-ins that is available on both systems to determine target serializers. Similar to the selection of semantic plug-ins, the plug-in manager simply prunes the list in cases where a particular serializer is required. Thereafter, it goes through the list of target serializers and asks them whether they fulfill the requirements (9). While doing this, the plug-in manager does not simply pass the requirements to the serializer. Instead, it first creates a new session object with the identifier of the serializer and it copies the configuration object. Thereafter, it passes the new session object, the copy of the configuration object as well as the plug-in description of the same plug-in on the remote device to the target plug-in. The serializer plug-in can use the plug-in description to determine whether it can communicate with the remote plug-in and it may store relevant parameters in the session object. Furthermore, it may refine the require-

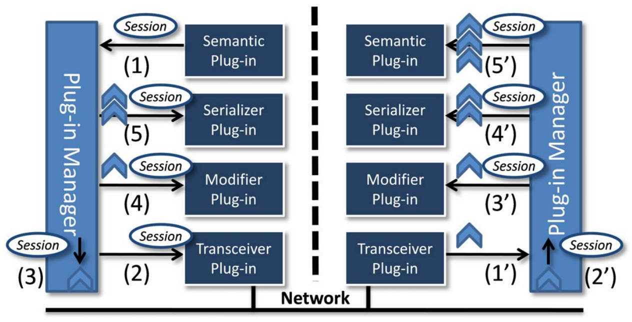


Figure 5: BASE Plug-in Connection

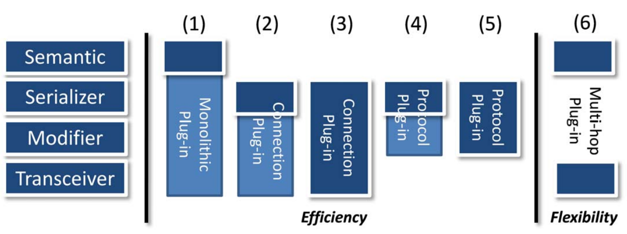


Figure 6: BASE Multi-Layer Plug-ins

ments specified in the configuration object in cases where it depends on certain plug-ins at lower layers. If the serializer responds positive to the negotiation request, the plug-in manager stores the session object and the negotiation continues with the refined requirements at a lower layer. If it responds negative, the negotiation continues with the next serializer from the target set using the original requirements and a new session object. Again, if a serializer is required but no serializer can fulfill the requirements, the complete negotiation fails.

Once a serializer has been determined successfully, the negation continues. Depending on the requirements, it may continue with a modifier or a transceiver. If a modifier is required, the procedure of determining an appropriate plug-in is simply replicated with modifiers (10) and transceivers (11). If the configuration fails at some point, the last successfully negotiated plug-in is invalidated. To do this, the session object is simply removed from the list and the negotiation continues.

Thus, this composition algorithm essentially resembles backtracking. The backtracking ensures that all possible compositions are systematically tested. However, it can also lead to an exponential runtime overhead. Yet, thrashing can only occur if devices are equipped with similar plug-ins at higher layers that require lower layer plug-ins which cannot be used. So far, we did not find this to be problematic in practice.

The negotiation phase ends when the transceiver plug-in has been selected which means that all requirements are fulfilled. The result of the negotiation is a linked list of session objects, i.e. (7'), (9'), (10'), (11'). This list stores the plug-in identifiers and the parameters needed during connection establishment for all layers below the semantic. By returning the top-most session object as the result of step (7), the semantic plug-in can finalize the composition by adding its own parameters.

3.4.2 Plug-in Connection

After the negotiation, the semantic can request the connection establishment using the session objects. To do this, it simply passes the session objects to the plug-in manager as depicted in step (1) on the left side of Figure 5. The plug-in manager uses the objects to connect the plug-ins in bottom-up fashion. It first calls the transceiver with the corresponding session object to establish the connection at the lowest layer (2). If the establishment succeeds, the transceiver returns a so-called connector that holds references to the input and output streams. The connector is comparable to a socket and it also provides a close operation.

When the transceiver on the client-side contacts an endpoint, it uniquely identifies a transceiver on the server-side, e.g. via IP address and ports. As a consequence, the server-side knows the plug-in at the lowest layer. Since there may be arbitrary plug-ins stacked on top of this layer, it is necessary to transmit the composition. To do this, the plug-in manager on the client-side transmits the session objects together (3) before continuing at higher layers. This enables the server-side to perform the connection establishment in parallel and it ensures that each plug-in can immediately contact its counterpart.

Once the session objects have been transmitted, the plug-in manager on the client-side initializes the remaining plug-ins. When a plug-in must be added the manager requests a connector from the plug-in. To do this, it passes the corresponding session object together with the connector of the transceiver (4). If the request succeeds, the modifier returns a new connector. This connector typically wraps the connector of the transceiver and provides new input and output streams using the underlying streams. The connector returned by the modifier may then be used to initialize the serializer and so on (5). As a last step, the manager passes the connector to the semantic plug-in as a response to the initial request (1).

At the server-side this process is mirrored. Since the server is passive, the connection establishment is started in response to the connection attempt of the client. Once the client-side has established a connection, the transceiver signals the new connection by passing a connector to the plug-in manager as depicted in step (1'). Using this connector, the plug-in manager receives the session objects from the client (2'). Thereafter, it initializes the modifier (3') and serializer (4'). As a last step, the plug-in manager forwards the connector to the semantic plug-in (5') which takes care of the actual data reception and interpretation. If this process fails at any point in time, the plug-ins may signal this through an exception and the plug-in manager takes care of closing the opened connection. The disconnection eventually causes a remote failure and the plug-in manager can perform the clean up.

3.5 Multi-Layer Plug-ins

Clearly, the plug-in negotiation and connection process imposes an overhead onto communication as it shifts the composition from development time to runtime. To reduce this overhead, the architecture enables plug-in developers to create multi-layer plug-ins. As depicted in Figure 6, multi-layer plug-ins can be used to increase efficiency or flexibility.

To increase the efficiency, a plug-in developer may decide to implement a complete stack that covers all aspects of communication within a single monolithic plug-in (1). In

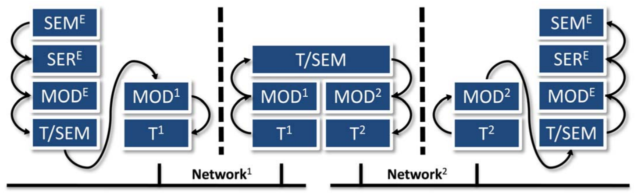


Figure 7: BASE Multi-hop Example

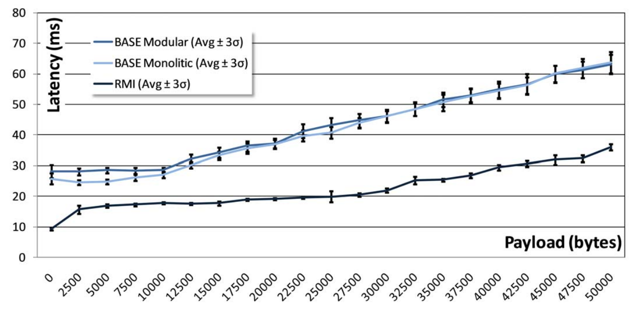


Figure 8: Latency Comparison (20 Remote Calls)

order to do this, the plug-in solely implements the interfaces of the semantic layer and it refrains from establishing connections. This essentially resembles the original architecture and thus, it is equally efficient but it suffers from the same deficiencies. Alternatively, a plug-in developer may also decide to implement a plug-in that covers all aspects of the connection. Thereby, a developer may decide to solely implement the highest entry layer (2) or in cases where the plug-in may be used at different layers, it can also support all of them (3). The same approach can be used for protocols that cover several aspects (4) and if they are optional, multiple entry-points can be supported (5).

To increase the flexibility, a developer may also decide to implement a multi-layer plug-in that traverses the layers multiple times. The primary use-case for such a plug-in is multi-hop communication. To support this, a plug-in may implement the transceiver and the semantic layer entry-points as shown in Figure 6 (6). As depicted in Figure 7, the final destination is used as target for the initial traversal of the layers which will lead to a stack that covers the semantic, serializer and modifier layers available on the source and the destination device (SEME, SERE, MODE). Since there is no single-hop connection between the devices, the transceiver of a multi-hop plug-in is the only viable selection. When the communication is about to be initialized, the transceiver of the multi-hop plug-in can request a connection to the next hop via the semantic entry-point. The resulting stack may include modifiers, e.g. for link-layer encryption, and a transceiver (MOD1, T1). After the connection establishment to the next hop, the plug-in manager transmits the stack configuration. The top of the stack is formed by the semantic of the multi-hop plug-in since it requested the connection. If the final destination has not been reached, the semantic can then request a stack to the next hop (MOD2, T2). Once this connection has been established, the plug-in can begin to forward data. On the destination device, the multi-hop plug-in can use the transceiver entry-point to forward the incoming connection to the manager. Once the end-to-end connection is running, the manager of the source device transmits the session objects of the initial traversal and the devices can communicate.

4. EVALUATION

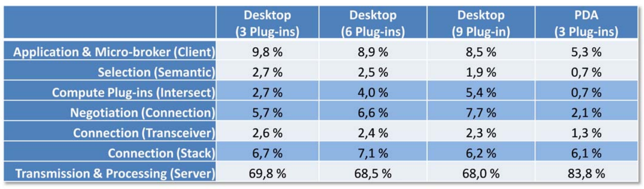
To determine whether the architecture achieves its goals, we implemented several plug-ins. To test the socket-style API for communication technologies, we implemented transceiver plug-ins for IP-, IR- and Bluetooth-based communication. Since the target operating systems (i.e. on Linux, Windows, Windows CE and Symbian) already provide connection-oriented abstractions (e.g. via WinSock), the development of a transceiver is straight-forward. Yet, due to the overhead of connection establishment on some devices, we had to develop a multiplexer that can be integrated into transceivers.

In addition, we have developed serializers and modifiers for object serialization, data compression using GZIP and encryption on top of the bouncy castle library. Furthermore, we have developed simple and complex multi-layer plug-ins, e.g. for multi-hop routing, and we have developed various semantics, e.g. for RMI and application level streaming. Thereby, we found that the plug-in interfaces are suitable for various technologies, protocols and abstractions. Given typical byte code sizes of 10-20 kilo bytes per plug-in, the reuse enabled by the redesigned architecture is a clear advantage for resource-poor devices. Moreover, we noticed that the programming effort for the plug-in interfaces is comparatively small, i.e. 80 lines for interfaces compared to 600-700 lines for a serializer.

For applications, access to the micro-broker is usually simplified via generated stubs. As a result, the details of the architecture can be hidden completely in many cases. To support fine-grained control, stubs can be generated as source code which enables customization when needed. By adding a few lines of code, an application can easily request the presence of certain plug-ins for a specific interaction. Thus, we argue that the architecture achieves its goals with respect to flexibility and simplicity.

To evaluate the impact of the architecture on efficiency, we discuss the results of several experiments. In the first, we compare the latency of executing remote method calls with varying payload sizes using different subjects. As subjects we use BASE with a monolithic plug-in (which closely resembles the original implementation), the modular version of BASE that uses individual plug-ins for the semantic, serializer and transceiver layers and an unmodified version of Java RMI [20]. As test environment we use 2 desktop PCs (Intel Core 2 Q6600, 2.4 GHz, 4GB RAM) running Windows Vista and Sun JRE1.6 that are connected via a switched Ethernet (1GBit). We measure the time to execute 20 identical remote calls and to determine outliers we repeat each measurement 1000 times. To avoid high variations, we deactivate the just-in-time compiler which increases the absolute values by a factor of 1.8.

Figure 8 shows the average with payload sizes ranging from 0 bytes to 50000 bytes. As indicated by the error bars, the standard deviation lies well below 3 %. When comparing the two versions of BASE with RMI, BASE introduces a notable overhead of approximately a factor of 3 in cases where no payload is transmitted and a factor of 2 in all other cases. We attribute this mainly to additionally created objects, the thread switches and buffers introduced by



	Desktop (3 Plug-ins)	Desktop (6 Plug-ins)	Desktop (9 Plug-in)	PDA (3 Plug-ins)
Application & Micro-broker (Client)	9,8 %	8,9 %	8,5 %	5,3 %
Selection (Semantic)	2,7 %	2,5 %	1,9 %	0,7 %
Compute Plug-ins (Intersect)	2,7 %	4,0 %	5,4 %	0,7 %
Negotiation (Connection)	5,7 %	6,6 %	7,7 %	2,1 %
Connection (Transceiver)	2,6 %	2,4 %	2,3 %	1,3 %
Connection (Stack)	6,7 %	7,1 %	6,2 %	6,1 %
Transmission & Processing (Server)	69,8 %	68,5 %	68,0 %	83,8 %

Figure 9: Latency Analysis (BASE Modular)

multiplexing the TCP connections in the transceiver plug-in and the differences in serialization - i.e. BASE implements object serialization to support J2ME CLDC. When comparing the two BASE versions, we observe 10-15 % overhead.

To determine the impact of an increased number of plug-ins and to determine the causes of the 10-15 % overhead of the architecture, we have performed a number of micro-measurements for remote method calls without payload (worst-case). The columns of the table shown in Figure 9 represents different setups. The rows show the percentage of time used for several stages of the call. These stages are (from top to bottom), the time spent above the plug-in layer at the client-side, the time for selecting the semantic plug-in, the time to compute the compatible plug-ins by computing the intersection set of the plug-in descriptions, the negotiation of the connection, the time required to establish a connection within the transceiver plug-in, the time to connect the plug-ins forming a stack on the client-side and the total transmission and processing time on the server.

The first three measurements (Desktop) are using the same setup as the previous experiment. To vary the setups, we increase the number of plug-ins on each system by adding one plug-in to the semantic, serializer and transceiver layers. As one expects, increasing the number of plug-ins increases the time to compute the set of usable plug-ins. Furthermore, it increases the time required during negotiation. However, it does not increase the time to connect a stack. Furthermore, one can observe that the overhead of the redesigned plug-in architecture is mainly an artefact of the more complex negotiation and the plug-in connection.

One may argue that an overhead of 10 % on desktops cannot be tolerated on resource-poor devices. To validate this, we have repeated the same experiment using two PDAs (XScale PXA270, 530MHz, 128MB RAM) connected via WLAN (802.11b with WPA encryption). When looking at the absolute time, we measure a latency of approximately 100ms for a remote call. However, as one can see from the 4th column (PDA), the percentage of time consumed by the modularization is well below 10 %. The reason for this are the latencies introduced by 802.11.

As a consequence, we can conclude that the overhead for the architecture typically stays around 10-15 % in cases where a stack needs to be computed and initialized. However, it is possible to reduce the overhead to 6-7 %, if a semantic plug-in caches the session objects. Finally, all overheads can be avoided, in cases where the communication stacks are reused to transmit multiple invocations. Thus, in cases where devices are interacting frequently with the same application requirements on communication, one can avoid repeated overheads without falling back to a monolithic design.

5. RELATED WORK

A multitude of middleware systems like CORBA [14] or Java RMI [20] have been developed for conventional systems to ease the task of developing distributed applications. Typically, these systems rely on a fixed set of existing communication protocols like TCP/IP or HTTP. They do not offer support for automated dynamic composition of stacks at runtime. The same is true for most middleware systems for resource-poor devices (e.g. [13], [17], [18]). These systems often provide only a restricted set of functionality, including no support for dynamic protocol reselection. As an exception, the Universally Interoperable Core (UIC) [16] can be dynamically extended with new protocols to interact with existing systems. However, the used stack is determined before the start of an interaction or even at installation time. BASE allows switching between different protocols for running interactions, too.

Dynamically reconfigurable middleware systems (e.g. [2], [5], [11], [15], [6]) can adapt their behavior at runtime to different application requirements, e.g. how marshalling is done, and environments. Still, like with UIC, this adaptation is usually not supported for already running interactions. Jini [19] allows using different protocol stacks for communicating with remote services. This is realized by integrating the protocol stack into a stub that is downloaded by the client. Jini provides no support for adapting the protocol stack used to access one given service. This is possible with our approach. The mundoCore middleware [1] allows creating custom protocol stacks by combining so-called protocol modules. However, mundoCore assigns each stack to a channel, which is then used to transfer multiple messages. BASE is able to automatically create an optimized protocol stack for each individual message, if required, without any further programmer interaction. Similar to the BASE invocation broker, the PIRATES middleware [10] is able to support different communication abstractions by means of a wrapper component. However, other aspects of the protocol stack such as serialization are statically built into the system and cannot be adapted. The DRAPS framework [12] supports the dynamic reconfiguration of protocol stacks during ongoing interactions by means of a state transfer mechanism. Although this is a powerful and efficient mechanism, it requires the sender and the receiver to reach a safe state to initiate the transfer which, in turn, requires connectivity to initiate the adaptation. However, in pervasive systems unpredictable changes to connectivity are often the root cause of protocol stack adaptation.

One.World [7] is a middleware system for pervasive computing environments which is based on Tuple Spaces. A central paradigm of One.World is to expose change, e.g. to allow application programmers to access environment-specific information and to act correspondingly. In contrast to the automatic adaptation capabilities of BASE, the responsibility for adapting to changes is completely laid off to the application programmer. Similar to BASE, the FAME2 [21] middleware relies on configurability to deal with the heterogeneity of pervasive computing. Yet, just like the original implementation of BASE, FAME2 uses monolithic plug-ins which induces associated limitations.

Vertical handovers, also known as media-independent handovers as specified in IEEE 802.21 [9] allow mobile users to roam between 802.11 networks and 3G cellular networks. To do so, the mobile device switches the used layer 2 communication technology automatically. BASE extends such layer 2 handovers to all layers.

6. CONCLUSION

Enabling the vision of pervasive computing requires appropriate middleware that can effectively and efficiently cope with heterogeneity. Due to the resource limitations of devices utilized in pervasive computing applications, configurability is a key requirement that extends to communication abstractions, protocols and technologies. In this paper, we presented a novel plug-in architecture that utilizes the runtime composition of plug-ins to support flexible communication. By configuring the generic middleware core with an appropriate set of plug-ins the middleware can be adapted to the target device and target applications. By automating the composition of plug-ins in a way that supports coarse- and fine-grained control, different applications can easily reuse the same plug-ins in vastly different communication stacks. The evaluation results show that the associated costs are acceptable and they indicate that many overheads can be avoided by a conscious plug-in design that uses caching and multiplexing. In cases where this is not possible, the plug-in architecture is flexible enough to support falling back to monolithic plug-ins at the cost of a higher development effort and a decreased flexibility.

7. ACKNOWLEDGMENTS

This work has been partially supported by CONET (Cooperating Objects Network of Excellence) and PECES (PErvasive Computing in Embedded Systems), both funded by the European Commission under FP7 with contract numbers FP7-2007-2-224053 and FP7-224342-ICT-2007-2 respectively.

8. REFERENCES

[1] E. Aitenbichler, J. Kangasharju, and M. Mühlhäuser. Mundocore: A light-weight infrastructure for pervasive computing. *Pervasive Mobile Computing*, 3(4):332–361, 2007.

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

[3] C. Becker, M. Handte, and G. Schiele. PCOM – a component system for pervasive computing. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom 04)*, Orlando, FL, USA, March 2004.

[4] C. Becker, G. Schiele, H. Gubbels, and K. Rothermel. Base – a micro-broker-based middleware for pervasive computing. In *Proceedings of the IEEE international conference on Pervasive Computing and Communications (PerCom)*, Mar. 2003.

[5] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski. The design and implementation of Open ORB version 2. *IEEE Distributed Systems Online Journal*, 2(6), 2001.

[6] E. P. de Freitas, M. A. Wehrmeister, C. E. Pereira, and T. Larsson. Reflective middleware for heterogeneous sensor networks. In *ARM '08: 7th workshop on Reflective and adaptive middleware*, pages 49–50, New York, NY, USA, 2008. ACM.

[7] R. Grimm. One.world: experiences with a pervasive computing architecture. *Pervasive Computing, IEEE*, 3(3):22 – 30, July-Sept. 2004.

[8] M. Handte, C. Becker, and G. Schiele. Experiences - extensibility and minimalism in BASE. In *Workshop on System Support for Ubiquitous Computing (UbiSys) at Ubicomp*, 2003.

[9] IEEE. 802.21 working group homepage, April 2009.

[10] D. Ingram. Reconfigurable middleware for high availability sensor systems. In *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, pages 1–11, New York, NY, USA, 2009. ACM.

[11] T. Ledoux. OpenCorba: A reflective open broker. In *Proceedings of the 2nd International Conference on Reflection (Reflection'99)*, pages 197–214, 1999.

[12] M. Niamanesh and R. Jalili. A dynamic-reconfigurable architecture for protocol stacks of networked systems. *Computer Software and Applications Conference, Annual International*, 1:609–612, 2007.

[13] Object Management Group. Minimum CORBA specification, revision 1.0, Aug. 2002.

[14] Object Management Group. The common object request broker: Architecture and specification, revision 3.0.3. online publication, March 2004. http://www.omg.org/.

[15] M. Roman, F. Kon, and R. H. Campbell. Design and implementation of runtime reflection in communication middleware: The dynamictao case. *Distributed Computing Systems, International Conference on*, 0:0122, 1999.

[16] M. Román, F. Kon, and R. H. Campbell. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online Journal, Special Issue on Reflective Middleware*, July 2001.

[17] M. Román, A. Singhai, D. Carvalho, C. Hess, and R. Campbell. Integrating PDAs into distributed systems: 2K and PalmORB. In *Proceedings of the International Symposium on Handheld and Ubiquitous Computing (HUC'99)*, Sept. 1999.

[18] D. C. Schmidt. Minimum TAO. online publication, June 2004. http://www.cs.wustl.edu/ schmidt/ACE_wrappers/TAO/docs/m

[19] Sun Microsystems. Jini technology core platform specification, version 1.2. online publication, December 2001.

[20] Sun Microsystems. Java remote method invocation specification, revision 1.8. online publication, 2002. http://java.sun.com/j2se/1.4/docs/guide/rmi/index.html.

[21] D. O. D. K. Wuest, B. Framework for middleware in ubiquitous computing systems. volume 4, pages 2262 –2267 Vol. 4, Sept. 2005.