R E F L E C T I V E   M I D D L E W A R E

Vol. 2, No. 5, 2001

Reflective Middleware: From Your Desk to Your Hand

Manuel Román, Ubicore LLC
Fabio Kon, University of São Paulo, Brazil
Roy H. Campbell, University of Illinois at Urbana-Champaign

*C*ommunication middleware simplifies the construction of component-based distributed applications.[1,2] However, middleware construction's inflexibility imposes limitations and presents major concerns.[3-7] While applications can detect changes in their execution environment, they cannot customize the underlying middleware to better accommodate these changes.

To solve this problem, recent research in reflective middleware[8] uses techniques derived from previous work in computational reflection to add flexibility to middleware. Reflective middleware exploits Gregory Kiczales' *meta-object protocol*, combining the ideas of computational reflection and object orientation.[9] His model distinguishes between base-level objects (concerning the systems' functional aspects) and metalevel objects (concerning aspects such as policies, mechanisms, or strategies). The base level of reflective middleware addresses the application program's functionality, while the metalevel designates collections of components that form the internal architecture of the middleware platform. Reflection allows the inspection and modification of these objects, enabling changes in the middleware's behavior.

As computer networks become pervasive and portable, and as embedded and handheld devices become more common, mobile users are creating a demand for a plethora of ubiquitous services. Because ubiquitous computing is characterized by constant change and a large degree of dynamism, users must interact with heterogeneous systems using different devices at different locations over different networks with different quality-of-service requirements. They must also share the systems with many other users. In practical terms, it is difficult to devise a fixed set of policies and mechanisms for communication, security, and resource allocation that accommodates such diversity, change, and scale.

Reflective middleware offers flexibility in this dynamic environment. Here, we discuss two reflective ORBs-dynamicTAO and the Universally Interoperable Core-that provide a solid base for supporting safe, dynamic configuration of high-performance distributed systems. We then introduce Gaia, a component-based operating system, built on a reflective middleware substrate, that exploits dynamicTAO and UIC.

DynamicTAO

DynamicTAO[10] is a reflective CORBA ORB, built as an extension of TAO[11] to take advantage of Doug Schmidt and his colleagues' extensive efforts to modularize and organize middleware software. TAO is a portable, flexible, extensible, and configurable ORB that complies with the CORBA standard and uses the Strategy design pattern[12] to encapsulate different aspects of the ORB internal engine. A configuration file specifies the strategies the ORB uses to implement aspects such as concurrency, request demultiplexing, scheduling, and connection management. At ORB startup time, the configuration file is parsed and the selected strategies are loaded. TAO is primarily targeted for static hard real-time applications such as Avionics systems. Thus, it assumes that once the ORB is initially configured, its strategies will remain in place until it completes its execution. There is very little support for on-the-fly reconfiguration.

DynamicTAO extends TAO to support on-the-fly reconfiguration while assuring that the ORB engine stays consistent (For the complete source code for dynamicTAO, see http://choices.cs.uiuc.edu/2k/dynamicTAO). This is achieved by reifying dynamicTAO's internal structure-that is, keeping an explicit representation of the ORB internal components and of the dynamic interactions among them. The reification lets the ORB change specific strategies without having to restart its execution. DynamicTAO is a reflective ORB because it allows inspection and reconfiguration of its internal engine. It achieves that by exporting an interface for

- transferring components across the distributed system,
- loading and unloading modules into the ORB runtime, and
- inspecting and modifying the ORB configuration state**.**

The infrastructure can also be used to dynamically reconfigure servants running on top of the ORB and even nonCORBA applications.

Architecture

DynamicTAO achieves reification through a collection of entities, or component configurators.[13] A component configurator holds the dependencies between a certain component and other system components. Each process running the dynamicTAO ORB contains a component configurator instance called *DomainConfigurator*. It is responsible for maintaining references to ORB instances and to servants running in that process. In addition, each ORB instance contains a customized component configurator called the *TAOConfigurator*.

The TAO Configurator contains hooks to which implementations of dynamicTAO strategies are attached. Hooks work as mounting points where specific strategy implementations are made available to the ORB. Figure 1 illustrates the reification mechanism in a process containing a single ORB instance. If necessary, individual strategies can use component configurators to store their dependencies on ORB instances and other strategies. These configurators may also store references to client requests that depend on the strategies. With this information, it is possible to manage strategy reconfiguration consistently.

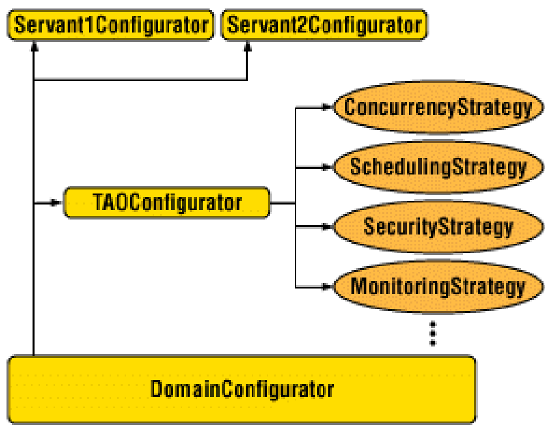

Figure 1. Reifying the dynamicTAO structure.

Component implementations are shipped as dynamically loadable libraries so that they can be linked with the ORB process at runtime. They are organized in categories representing different aspects of the ORB internal engine or different types of servant components.

Figure 2 depicts the dynamicTAO architectural framework. The *Persistent Repository* stores category implementations in the local file system. It offers methods for manipulating (for example, browsing, creating, or deleting) categories and the implementations of each category. A *Network Broker* receives reconfiguration requests from the network and forwards them to the *Dynamic Service Configurator*. The latter contains the DomainConfigurator (shown in Figure 1) and supplies common operations for dynamic configuration of components at runtime. It delegates some of its functions to specific component configurators (for example, a TAOConfigurator or a certain *ServantConfigurator*).

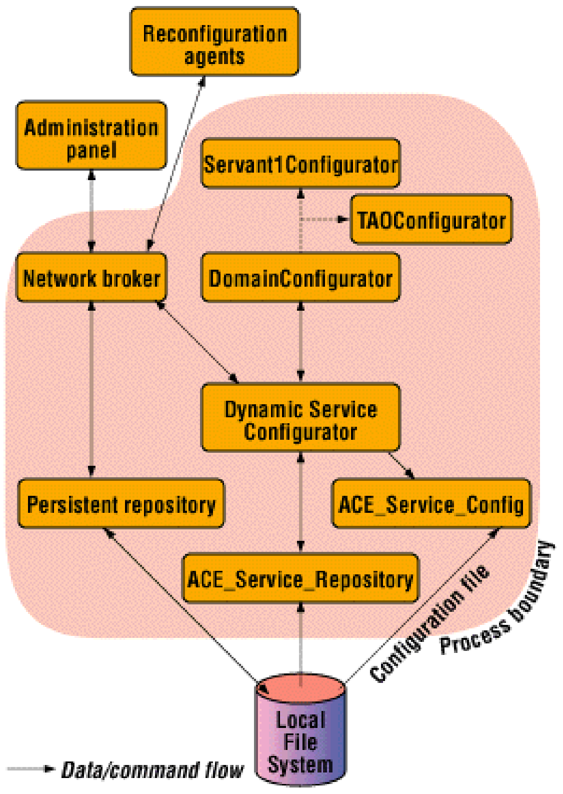

Figure 2. DynamicTAO components.

ORB consistency

Our goal is to have a minimal ORB core running without interruption while ORB strategies and servants are dynamically updated. Supporting ORB on-the-fly reconfiguration introduces several important issues that do not exist when dealing with static startup configuration. Changing an implementation at runtime implies checking possible dependencies with loaded implementations of the same category, dependencies with clients that are using old implementations, and conflicts with implementations of the same category running on other ORBs.

The initial state of a new strategy might depend on the final state of the old strategy that is going to be replaced. Also, we cannot execute some reconfiguration operations immediately because their execution would leave the ORB's state inconsistent. Switching from one strategy to another and deleting a strategy are two of the more conflicting operations. We must control several aspects so the ORB remains stable after the operation completes.

To change from one strategy to another, we must follow at least two steps. First, we determine whether the new strategy implementation is compatible with the remaining implementations by checking other strategies running in the same ORB and, in some cases, the strategies of the same category on other ORBs. When we change an implementation in a certain ORB, we might be required to change the implementations on remote ORBs (for example, if these implementations are part of a distributed protocol). Second, we might need to transfer an implementation's state to the new implementation that is going to replace it.

We must also consider how changes in a strategy implementation affect client requests that are being processed. It greatly depends on the category of the implementation. In the case of the marshalling and unmarshalling strategies, for example, changes immediately affect any subsequent request. In the case of concurrency strategies, however, changing an implementation might only affect new clients. [7]

Bringing reflective middleware to your hand

Mark Weiser defines ubiquitous computing as the concept of "everything, everywhere, all the time computing."[14] Users and their surrounding environments are digitally augmented, therefore allowing new levels of interaction and enhanced productivity.

From the software infrastructure point of view, one of the keys to enabling this continuous computing is guaranteeing seamless interoperability among ubiquitous computing devices. Although this is the task of communication middleware, its current construction assumes a set of hardware and software requirements suitable for desktop computers, not for handheld and embedded devices.

Communication middleware implementations such as COM, Java, and CORBA are too big to fit in devices with limited resources, and they do not provide configuration tools that would allow adapting them to different architectures. The problem is that applications typically require only a small subset of the middleware functionality, but they are forced to be linked with libraries providing the entire functionality. In the case of CORBA, for example, a client program that simply invokes a method on a remote object requires only the client side functionality from the ORB and either the dynamic invocation interface or the static invocation interface. Unfortunately, most ORB implementations include the entire functionality in a single library or they provide separate libraries for client and server sides, but without any option to choose a specific subset of this functionality.

Most existing middleware implementations have been designed to assist in the development of server applications running on workstations and desktops. The execution patterns and requirements of these applications are normally well known, and therefore communication middleware platforms export functionality customized for these scenarios (for example, banking applications and airline reservations).

However, in the case of ubiquitous computing, previous execution patterns no longer apply. The nature of the ubiquitous computing scenarios differs from previous scenarios. In addition, because of the heterogeneity associated with ubiquitous computing, it is not possible to provide a single static middleware implementation that fits all scenarios. Therefore, the flexibility reflective middleware introduces turns out to be an elegant approach to cope with the requirements for ubiquitous computing.

Analyzing the requirements for ubiquitous computing

When considering the devices found in ubiquitous computing scenarios (for example, PDAs, sensors, phones, and appliances), there are at least three common properties that usually apply:

- limited resources,
- heterogeneity, and
- a high degree of dynamism.

Limited resource availability varies from device to device. While devices such as handheld PDAs increasingly offer powerful CPUs and larger amounts of memory, there are other devices, such as sensors and small appliances, in which it is not possible or cost effective to add more resources. Heterogeneity is probably one of the predominant properties in ubiquitous computing. Different hardware platforms, operating systems, and software platforms imply changes in some parameters such as byte ordering, byte length of standard types, and communication protocols. Finally, the degree of dynamism present in ubiquitous computing does not exist in traditional servers and workstations. A PDA, for example, interacts with many devices and services in different locations, which implies many changing parameters such as the type of communication network, RPC protocols, and security policies. Therefore, because we can't predict all possible combinations, the software running on the PDA must be able to adapt to different scenarios to cope with such dynamism.

All previous properties affect the design of the middleware infrastructure required for ubiquitous computing. Conventional middleware platforms are not appropriate, because they are too big and inflexible. It is true though that we can customize existing middleware platforms manually for every particular device. However, this process is far from realistic, because it is not flexible enough or suitable for coping with dynamic changes in the execution environment. Only reflective middleware presents a comprehensive solution to deal with ubiquitous computing. We can maintain the applications (base level) unaltered and push these changes to the middleware platform (metalevel), which offers tools to modify the platform's behavior.

However, existing reflective middleware implementations are not yet appropriate for mobile handheld and embedded devices.[4,5] We have detected three main problems that make them unsuitable for ubiquitous computing scenarios: size, configuration options, and dependence on a single middleware platform.

Existing implementations are too heavy to fit the devices found in ubiquitous computing scenarios. The problem in most of the cases is that the middleware architecture is built using a monolithic approach and thus imposes large memory requirements. We describe a reflective middleware architecture based on component technology that lets you achieve tiny implementations (for example, a client-side CORBA ORB for PalmOS of 18KB).

Regarding the configuration options, existing implementations normally allow modifying concurrency policies, marshaling strategies, and other concerns. However, it is not only a matter of reconfiguring policies associated with some pre-established categories. In some cases, it is important to modify the internal architecture of the entire middleware engine. Therefore, the meta-interface the reflective middleware exports should not only let you choose certain predefined category implementations, but it should also let you add and remove category types and reorganize the middleware engine's internal architecture. This property allows the reconfiguration of the middleware from the point of view of the required functionality, making it possible to remove functionality when it is no longer needed and add new functionality when required.

Finally, most existing reflective middleware implementations depend on a single middleware platform (for example, CORBA, COM, and Java RMI). However, in practical ubiquitous computing scenarios, applications might use different middleware platforms on different systems. Therefore, the reflective middleware infrastructure must be able to reconfigure its internal engine to interoperate with different middleware platforms.

Ubiquitous computing scenario

To better understand all the properties and requirements we've mentioned, consider this example. A user carrying a handheld device enters a room that contains a device to control the lights and another device to control the background music. All devices are connected to a wireless network. The user can use the handheld device to control the lights (turning them on and off and setting their intensity level) as well as the music's volume. We will also assume that we have a reflective middleware platform that implements all the properties mentioned before (customizable size, flexible configuration options, and middleware-platform independence).

The light and music controllers run a single instance of an object to export their functionality, and they only require server-side functionality. The light controller uses CORBA, and the music controller uses SOAP.[15] At this point, we have enough information to configure the middleware platform these devices use; however, it is important to decide how we should configure it-statically or dynamically.

Because of the scenario's properties, it is unlikely the controller devices' requirements will change. They will always be located in the same environment with a well-defined access pattern, not many changes will be expected, and the type of network they use will not change. Therefore, the best solution (from the point of view of simplicity and code size) is to statically configure the middleware platform and then install it in the device. According to the previous requirements, the light controller's optimal configuration would be a CORBA server side with a minimal single-threaded object adapter, which simply allows registration of one object at a time and does not implement any other policy. For the light controller, we statically configure the middleware platform to export SOAP server-side functionality.

Unlike the controller devices, the handheld device only requires client-side functionality, because it will not receive any request from the other two devices. However, the handheld device needs a configuration that lets it interact both with the CORBA and SOAP objects. Because this device is highly mobile, and it's impossible to know *a priori* what its requirements will be, it isn’t appropriate to statically configure the middleware infrastructure. Therefore, the middleware platform running on the device must provide tools to support dynamic reconfiguration. When the device enters the room, it does not know about the controllers and thus requires a discovery mechanism to provide information about these controllers, such as the exported interfaces and the properties of their middleware platforms. The properties of the controllers’ middleware are used to reconfigure the handheld device’s middleware, allowing interoperability. After the reconfiguration, applications running on the handheld device use the description of the controllers’ interfaces to dynamically create requests.

In our scenario, we dynamically reconfigure the handheld device’s middleware to include client-side dynamic method invocation mechanisms for CORBA and SOAP. Providing middleware reconfiguration tools is a key requirement, but it is not enough. An additional software infrastructure associated with the room is required to discover existing devices and services, learn about their properties, and reconfigure them whenever changes are detected.

Our scenario shows how a reflective middleware platform customized for ubiquitous computing offers the tools to effectively

- integrate heterogeneous devices,
- cope with the highly dynamic behavior of such scenarios, and
- adapt its size and architecture to the required functionality.

Universally Interoperable Core

UIC (www.ubi-core.com) is a reflective middleware infrastructure that is customizable to ubiquitous computing scenarios and addresses the problems found in existing middleware platforms. UIC defines a skeleton based on abstract components, which encapsulate the standard functionality aspects common to most object request brokers (that is, network and transport protocols, connection establishment, marshaling and demarshaling strategies, method invocation, method dispatching, scheduling, object reference generation and parsing, object registration, client interface, server interface, object interface and attributes, memory management, and concurrency strategies). Concrete dynamically loadable components specialize these abstract components to implement the properties required for particular middleware platforms, devices, and networks.

UIC supports handheld devices and permits customization of the skeleton (both statically and dynamically) for heterogeneous devices and environments. The possible configurations range from minimal functionality versions with minimal memory and resource requirements to fully functional versions. This approach differs from existing solutions that port existing desktop middleware platforms to handheld devices.

Two key principles drive the design of UIC-simplicity and *What You Need Is What You Get.*[6] The simplicity principle requires a clear and easy to understand decomposition of the UIC into modules. The modular design provides separation of concerns and encapsulates the mechanisms required for each specific task in different components. This modularity minimizes unwanted dependencies, therefore allowing easy customization.

The second principle, WYNIWYG, requires that only the required functionality is present in an instance of the UIC. However, because of the UIC design, whenever more functionality is required, it can be easily introduced either statically (rebuilding the UIC with the new functionality) or dynamically (adding the new functionality at runtime). The same rule applies when some functionality is no longer required and, therefore, can be removed. It is important to note that this customization applies to the implementation of the components, the exported interfaces, and the structure of the skeleton. While the UIC defines a standard skeleton structure targeted to object-oriented request brokers (CORBA, Java RMI, and DCOM), nothing prevents the application developer from changing the structure to meet other requirements (for example, non object-oriented RPC platforms and RTP streaming).

Personalities

The UIC is an abstract entity that must be specialized to meet the particular application's requirements. This specialization defines the core's behavior and determines the type of entities with which it will be able to interact (for example, CORBA servers or Java RMI servers). We use the term *personality* to refer to the particular instance of the UIC obtained after the specialization**.**

Personalities can be classified as client-side, server-side, or both, depending on the functionality they provide. A client-side personality provides functionality to send requests and receive replies, while a server side-personality is capable of receiving requests and sending replies. This classification can be mapped directly to devices that simply need to access remote functionality, devices that need to export functionality, and devices that require both.

The UIC can be classified as single-personality and multipersonality. A single-personality UIC is capable of interacting with a single middleware platform, while a multi-personality UIC can interact with more than one platform at the same time. Note, however, that a multi-personality UIC is not equivalent to having a collection of single-personality UICs. Having several single-personality UICs implies that applications have to decide which UIC personality to use depending on the object they want to interact with. On the other hand, with multi-personality UICs, applications always use the same UIC instance and the same method invocation interface regardless of the type of the remote object. In this last case, the UIC is responsible for automatically choosing the right personality.

Configurations

We can configure UIC personalities either statically or dynamically. In static configurations, we build personalities at compile time by statically assembling all the components together. The result is a single component (the personality) that we cannot dynamically reconfigure (the only possibility is to replace the personality). The main benefit of this configuration is the personality's size.

In dynamic configurations, personalities are a collection of dynamically loadable libraries that we can fully reconfigure at runtime. The dynamic configuration's main benefit is the ability to modify the architecture of the personalities dynamically without affecting the applications. The dynamic personality's main drawback is that the core's size increases because tools for loading and unloading components are required, and each core's component becomes an independent dynamically loadable library.

There is also a hybrid configuration that combines some of the benefits of static and dynamic configurations: Components that are not likely to change are linked together, while components that will be updated or modified are built as independent loadable libraries. Depending on the number of components that can link together, the final personality's size can be significantly reduced from a purely dynamic configuration.

Benefits

The UIC is targeted to allow the handheld devices' integration with limited resource availability in standard distributed object models. The result is a homogeneous infrastructure for the applications' and services' development that hides device heterogeneity. Most of the existing solutions that integrate handheld devices in distributed environments are based on proxies, which transform native method invocation to a proprietary format that can be used by handheld devices. However, in the case of the UIC, no proxies are required since it allows interoperation with native middleware platforms. Therefore, service developers do not need to provide a proxy for each type of device.

Another key benefit of the UIC is its design. The component-based infrastructure facilitates the development of customized personalities by simply re-implementing a set of components. For example, an application programmer that requires a specialized marshaling strategy or a particular connection caching strategy simply has to re-implement those components and reuse the rest without modifications. The standard interfaces defined by the UIC components guarantee that the new component will seamlessly interoperate with the existing ones.

As an example, consider the UIC component responsible for generating the header of request invocations and parsing the header of reply invocations. This functionality is required for every middleware implementation, and therefore a generic abstract interface is provided. This interface defines two methods, the *sendRequest(Request *req)* method which receives as a parameter an abstract request object, and *receiveReply(Reply *rep)*, which generates an object that contains information related to the reply. Specializations of this generic component have to implement this generic interface, encapsulating the specific properties of the middleware protocol.

Components in the UIC do not share state. They encapsulate whatever state they require as an object, which is passed as a parameter. This technique removes dependencies and simplifies the reconfiguration task. Components simply keep a reference to other components, whose functionality is defined by the abstract interfaces provided by the UIC skeleton. In this way, a component of the UIC processes certain information and generates some results that are automatically sent to another component. When the results are passed to another component, the component passing them does not have to keep any state about the generated results.

Because of the design and the WYNIWYG principle, we can produce minimal personalities, compatible with standard middleware platforms. The personalities' size can range from 18KB for a client-side CORBA personality running on a Palm OS device to 48.5KB for a client-server CORBA personality running on a Windows CE device. Another benefit is that it is possible to predict the personality's size and trigger adaptations or configure internal personality parameters according to these size parameters.

The 18KB CORBA ORB specializes the UIC skeleton with components for establishing TCP connections, creating request headers, encoding and decoding basic types based on CDR, converting string references into objects references, and parsing reply headers. It does not include encoding rules for arbitrary complex types (for example, structs, sequences, or unions). These rules are provided by applications according to their requirements. As an example, an application that requires access to the CORBA naming service, uses a helper class that contains the definition of structs and sequences required to provide and receive information to and from the naming service. This tiny ORB relies on the IDL compiler to generate the required parameters, which use the marshaling mechanisms exported by the ORB to marshal and demarshal themselves. Although this mechanism can be seen as a limitation, according to our experience, applications know the type of parameters they will handle; therefore, it is reasonable to consider that these applications will provide the definition of the parameters they manipulate. This strategy provides an important space saving, and contrasts with standard ORBs, which by default include mechanisms to marshal and demarshal any arbitrary complex type as part of the ORB. Note, however, that if an application requires the ORB to include those encoding rules, we can do it by attaching a component that exports such functionality to the UIC.

Upgrading personalities, either statically or dynamically, allows handheld devices to adapt to different computing scenarios. As a result, we can access more services and information. Finally, by pushing all reconfiguration mechanisms to the ORB, we can implement "universal" clients and servers that can be used with different middleware implementations without having to change the application implementation. Marshaling, demarshaling, parsing headers, generating headers, scheduling, and dispatching requests are tasks present in every middleware implementation. What makes each middleware platform different is how those tasks are implemented. Reflective middleware deals with these particular details and offers tools to inspect and manipulate them either statically or dynamically.

UIC Multipersonality Core

The UIC Multipersonality Core is a specialization of the UIC that provides tools for reflection. These tools let us introspect the internal architecture of the Multipersonality Core and also introduce changes in the Core. As in the case of dynamicTAO, reflection is applied to the dependencies among the components that compose the Multipersonality Core (Component Configurator pattern).[13] Therefore, changes on those dependencies are translated into changes on the Multipersonality Core's behavior. For example, attaching SOAP specialized components to a multipersonality core leads to a behavior change of the

ORB, allowing it to interact with SOAP objects. Due to the diversity of functional aspects, it is important to make sure that the type of a component matches the type of the hook where it is attached. We can achieve this by using runtime type information.

Figure 3 illustrates the Multipersonality Core's architecture, which is divided into two parts: the Core Configuration and the Dynamic Configurator (tools for dynamic reconfiguration). The Core Configuration is the collection of UIC components that defines the Multipersonality Core's behavior. The Dynamic Configurator implements the reflective mechanisms and stores the information about dependencies among components. As Figure 3 shows, the Dynamic Configurator is implemented as a servant object that runs on top of the current Multipersonality Core configuration. This approach introduces two main benefits:

1. it allows dynamic modification of the reflective mechanisms using different instances of the Dynamic Configurator; and
2. it registers the same Dynamic Configurator servant with more than one personality at the same time, therefore allowing different types of clients (for example, CORBA and SOAP) to reconfigure the Core**.**

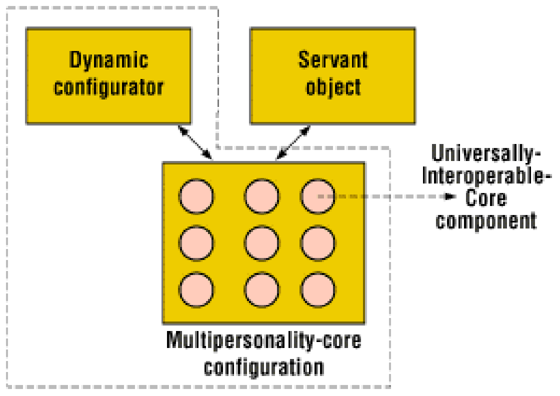
Dynamic configurator
Servant object
Universally-Interoperable-Core component
Multipersonality-core configuration

Figure 3. The UIC Multipersonality Core.

Performance and size

Here we present the size of different personalities as well as the performance of one of those personalities (the client and server CORBA personality). Existing personalities are fully interoperable with existing middleware implementations. The CORBA personalities, for example, allow sending requests to CORBA compliant objects, and CORBA compliant objects can also send requests to objects implemented on top of a server-side UIC CORBA personality. Therefore, we can register UIC CORBA objects with existing CORBA naming services, traders, and event services.

Table 1 contains the sizes of some UIC personalities built for different platforms. The UIC Hybrid Multipersonality allows manipulating personalities (for example, ClientCORBA, ServerCORBA, and ClientJavaRMI) as standalone components. Therefore, we can add and remove personalities dynamically, though it is not possible to reconfigure individual personalities. With this HybridMultipersonality we can register a servant object implementation with different personalities simultaneously. The size of this personality includes the associated DynamicConfigurator. We can implement the

CORBADynamicServer and the SimpleTCPDynamicServer (customized TCP based protocol implementation) personalities as components that can be installed with the HybridMultipersonality. Finally, the CORBAStaticClient, the CORBAStaticServer, and the CORBA Static Full (client and server) are standalone implementations that we can use without the HybridMultipersonality. All CORBA personalities described in this article export a customized dynamic method invocation mechanism, implement a subset of data types (all simple types, structs, and some sequences) and detect exception conditions raised at the server, but do not parse the exception.

UIC Personality	WindowsCE (SH3) (Kbytes)	Windows 2000 (Kbytes)	PalmOS
HybridMultipersonality	39	88	N/A
CORBA Dynamic Server	28	80	N/A
SimpleTCP Dynamic Server	30	84	N/A
CORBA Static Client	29	72	18KB
CORBA Static Server	35	84	N/A
CORBA Static Full (client + server)	48.5	100	31KB

Table 1. Sizes for some UIC configurations.

Figure 4 presents performance numbers that prove two important issues. First, the flexibility provided by reflective middleware is not necessarily translated into slower execution times. Second, reflective middleware makes it possible to obtain highly efficient customized ORB implementations that meet the exact requirements of applications.

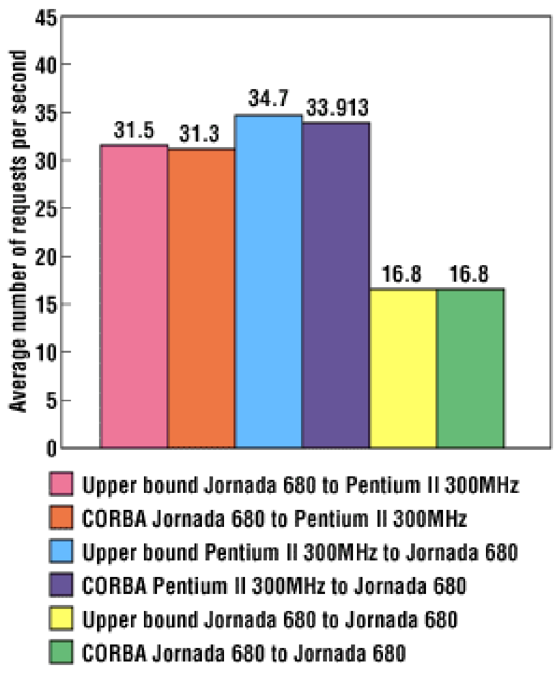

Figure 4. Requests per second using the UIC Client-Server Static CORBA Personality with a customized and dynamic invocation interface. Left to right column order corresponds to top to bottom key order.

Figure 4 contains performance numbers in terms of number of requests per second. The test involves a CORBA client and a CORBA server object using the UIC CORBA Static Full Personality. The server implements a method called *cubeIt*, which receives a long integer and returns its cube. The client repeatedly invokes the operation using our customized dynamic invocation interface implemented by the CORBA personality. The client produces statistics based on the average total round-trip time for a remote method invocation. The test consists of 300 requests repeated 10 times. The final result is the average of the result of those 10 tests (their value is in turn the average of the 300 invocations). The socket connection is opened in the first request and reused afterwards.

In order to provide an upper bound for the maximum rate of CORBA requests we could achieve, we built a simple program that does not use CORBA and has a client that sends a 52-byte packet (same size as the packet sent in the CORBA test) to the server through a TCP/IP socket, the server reads the packet (does not process it) and replies with another 28-byte packet (same size as the packet sent in the CORBA test) which is then read by the client. This simple program has no additional computational overhead. Neither the client nor the server processes the packets; this gives the maximum rate of requests that can be sent using the header sizes used by the CORBA personality. The CORBA personality incurs a number of overheads besides the network time. It must create the request header, marshal the client parameters, parse the header at the server side, demarshal the parameters, find the object, call the method, marshal the result parameters, create the reply header, parse the reply header, and demarshal the result parameters.

Figure 4 presents three scenarios, and each one divided into two subtests: the upper bound CORBA test case. first scenario is a UIC server object running under Windows 2000 on Pentium II 300MHz with 128MB of RAM, client CE 2.11 Jornada 680 Handheld PC SH3 32 bit processor at 133MHz, 16 MB RAM. In second scenario, same devices are used, but the UIC CORBA client object runs on the Windows 2000 device and the server on the Jornada 680.

The third scenario consists of two Jornada 680 Handhelds, one of them hosting the UIC CORBA client and the other one hosting the UIC CORBA server. The network in all the scenarios is a RangeLAN2 wireless network with a maximum theoretical bandwidth of 1.6Mbps. Many users not involved in this experiment share the wireless network so the results presented here are a lower bound to what the UIC performance can be. Also, the personalities used for these tests could be further optimized. The UIC CORBA implementation does not introduce any major overhead, and the times are clearly bounded to the netowrk delay times. This scenario is not unrealistic since most ubiquitous computing scenarios present several devices sharing the same network, and, therefore, network delays will exist.

For comparison, we ran the same test on desktop computers running Windows 2000 on an Intel Pentium III processor 733MHz with 256MB of RAM. In this test, we ran the client in one of the desktops and the server in the other. In this scenario the network is a 100Mb Ethernet, which is also shared by users not involved in the experiment. The results we obtained are depicted in Figure 5 and show an additional overhead of 4.4 percent in the CORBA test due to RPC related functionality (non-existing functionality in the upper bound test).

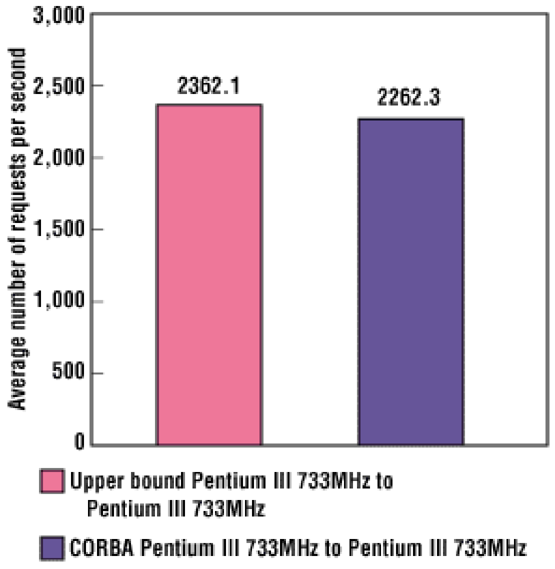

Figure 5. Requests per second using the UIC Client-Server Static CORBA Personality with a customized and dynamic invocation interface. Left to right column order corresponds to top to bottom key order.

Active spaces

We have discussed reflective middleware implementations that allow seamless interoperation among heterogeneous devices and make it possible to dynamically adapt to changing requirements. However, we have not yet addressed some important questions. Who triggers the adaptations? How can a middleware implementation discover relevant entities and learn about their properties? Where are specific component implementations stored and how can they be obtained? Who decides which is the optimal ORB configuration?

Our current research focuses on the key issues that will enable arbitrary ubiquitous computing scenarios and provide answers to these questions. We consider that handheld and embedded devices require an external software infrastructure, running on the physical space where those devices are located. Such software infrastructure introduces a new computational model that we call *active spaces*. We define an active space as an interactive and programmable system composed of physical devices and digital services associated with a physical space, uniformly coordinated by a software infrastructure and populated by a dynamic group of mobile users. The inherent complexity associated with active spaces requires a software infrastructure capable of orchestrating all the elements present in such environment, by allowing them to operate independently but in a cooperative fashion.

This coordination is analogous to the functionality exported by a traditional operating system that is responsible for managing a collection of hardware resources within a single computer.[16] Thus, what we need is an operating system for active spaces, and our group at the University of Illinois is developing just that.

Gaia is a component-based operating system built on a reflective middleware substrate. We can divide it into four main blocks:

- Unified Object Bus;
- Gaia Services;
- Gaia Application Model; and
- Gaia Active Space.

The Unified Object Bus is a dynamic, platform-independent component execution environment. It provides tools to locate, distribute, load, and unload components dynamically and it is responsible for seamlessly integrating different middleware infrastructures. All services and applications are built on top of this Unified Object Bus.

Gaia is not only a toolkit to assist in the development of ubiquitous computing applications, but is also an active system capable of orchestrating users, services, and devices in active spaces. We achieve this through a collection of services responsible for managing different aspects of the active space functionality-discovery, security, location, user accounts, data and code distribution and adaptation, and state.

Using distributed objects, Gaia abstracts physical spaces and their heterogeneous computing infrastructure as homogenous first class entities that can be addressed and manipulated. Thus, it is possible to develop applications that run in the context of an active space rather than in the context of a particular device. To support this concept, Gaia defines an application model inspired on the Model-View-Controller design pattern. [17]

As an example, consider an active hospital. Each active surgery room runs an instance of an application that uses the patient as the model of the application (the patient’s vital signs), a monitor and the surgeon’s headphone as views of the application, and the surgeon hands as the controller of the application (for example, special movements of his hands to indicate certain commands). The Gaia application model automatically instantiates two adapters: one between the model and the monitor, and another one between the model and the headphone (both of them configured as views). As a result, the monitor displays different charts representing the patient’s vital signs, and the headphone "tells" the doctor the values of those vital signs.

Finally, Gaia provides a programmatic interface to manipulate the active space. The idea is to abstract the active space as an object with well-defined attributes and operations. This abstraction allows manipulating a space as a homogeneous execution environment instead of a collection of heterogeneous entities. The underlying reflective middleware infrastructure provides a solid base for dynamically managing the policies and mechanisms used in the active space. More information on the current status of Gaia can be found at http://choices.cs.uiuc.edu/gaia. See the Related Work sidebar for additional research on reflective middleware.

Acknowledgments
This research was supported by NSF grant NSF0086094.

References

1. *CORBA v2.2 Specification*, Object Management Group, Framingham, Mass., 1998.

2. N. Brown and C. Kindel, *Distributed Component Object Model Protocol DCOM/1.0,*1998, www.microsoft.com/com (current 4 June 2001).
3. S. Maffeis, "Adding Group Communication and Fault-Tolerance to CORBA," *Proc. USENIX Conf. Object-Oriented Technologies*, Monterey, Calif., 1995, pp. 135-146.
4. G. Blair et al., "An Architecture for Next Generation Middleware." *Proc. Middleware 98*, Lake District, England, Springer-Verlag, London, 1998.
5. Y.M. Wang and W.J. Lee, "COMERA: COM Extensible Remoting Architecture," *Proc. Fourth USENIX Conf. Object-Oriented Technologies and Systems (COOTS 98)*, pp. 79-88.
6. R. Campbell, A. Singhai, and A. Sane, *Reflective ORBs: Support for Robust, Time-Critical Distribution*, Springer-Verlag, New York, 1997.
7. M. Roman, F. Kon, and R.H. Campbell, "Design and Implementation of Runtime Reflection in Communication Middleware: the dynamicTAO Case," *Proc. ICDCS99 Workshop on Middleware,* IEEE CS Press, Los Alamitos, Calif., 1999, pp. 122-127.
8. F. Kon, G. Blair, and R.H. Campbell, *"*Workshop on Reflective Middleware," *Proc. FIP/ACM Middleware2000*, 2000.
9. G. Kiczales and J. des Rivières, and D.G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, Cambridge, Mass., 1991.
10. F. Kon et al., "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms and Open Distributed Processing (Middleware2000),* Springer-Verlag, New York, 2000.
11. D.C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Comm. Magazine*, IEEE CS Press., Los Alamitos, Calif., vol. 37, no. 4, 1999, pp. 54-63.
12. E. Gamma et al., *Design Patterns, Elements of Object-Oriented Software*, Addison-Wesley, Reading, Mass., 1995.
13. F. Kon and R.H. Campbell, "Dependence Management in Component-Based Distributed Systems,*"* *IEEE Concurrency*, vol. 8, no. 1, Jan.-Mar. 2000. pp. 26-36.
14. M. Weiser, "The Computer for the 21st Century," *Scientific American*, Scientific American, New York, 1992, pp. 94-104.
15. K. Scribner and M.C. Stiver, "Understanding SOAP: The Authoritative Solution," SAMS, 2000.
16. M. Roman and R.H. Campbell, "Gaia: An Operating System to Enable Active Spaces,*" Proc. Ninth SIGOPS European Workshop*, Kolding, Denmark, 2000.
17. G.E. Krasner and S.T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *Object-Oriented Programming*, vol. 3, no. 1, 1998, pp. 27-49.

Related Work

Recent research in middleware has identified limitations on existing CORBA implementations, which led to ORB extensions for dealing with specific aspects such as real-time,[1] group communication,[2] and fault-tolerance.[2] Our goal, on the other hand, is to provide a generic skeleton in which different kinds of customizations can be performed using reflection.[3]

The Distributed Multimedia Research Group at the Lancaster University has proposed a reflective architecture for next generation middleware.[4,5] They developed a prototype using the Python interpreted language in which the programmer is able to inspect and change the implementation at runtime. The level of reflection is much higher than in dynamicTAO and UIC because, in their Python system, it is possible to add or remove methods from objects and classes dynamically and even change the class of an object at runtime. Their research has emphasized dynamic configurability through a well-defined *open binding* model, which allows multiple reflective levels. In contrast, our research concentrates on a simpler reflective model, focusing on high performance and customization to devices with limited resources. In our model, the reflective mechanisms are not included in the normal flow of control; they are only invoked when needed.

COMERA (COM Extensible Remote Architecture) provides a framework based on Microsoft COM that allows users to modify several aspects of the communication middleware at run-time.[6] It relies on the *Custom Marshaler* interface exported by COM, as well as the componentized architecture design that allows the use of user-specified components. By using COMERA, system developers can customize the middleware according to application requirements.

Reflective middleware is coming to the fore as an active research field. Dozens of researchers around the globe are focusing in this area, which is the subject of the *Workshop on Reflective Middleware*[7] that is held in conjunction with the IFIP/ACM Middleware conference. The workshop brings different points of view on issues like mathematical models, formal verification, architectures, systems, applications, and performance of reflective middleware.

1. T.H. Harrison, D.L. Levine, and D.C. Schmidt. "The Design and Performance of a Real-time CORBA Object Event Service," *Proc. OOPSLA*, ACM Press, New York, 1997.
2. S. Maffeis and D.C. Schmidt, "Constructing Reliable Distributed Communication Systems with CORBA," *IEEE Comm. Magazine*, IEEE CS Press, Los Alamitos, Calif., vol. 14, no. 2, 1997.
3. P. Maes and D.Nardi, *Metalevel Architectures and Reflection*, North-Hollan, 1987.
4. G. Blair et al., "An Architecture for Next Generation Middleware." Proc. Middleware 98, Lake District, England, 1998.
5. G. Blair et al., "On the Design of Reflective Middleware Platforms," *Proc. IFIP/ACM Middleware2000 Workshop on Reflective Middleware*, IEEE CS Press, Los Alamitos, Calif., 2000, pp. 3-5.
6. Y.M. Wang and W. Lee, "COMERA: COM Extensible Remoting Architecture," *Proc. Fourth USENIX Conf. Object-Oriented Technologies and Systems* (COOTS 98), 1998.
7. F. Kon, G. Blair, and R.H. Campbell, "Workshop on Reflective Middleware," *Proc. IFIP/ACM Middleware'2000*, 2000 .

Manuel Roman is a PhD student in computer science at the University of Illinois at Urbana-Champaign. He worked on dynamicTAO, a dynamically reconfigurable reflective ORB, and 2k, a network-centric operating system based on distributed objects. He is currently working on the overall design and implementation of Gaia OS, a component-based middleware OS to enable active spaces. Roman is also the manager of Ubicore LLC, which is the company responsible for the Universally Interoperable Core. He received his BS and MS degrees in computer science from La Salle School of Engineering (Ramon Llull University) in Barcelona, Spain. Contact him at Digital Computer Lab, 1304 W. Springfield Ave., Urbana, IL 61801, USA; mroman1@cs.uiuc.edu, mroman@ubi-core.com.

Fabio Kon is an assistant professor in computer science at the University of São Paulo, Brazil. He participated in designing and implementing SODA, a consistent distributed file system based on leases; MAXAnnealing, a tool for algorithmic musical composition; a scalable multimedia distribution system; dynamicTAO, a dynamically configurable reflective ORB; and 2K, a network-centric operating system based on distributed objects. Kon received his BS and MS degrees in computer science from the University of São Paulo and a BA degree in music from the São Paulo State University. He received his PhD in computer science from the University of Illinois at Urbana-Champaign in automatic configuration of component-based distributed systems. Contact him at Rua do Matao, 1010, Sco Paulo - SP, 01423-001, Brazil; kon@ime.usp.br.

Roy H. Campbell is a professor of computer science at the University of Illinois at Urbana-Champaign. His research interests include operating systems, distributed multimedia, network security, and ubiquitous computing. His recent research accomplishments include: VDP, an adaptive continuous media transport protocol; a robust video compression and packetization scheme for unreliable networks; and dynamic security policy systems for distributed objects, mobile computers, and active networks. Campbell is currently involved in the active spaces research project, as well as other security projects. He received his BSc in mathematics from the University of Sussex, and his MSc and PhD in computing from the University of Newcastle upon Tyne. Contact him at Digital Computer Lab, 1304 W. Springfield Ave., Urbana, IL 61801, USA; roy@cs.uiuc.edu.