Towards the Development of Ubiquitous Middleware Product Lines

Sven Apel, Klemens Böhm

Department of Computer Science
Otto-von-Guericke-University Magdeburg
email: {*apel* | *kboehm*}*@iti.cs.uni-magdeburg.de*

Abstract. Ubiquitous computing is a challenge for the design of middleware. The reasons are resource constraints, mobility, heterogeneity, etc., just to name a few. We argue that such middleware has to be tailored to the application scenario as well as to the target platform. Such tailor-made middleware has to be built from minimal fine-grained components, and the system structure must be highly configurable, as we will explain. We propose to use the well-known mixin layer approach to build the flexible lightweight middleware envisioned. We show that the thoughtful use of mixin layers is promising in this specific domain and allows to deal with issues such as device heterogeneity and resource constraints. To do so, we present the design and implementation of a middleware and three configurations derived from it. Our evaluation criteria are the number of supported features and the memory footprint. The middleware configurations derived perform well in these respects.

1 Introduction

Ubiquitous computing [26] is becoming reality. Everyone is connected everywhere and at any time, to consume and provide information. Computers become more and more transparent [25]. Middleware plays a key role to let this vision become true. It supports the application programmer who builds distributed applications and services, e.g., for electronic health care or intelligent buildings (more examples in [26, 25]). Such middleware must deal with the various characteristics of ubiquitous computing scenarios, e.g., resource-constrained devices, heterogeneity, mobility, bandwidth fluctuations, connection interruptions. Conventional middleware is not sufficient in these respects. It targets at static distributed systems with fixed hosts where resources are not tightly constrained.

This article attempts to validate the following hypothesis: the combination of software engineering and distributed computing principles will support the development of advanced ubiquitous applications, middleware services etc. well. To do so, we focus on the resource constraints of partly mobile ubiquitous devices, e.g. cell phones, wearable microchips, smart cards, autonomous robots, sensors, actuators, etc., and their heterogeneity in terms of hardware (e.g. processor, memory, communication media) and software (e.g. operating system, network protocol). We present a flexible middleware which one can easily port to

different hardware and software. It provides a device-independent interface to applications. As the design and implementation method, we propose the mixin layer approach. Mixin layers are known as one method for the implementation of *product-line architectures (PLA)* [22, 1]. Middleware for ubiquitous computing should benefit from the PLA concept as well, in terms of configurability, reusability and extensibility. By deploying the mixin concept, we want to verify if this is indeed the case. More generally, we wonder if implementation of general middleware concepts in mixin layers is feasible. The answer is not obvious because some concepts are known as crosscutting concerns, or are formulated in an abstract, 'high-level' manner. Further, are mixin layers a good implementation method for middleware that one can easily port to other devices? To address these issues, we have designed and implemented a middleware *PLA*[1] presented here. We then describe three middleware configurations which are tailored[2] to fit three specific ubiquitous application scenarios. We do so to show that our approach can lead to flexible lightweight middleware for ubiquitous computing. The derivation of these configurations consists of only a few steps. This is not straightforward – only the thoughtful use of mixins and the careful deployment to the middleware domain results in such ease. If one designs the layers carefully such that there are only few fine-grained device-specific layers, portability is much easier. Further, we investigate the relationship between the memory footprints of the configurations and the number of features integrated: We observe that few features result in a small footprint. As a result, configurability of middleware does not necessarily collide with small footprint. This is an important finding because other approaches cannot provide such a degree of configurability in combination with small footprint, as Section 6 will explain. Finally, we say why these results can be generalized to other middleware.

This article is structured as follows: Section 2 introduces an ubiquitous computing application scenario and points to problems regarding middleware and applications. Section 3 reviews the software engineering methods deployed here. Section 4 presents our middleware, built according to the mixin layer approach. We then discuss implementation results and experiences concerning the configuration. Section 6 reviews related work. Finally, we conclude.

2 A Ubiquitous Computing Application Scenario

This section sketches an application scenario for ubiquitous computing. Based on this, we list challenges at the middleware and the application level. We point to the weaknesses of conventional middleware approaches.

2.1 Application Scenario

With ubiquitous computing, computers become an even more integral part of everyday life. They act behind the scenes, transparently for humans. The scenario

[1] In the remaining article, 'middleware' and 'middleware PLA' are synonyms.

[2] We refer to tailoring as configuration process with special focus on memory footprint. To do so, unneeded functionality is removed consequently.

presented, in parts borrowed from [25], includes conventional aspects as well as more visionary ones. Starting point is a room with many common mobile and ubiquitous devices, e.g., PDA and Smartphones. They are general-purpose devices which include various communication media, e.g., WLAN, Bluetooth, IR, etc. If a person enters the room, the PDA can contact the embedded devices available. For instance, the PDA can communicate with the light switch to raise or dim the light. To facilitate this, the dimmer offers an appropriate service interface. A primitive dimmer only provides a basic service to dim or light up. A more complex dimmer can provide additional information about the minimum and maximum dim level or provide a timeout mechanism to adjust the light automatically. A 'more ubiquitous' scenario is that the light dimmer adjusts itself by communicating with the PDA behind the scenes. A person enters the room, and the light adjusts itself, using personal information from the PDA. Other devices in the room act more autonomously, e.g., a climate-control unit which adjusts the air condition to the current climate, to the current time of day and the current season as well as to the presence of a person. Further, think of a digital paper scrap which people use to take notes. Notes are then stored on a central notes server. A more common device is a home-entertainment system, including a music box, a dvd recorder and a TV set. It apparently provides a lot of controllable functionality and interacts with itself and the PDA extensively. For instance, it provides information on the TV program. This information can control the programming of the dvd recorder. In a more ubiquitous setting, the dvd recorder reacts to program changes or records telecasts which match a profile autonomously. The next step is that the dvd recorder in cooperation with the TV set learns the customs of persons and generates personal profiles itself.

2.2 Problems Occurring

In the scenario introduced, certain problems occur, which we describe next. Common middleware cannot deal with these problems, as we will explain.

Ubiquitous computing middleware must run on the various devices. Frequently, devices are embedded systems. They are developed for a special purpose, e.g., to control the light, and have a low resource consumption. Cost-effective thinking requires this, in particular if the number of these devices is huge. Next to these embedded special-purpose devices, general-purpose devices (PDA, Desktop PC, Server) are part of ubiquitous-computing scenarios. These devices are not resource-constrained and provide much more functionality. In our scenario, the PDA communicates with other devices, displays information (e.g., air-condition level) and processes it (e.g., television-program based programming of the dvd recorder). The spectrum of resources consumed is extremely broad, as well as the one of functionality provided.

Another challenge is to overcome the heterogeneity of devices. They use different hardware and software. The middleware must bridge them and must provide a well-defined device-independent interface to the application programmer. Hence, the middleware consists of device-specific and device-independent parts.

Naturally, the device-independent part must be as large as possible (in relative terms) to maximize reusability.

Our middleware is supposed to hide these specifics, in order to support the development of ubiquitous applications. Conventional middleware approaches, e.g., CORBA, DCOM, Java-RMI, are not suitable for our scenario. They are too heavyweight and cannot be customized to application requirements. It would be quite impossible to port them to other devices or platforms and to get them to work in resource-constrained environments. The monolithic system structure prevents the reuse of logical device-independent functionality. However, research effort has tried to improve standard CORBA to fit ubiquitous computing. It has been shown that refactorization of CORBA implementations yields higher configurability [27]. However, our expectation in the long run is that customizability of carefully designed middleware product-lines is even higher. This is why we think that the issue merits attention. Finally, dynamic adaptation [20, 12] does not solve the problem in our specific context either, as Section 6 will explain.

All this motivates the design of a ubiquitous-computing middleware with the following features:

- minimal memory footprint and lightweight implementation, to save resources,
- run on heterogenous hardware and software,
- provide uniform device-independent application interface,
- customizability, reusability, and extensibility.

3 Relevant Software Engineering Issues

This section presents our solution to the problems discussed in Section 2. It uses the mixin layer approach. This is because this approach is known to facilitate configurable and reusable software, e.g., product-line architectures [22]. To ease understanding, we provide some background information on this software engineering method. The so-called *collaboration-based design* is a feasible design method to serve as a basis for mixin layer implementations. We briefly review it here as well. Finally, we outline the expected benefits of these approaches for ubiquitous computing, before looking at our realization in the next section.

3.1 Collaboration-Based Design

Parnas [19] introduced collaboration-based design first. The idea is to build software incrementally, using minimal building blocks and starting from a minimal base. Exchanging, adding and removing such building blocks, also called *layers*, yields reusability, extensibility, and customizability. Batory et al. have mapped this concept to the object-oriented world [1, 22]. They observe that a new *software feature* often extends or modifies numerous existing classes. Based on this observation, they perceive features as *collaborations of class/object fragments*, also referred to as *roles*. Figure 1 collaborations. Classes are arranged

vertically ($c_1 - c_3$). Collaborations are arranged horizontally and span several classes ($f_1 - f_3$). Several features of a software system result in a stack of collaborations. In our context, examples of features are 'remote procedure calls' or 'remote object invocation'. Collaborations with the same interfaces are easily exchangeable. They are an instance of large-scale components [1]. A collaboration of objects implements a feature and is part of a layered stack.[3]

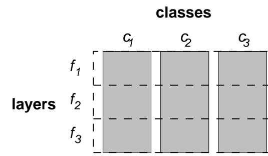


Fig. 1. Stack of collaborations

3.2 Mixin Layers

The mixin layer approach allows to implement collaboration-based designs. The mixin layer approach is based on the *GenVoca* component model [1] to support large-scale components, easy exchangeability, and syntactic consistency checking. These characteristics allow for the development of configurable and reusable software. Different languages can be used to implement mixin layers: *C++* [23], *AHEAD Tool Suite* [2], *Java Layers* [7], *Sather* [17]. In the code snippets that follow we use the C++ notation.

Mixins are types whose super-types are specified parametrically [5]. Mixins facilitate the same sub-type specialization to be applied to different (super-)-types. In other words, they allow the specialization of multiple classes with a single reusable class. For example, think of the three unrelated classes `Buffer`, `Message`, `Printer`. Suppose that one wants to add a locking feature that restricts access to these objects. With conventional object oriented approaches, one must add a different sub-type to each class, e.g., `LockableBuffer`, `LockableMessage`, `LockablePrinter`. Each of these types adds the methods `lock()` and `unlock()`. With mixins in turn, a single mixin `Lockable` extends all of those super-classes (see Figure 2). Methods `lock()` and `unlock()` are defined only once. Instantiation of mixins generates new class hierarchies. For instance, the instantiation `Lockable<Buffer> lb;` generates the class hierarchy depicted in Figure 3.

```
1  template <class BaseType>
2  class Lockable : public BaseType {
3      bool m_locked;
4      void lock() {m_locked = true;}
5      void unlock() {m_locked = false;} }
```

Fig. 2. A simple mixin class for synchronization support.

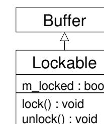


Fig. 3. A lockable buffer

Mixin layers are mixins containing nested types which can be mixins themselves [22]. Mixin layers are used to coordinate changes and extensions to classes

[3] We use the terms feature or layer as synonym for collaboration.

that collaborate. The mixin layer approach allows to add a new feature/layer in form of a set of sub-classes to a software system using one implementation unit. A single mixin layer is able to implement a feature that crosscuts multiple classes. Mixin layers are equivalent to collaborations, whereas nested mixins are equivalent to object roles. Consider the following example: A program library provides a buffer to store data elements and an iterator to traverse the data elements. A possible refinement is to store and manage arrays of data elements. Applying this new feature requires modifications of both buffer and iterator. Figure 4 depicts the two mixin-based feature implementations `BufferLayer` (Lines 1–4) and `ArrayBufferLayer` (Lines 5–8). The BufferLayer simply consists of a `Buffer` class and an `Iterator` class. `ArrayBufferLayer` is a mixin layer, which expects a template parameter (`BaseType`). Line 10 contains the instantiation of `ArrayBufferLayer` using `BufferLayer` as super-type. This instantiation connects both layers and their corresponding nested types (`Buffer`, `Iterator`) using inheritance (Lines 5–7). It refines the basic buffer abstraction with the array feature. This feature crosscuts the classes `Buffer` and `Iterator`. The 'Array' feature is encapsulated in a single implementation unit. This eases the composition of mixin layers and therefore the configuration of the target software. To see this, think of ten further buffer features, e.g. locking, synchronization, complex data types. Composing the buffer implementation using these mixin layers requires only *one* instruction, e.g., `Lockable<Sync<...<ArrayBufferLayer<BufferLayer>>...>>buf;`. This example illustrates the ease of configuration of mixin layer-based implementations. (more examples in [1, 22, 2])

```
1  class BufferLayer {
2      class Buffer {};    // store simple data types
3      class Iterator {}; // traverses element-wise
4  };
5  template <class BaseType> class ArrayBufferLayer : public BaseType  {
6      class Buffer : BaseType::Buffer {/* stores arrays of elements */}
7      class Iterator : BaseType::Iterator {/* iterates array-wise */}
8  }
9  ...
10 ArrayBuffer<Buffer> abuf;
```

Fig. 4. A base and a mixin layer: the buffer abstraction and array management feature

3.3 Benefits for Ubiquitous Computing

Mixin layers offer several benefits for the development of middleware for ubiquitous computing. As mentioned before, this article focuses on resource constraints and heterogeneity.

Resource constraints. Mixin layers offer modularity and flexibility and thus seem to be ideal candidates for the design and implementation of tailored software for ubiquitous computing. To accomplish this, the system components (the

mixin layers) must be fine-grained. Middleware must be customized to the specific hardware and to the requirements of the application, e.g., performance.

Heterogeneity. Another issue is the heterogeneity of the devices involved. The objective of our middleware (as well as of other middleware) is to bridge this heterogeneity and to provide components which can work with different hardware, operating systems and network protocols. How can mixin layers help in this respect? An interesting feature of mixin layers in the middleware domain, for heterogeneous environments, is decomposition of middleware functionality into device-specific and device-independent components. The goal is to minimize the number of device-specific components. This leads to middleware which is easier to port to new platforms. Summing up, the thoughtful use of mixin layers seems to be promising to deal with heterogeneity in ubiquitous computing.

In summary, mixin layers might help to address the following issues in ubiquitous computing where common middleware solutions are not sufficient:

- resource constraints (step-wise refinements, minimal exchangeable layers)
- heterogeneity (composition of device-specific and device-independent layers)
- lack of customizability (mixin layer as large-scale components, separation of crosscutting concerns)

Admittedly, the mixin layer approach also has some disadvantages: It is not always practical to implement a feature as a single mixin layer. The implementation units of such features are often spread over several other feature implementations. Mixin layers only allow to refine related classes, namely those included in the stack of basic layers. Related approaches like *aspect-oriented programming (AOP)* [13] and *multi-dimensional separation of concerns (MDSC)* [18] can refine unrelated classes as well, using one implementation unit. AOP and MDSC support refinements on statement and expression level as well as a regular-expression based mechanism to specify code positions where refinements are applied (*join points*). On the other hand, they lack a component model, e.g., imported and exported interfaces or symmetric components. Hence, it is difficult to build configurable product-lines. The mixin approach in turn is sufficient for building the base functionality of our middleware, as we will explain in Section 4. In general the approaches mentioned are equivalent with regard to modularization of crosscutting concerns and customizability [2]. In the long run, we think that only a combination of these will lead to success, if more complex functionality is added, e.g., fault-tolerance or security. Each feature will be implemented using the appropriate method. However, these issues are beyond the scope of this paper. This paper in turn investigates the benefits and limits of mixin layers to build middleware product-lines.

4 Middleware Design

This section presents the design and implementation of a flexible lightweight middleware for ubiquitous computing, based on collaborations and mixin layers. To assess the benefits of these methods, the implementation of the following

functionality should suffice: The middleware provides standard *remote object invocation (ROI).* Well-known subconcepts of ROI, e.g., marshaling, are part of the implementation, but are not discussed here. Moreover, we leave ubiquitous-computing specific features, e.g., server-initiated computation. Their implementation would not provide significant further insight (we argue). Our concern is the deployment of collaborations and mixin layers for ubiquitous middleware. Arguably, implementation of middleware functionality in mixin layers is not obvious. In addition, a solution must cope with devices that are resource-constrained and with heterogeneous environments. Our design described next, i.e., the specific arrangement of collaborations or the specific choice of the roles, is only one possible solution (but nevertheless appropriate, as we will show).

The result of collaboration-based design and of the mixin layer approach is a set of components. They can be composed to various middleware platforms. Subsequently, we refer to these platforms as *configurations*. When designing our middleware, we have found it natural to distinguish between components with client-side functionality and those with server-side functionality. The feature of managing and registering remote objects is server-side, but only the client sends requests to the server, to give some examples. Moreover, we identified some features used by client and server. This motivates the following terminology: *general layers*, *client layers* and *server layers*.

Several figures depict the collaboration/layer stack and the roles included. The rounded boxes represent roles (the dashed boxes mark derived roles) and the grey boxes in the background represent the collaborations. We organized the stack of layers in bottom-up order. We use UML-like arrows to represent relations between object roles (inheritance, composition, etc.). Explaining all roles in detail is beyond the scope of this article and is actually not necessary for understanding. We focus on the overall structure, and we say how to design collaborations and to implement mixin layers for ubiquitous middleware.

4.1 General Middleware Layers

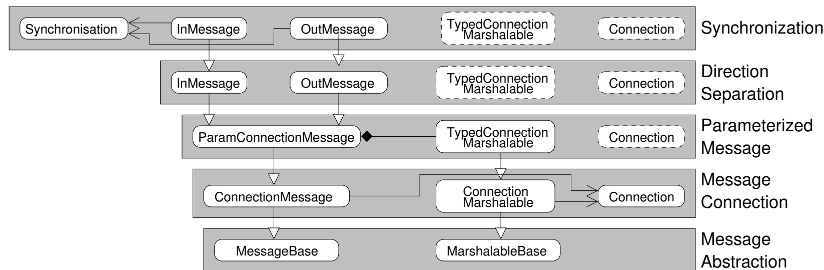


Fig. 5. The stack of general middleware layers

The general layers provide basic functionality for both client side and server side. Figure 5 depicts the stack of general layers. The abstraction of basic messages form the bottom layer of the stack. These messages are transferred between

client and server. A marshaling mechanism serializes messages; connections are based on sockets. Messages may have parameters that are typed. The parameters are used in higher layers as function arguments or to identify operations and instances on clients and servers. Our approach allows to decide at compile-time which data types are supported. Avoiding types that are not needed reduces the memory consumption. Other variation points, again well known, are the connection type (UDP or TCP), the direction of communication (unidirectional or bidirectional) and the synchronization strategy (synchronous or asynchronous). The variation points as well as the different data types supported are implemented as different layer variants to enhance configurability. For instance, two different layers exist for the synchronization feature (synchronous and asynchronous). At configuration time, the programmer has to choose one.

4.2 Client-Side Layers

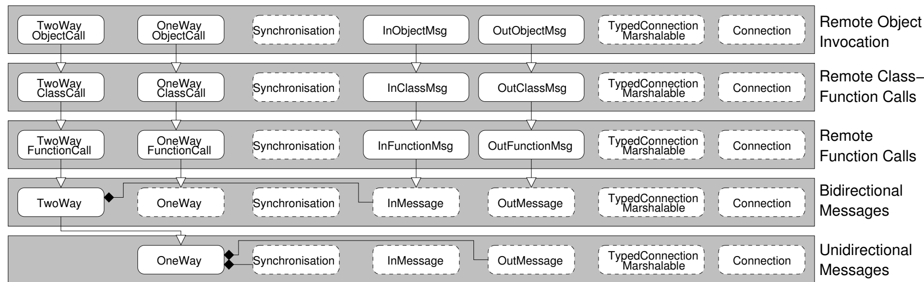


Fig. 6. The stack of client-side layers

Based on the general layers, the client layers facilitate uni- and bidirectional messaging (see Figure 6). We use these messaging functions to implement remote function and class-function calls and remote object invocation. We use parameterized messages to deliver the identifications of functions, classes, objects and their arguments. Response messages deliver results. Next to function calls, we provide operations for creating and deleting objects.

4.3 Server-Side Layers

Figure 7 shows the server-side layers, but does not show all roles, due to space limitations. The server layers are shown as light grey boxes, the client layers as dark grey boxes. Client and server layers that correspond to each other are often required in combination, e.g., remote function calls and remote functions. This interleaving does not mean that a server implementation always requires the client layers and vice versa. If a configuration does not need certain client layers, one has to remove them during the configuration process (cf. Section 5). Consider the light dimmer from Section 2. It only needs to obtain incoming function calls, but does not need to issue remote function calls.

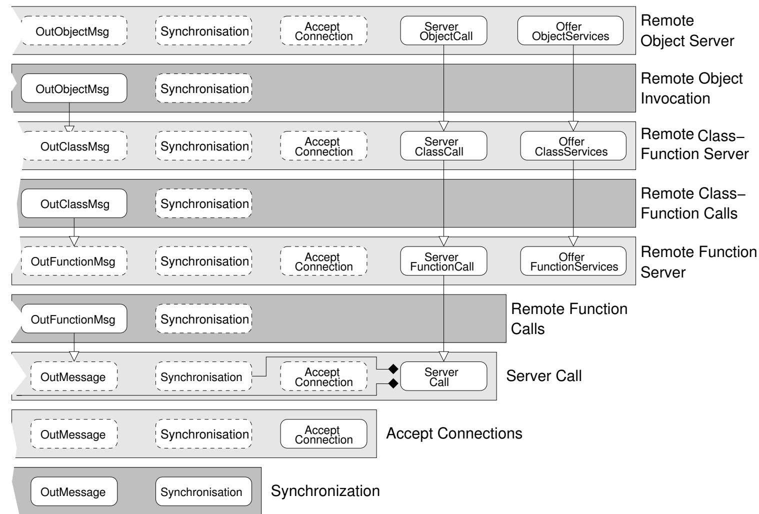


Fig. 7. The stack of server-side layers

The basic server layers listen for client requests and accept connections. We have implemented a single-threaded and a multi-threaded variant. Our middleware deserializes incoming messages. Depending on the connection type chosen, the incoming messages are transferred using byte streams or datagrams. Based on these mechanisms, the server side provides a remote function server, a remote class-function server, and a remote object server. The programmer can use several functions for registering and managing remote functions, classes and objects. A client can specify the desired function or object as well as the desired operation (create, delete, invocate) using parameterized messages.

4.4 Implementation

In order to verify our hypotheses, we have implemented the middleware design presented so far in full (in C++). To do so, we have used the template mechanism, nested classes and parameter-based inheritance, as described in Section 3. To save implementation work, we use the *gSoap* communication library [10]. While designing and implementing the system, we have kept in mind that the communication library should be easily exchangeable, e.g., with a lightweight binary protocol. We have used the low-level functions only to (un-)marshal and to send or receive SOAP [4] messages. We have not used remote-function-call mechanisms or other high-level functions. gSoap is the only device-dependent part of our middleware. Because space is limited, we cannot discuss all issues at code level. Instead we refer to Figure 8. It depicts the interface of one mixin layer, the roi layer in C++. The layer has four nested classes (Lines 2–3) which represent the corresponding roles (cf. Figure 6). Each nested class inherits from the corresponding classes of the base layer, represented by `BaseLayer`. Beyond the

```
1  template <class BaseLayer> class ROILayer : public BaseLayer {
2      class OutObjectMessage : BaseLayer::OutClassMessage {};
3      class InObjectMessage : BaseLayer::InClassMessage {};
4      class OneWayCall : BaseLayer::OneWayCall {};
5      class TwoWayCall : BaseLayer::TwoWayCall {}; };
```

Fig. 8. The Remote Object Invocation Mixin Layer

short example the implementation examples from Section 3 and the discussion of design issues there should shed light on our implementation.

5 Results

This section discusses experiences from the implementation, together with three configurations: a sensor-actuator-system, a web service/client and a roi client/-service. They are useful for the scenario described in Section 2.

5.1 Configuration

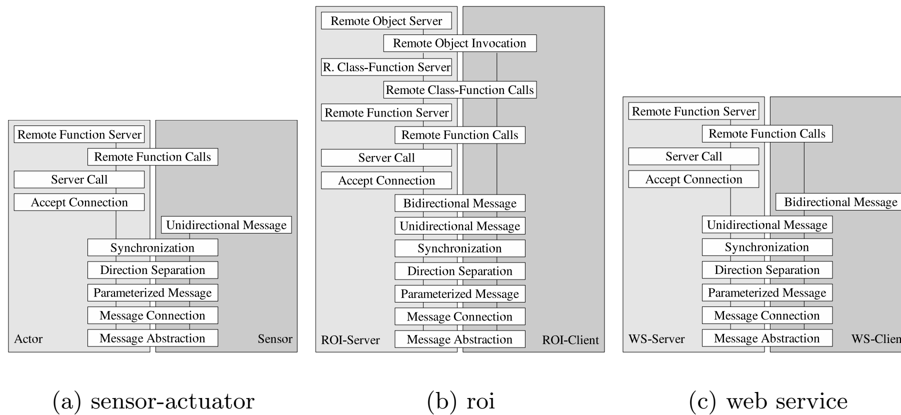
Instantiation (combination) of the mixin layers configures new middleware platforms (see Section 3). A *GenVoca* grammar describes the possible configurations [1] (not shown here for lack of space). Using this grammar, we have calculated the number of configurations possible by adding the numbers of all combinations of layers of our middleware permitted: $192 * (2^n - 1)$ different server configurations, where n is the number of data types supported, and $96 * (2^n - 1)$ client configurations. As a result, the degree of configurability is high. This is required for tailoring the middleware to work in ubiquitous computing scenarios.

To convey the ease of the configuration procedure and the flexibility of the implementation, we now describe the three configurations we have derived.

Sensor-Actuator Middleware. A sensor-actuator middleware is useful for ubiquitous devices like the our light dimmer, which only needs a small subset of the functionality. For communication between sensors and actuators, we chose asynchronous unidirectional remote procedure calls. In our scenario, a light sensor only needs the client features. We add the server-side features only to the actuators (a light dimmer), which receive messages. Figure 9(a) depicts the features chosen. In our example application, we have used the sensor to send a measurement to the actuator. Both devices display status information.

Remote-Object-Invocation Middleware. Our configuration of a remote-object-invocation (*roi*) middleware consists of nearly all layers implemented. It is used for ubiquitous devices which provide a rich set of functionality and provide many services. In our scenario, the home entertainment system runs a fully functional object server. Next to remote object invocations, remote function calls and remote class-function calls are also available. We have chosen synchronous communication. Figure 9(b) depicts the layers of the client and of the server. To complete the proof-of-concept implementation, we have implemented a simple service on top of the roi-middleware.

Web-Service Middleware. The web-service (*ws*) middleware supports the implementation of web services. This configuration is useful to access ubiquitous devices from the internet using *SOAP* [4]. Similarly, ubiquitous information systems like a digital newspaper can collect information from the Web and display it. The web-service middleware provides the following functionality: SOAP-conformant remote function calls as well as synchronous and asynchronous communication. When using gSoap, creating SOAP messages that conform to the standard is easy. Our web-service middleware is useful to implement all types of web-services and corresponding clients. Our example server can receive SOAP messages and can reply to every common SOAP client with the same interface. The analogous is true for our client as well. It can connect to any compatible web service. Figure 9(c) depicts the layers of the client and the server.



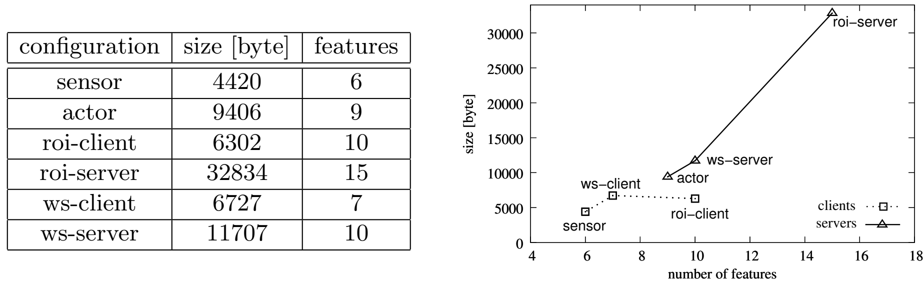
(a) sensor-actuator (b) roi (c) web service

Fig. 9. Three different middleware configurations

5.2 Discussion

Section 5.1 has shown that a broad range of configurations is possible. Moreover, Sections 3 and 5.1 have discussed the easy composition of layers to create a configuration. Some configurations may differ only in a few features. But the three examples show the broadness of the application scenarios supported. Let us now have a closer look at the resulting configurations. Figure 10 shows the memory footprint and the number of features supported, with a distinction between client-side and server-side. The memory footprint is the size of the binary code. We have obtained it using the linux `size` command. We have left aside the code of the underlying communication protocol library. The ws client is bigger than the roi client because it has to process and transfer additional web-service-specific information like namespaces, etc. The binary code size of clients ranges from 4423 to 6631 bytes and the one of the servers from 9310 to 32738 bytes. The server-side results show that the memory footprint of a minimal system

configuration (the actuator) is only 28% of the one of the maximal system configuration (the roi server). At the client side, the minimal configuration is about 65%. The binary code size of the web service lies between them. This is because it has more features than the sensor-actuator middleware and much less features than the roi service. As a result, configuring middleware that does not waste



configuration	size [byte]	features
sensor	4420	6
actor	9406	9
roi-client	6302	10
roi-server	32834	15
ws-client	6727	7
ws-server	11707	10

Fig. 10. Memory footprint and number of features of three configurations

resources is easy, using the mixin layer approach. As our implementation has shown, decomposition of middleware functionality into fine-grained components is possible (cf. Figure 9). With less features, the code size and the amount of data and consequently the binary code size decreases significantly (see Figure 10). So it is in the hand of the application programmer to tailor the middleware and fit it to the application requirements and target platform. Configurability and tailoring make it possible to build middleware for embedded ubiquitous devices.

To deal with hardware and software heterogeneity, we differentiate between device-specific and device-independent layers. Only the layers which communicate directly with the hardware or the underlying software (operating system, protocol stack) are device-specific.

The reader should note that a performance analysis is not meaningful in the current context, for various reasons: (1) We have used a SOAP-based communication library. The overhead for parsing and generating the XML/SOAP messages would falsify performance numbers. (2) A direct comparesion to other middleware solutions, e.g. [11, 24, 16], is not meaningful, because the set of features implemented (communication protocol, marshaling strategy, data types supported, etc.) is different. – The design of mixin layers that results in configurations both with small footprint and good performance is an interesting issue, but is beyond the scope of this article (obviously, the problem is more difficult).

Finally, our results generalize to other middleware as well, not only in the ubiquitous computing domain. For example, one can build middleware for mobile computing using more large-scale components, to reduce the maintenance overhead. Reflective architectures like [20] or [12] could be implemented using mixin layers and could work together with current base level components. However, reconciliation of the objectives performance, small memory footprint, as well as configurability, reusability and extensibility is an open issue.

6 Related Work

Conventional middleware technology (e.g. CORBA, SOAP, Java-RMI) hides the internal communication. It is designed primarily for fixed hosts with adequate resources and a static network structure. It does not run in non-conventional application scenarios, e.g., embedded systems and ubiquitous computing. Middleware technologies have emerged to meet the requirements of these scenarios, e.g., real-time constraints, reliability, as well as environment-specific issues, e.g., resource constraints, bandwidth fluctuation, connection interrupts, dynamic changes of network topology. However, some research has enhanced CORBA-based middleware to become flexible, customizable and lightweight: *OpenCorba* [15], *OpenORB* [3] and *dynamicTAO* [14] extend CORBA by a reflective architecture. These systems *reify* important characteristics of the behavior and of the structure of the middleware, such as scheduling strategy and resource management. The application can access and modify this information using a meta-interface. This allows to customize the middleware at runtime. A similar reflective CORBA-independent approach is *CARISMA* [6]. Its focus is on context-awareness and on policy conflict resolution. We for our part have focused on customizability at compile-time. By doing so, we do not need a reflective architecture which would consume a significant amount of resources (we argue). Furthermore, reflection is simply not needed in all ubiquitous devices and services. (Think of the primitive light dimmer.) On the other hand, mixin-based middleware may serve as a basis for a runtime-adaptable implementation, which combines the advantages of mixin layers and reflection.

UIC [20] and *ReMMoC* [12] are two examples of middleware with a focus on device heterogeneity. Both assume that different devices use different middleware technologies, e.g., *SOAP*, *CORBA*, *Java RMI*, and provide mechanisms to deal with this heterogeneity. UIC is based on dynamicTAO. It implements a reflective architecture and a minimal core of functionality. If the reflective architecture detects the presence of a remote device, it loads the adequate middleware component. ReMMoC uses a similar approach. While this is a significant contribution for conventional application scenarios, it seems to us that the runtime overhead of reflection may not always be acceptable in ubiquitous devices. Further on, reflection may not be required in some ubiquitous application scenarios.

TAO [21] is another prominent approach to achieve customizability, based on design patterns. We believe that modern component models have a stronger focus on modularization and configurability than design patterns. Moreover, the mixin layer approach supports the development of PLA well [2]. It is the high degree of configurability and tailoring that makes PLA a suitable candidate for the development of ubiquitous computing middleware.

Zhang and Jacobsen have shown how to improve customizability and flexibility of middleware by refactorization [27]. They utilized AOP to remodularize orthogonal, entangled middleware features, e.g., the dynamic programming model or portable interceptors. Colyer and Clement argue that AOP can help to cope with the rising complexity of middleware [8]. They have refactored several middleware crosscutting concerns successfully. They have argued that AOP can

scale to size of commercial middleware projects. Our approach towards building a product-line does not focus on refactoring. Rather a carefully planned and designed middleware product-line makes refactoring unnecessary.

[24, 11, 16, 9] focus on middleware for embedded and real-time systems. Their work addresses performance and resource consumption issues. Tailoring and customization of middleware using modern software engineering methods are not discussed. But such methods are key to overcome device heterogeneity, resource constraints and lack of customization functionality.

7 Conclusion and Further Research

Software engineering methods advance the design and implementation of middleware for ubiquitous computing. We have proposed the use of collaboration-based design and mixin layers to build lightweight flexible middleware for this domain, to provide a device-independent interface to applications. We have implemented a set of fine-grained basic components. We have generated three middleware configurations, tailored to specific application requirements. The configuration phase consists of a few steps only. A GenVoca grammar describes the combinations permitted. As a result, tailoring of the middleware has been successful in terms of memory footprint. Reusability and configurability of a mixin-based implementation helps to deal with device heterogeneity.

As future work, we want to integrate new features like security, persistence or fault-tolerance. Here, other software engineering methods like aspect-oriented programming, multi-dimensional separation of concerns or feature-oriented domain modeling look promising. Another issue is the performance of mixin-based middleware, in combination with reusability, customization and extensibility.

Acknowledgments. We thank Helge Sichting for much help with this study. We acknowledge the generous support of METOP GmbH, Magdeburg.

References

1. D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4), 1992.
2. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. In *Proc. of the 25th Int. Conference on Software Engineering*, 2003.
3. G. S. Blair et al. Reflection, Self-awareness and Self-healing in OpenORB. In *Proc. of the 1st Workshop on Self-healing Systems*, 2002.
4. B. Box et al. Simple Object Access Protocol 1.1. Technical report, W_3C, 2000. http://www.w3c.org/TR/SOAP.
5. G. Bracha and W. Cook. Mixin-Based Inheritance. In *Proc. of ECOOP / OOPSLA*, 1990.
6. L. Capra, W. Emmerich, and C. Mascolo. CARISMA: Context-Aware Reflective Middleware System for Mobile Applications. *IEEE Transactions on Software Engineering*, 29(10), 2003.

7. R. Cardone, D. Batory, and C. Lin. Java Layers: Extending Java to Support Component-Based Programming. Technical Report CS-TR-00-11, Computer Sciences Department, University of Texas, 2000.
8. A. Colyer and A. Clement. Large-scale AOSD for middleware. In *Proc. of the 3rd Int. Conference on Aspect-Oriented Software Development*, 2004.
9. E. Eide et al. Dynamic CPU Management for Real-Time, Middleware-Based Systems. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004)*, Toronto, Canada, 2004.
10. R. A. Engelen and K. A. Gallivan. The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks. In *Proc. of IEEE CCGrid Conference*, 2002. http://www.cs.fsu.edu/˜engelen/soap.html.
11. C. Gill et al. ORB Middleware Evolution for Networked Embedded Systems. In *Proc. of the 8th Int. Workshop on Object Oriented Real-time Dependable Systems (WORDS'03)*, Guadalajara, Mexico, 2003.
12. P. Grace, G. S. Blair, and S. Samuel. ReMMoC: A Reflective Middleware to Support Mobile Client Interoperability. In *Proc. of the Int. Symposium on Distributed Objects and Applications (DOA 2003)*, Catania, Italy, 2003.
13. G. Kiczales et al. Aspect-Oriented Programming. In *Proc. of ECOOP*, Berlin, Heidelberg, and New York, 1997.
14. F. Kon et al. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proc. of Middleware'2000*, New York, 2000.
15. T. Ledoux. OpenCorba: A Reflective Open Broker. In *Proc. of the 2nd Int. Conference on Meta-Level Architectures and Reflection*, 1999.
16. A. D. McKinnon et al. A Configurable Middleware Framework with Multiple Quality of Service Properties for Small Embedded Systems. In *Proc. of 2nd IEEE Int. Symposium on Network Computing and Applications*, Cambridge, MA, 2003.
17. S. M. Omohundro. The Sather Programming Language. *Dr. Dobb's Journal*, 18(11), 1993.
18. H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In *Proc. of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2000.
19. D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transaction on Software Engineering*, SE-5(2), 1979.
20. M. Román, F. Kon, and R. Campbell. Reflective Middleware: From Your Desk to Your Hand. *IEEE Distributed Systems Online (Special Issue on Reflective Middleware)*, 2(5), 2001.
21. D. C. Schmidt et al. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), 2002.
22. Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collabroation-Based Designs. *ACM Transactions on Software Engineering and Methodology*, 11(2), 2002.
23. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.
24. V. Subramonian et al. Middleware Specialization for Memory-Constrained Networked Embedded Systems. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004)*, Toronto, Canada, 2004.
25. M. Weiser. The Computer for the 21st Century. *Scientific American*, Sep. 1991.
26. M. Weiser. Hot Topics: Ubiquitous Computing. *IEEE Computer*, 26(10), 1993.
27. C. Zhang and H.-A. Jacobsen. Refactoring Middleware with Aspects. *IEEE Transations on Parallel and Distributed Systems*, 14(11), 2003.