# Making Middleware Communication Architecture Reconfigurable

Nikolaos D. Georgantas, Daniele Sacchetti, Valérie Issarny



## To cite this version:

Nikolaos D. Georgantas, Daniele Sacchetti, Valérie Issarny. Making Middleware Communication Architecture Reconfigurable. [Research Report] RT-0279, INRIA. 2003, pp.19. inria-00069899

## HAL Id: inria-00069899
## https://inria.hal.science/inria-00069899

Submitted on 19 May 2006

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Making Middleware Communication Architecture Reconfigurable*

Nikolaos Georgantas — Daniele Sacchetti — Valérie Issarny

**N° 0279**

Avril 2003

THÈME 1

ISSN 0249-0803

INRIA
ROCQUENCOURT

# Making Middleware Communication Architecture Reconfigurable

Nikolaos Georgantas*, Daniele Sacchetti †, Valérie Issarny ‡

Thème 1 — Réseaux et systèmes
Projet ARLES

Rapport technique n° 0279 — Avril 2003 — 19 pages

**Abstract:** To deal with emerging network technologies and services, as well as with varying application requirements and dynamic environmental conditions, middleware has to be reconfigurable. This requirement highly affects the middleware communication architecture, which comprises inherent middleware communication mechanisms and the underlying network architecture. In this report, we introduce an architectural model for reconfigurable middleware communication architecture. Towards this objective, our model re-applies application layer semantics to the middleware communication architecture. It specifies a complete interaction and addressing mechanism for middleware communication architecture objects. Further, an incorporated reflection architecture enables dynamic reconfiguration.

**Key-words:** middleware, communication architecture, flexibility, reconfigurability, RPC, pipeline model, reflection, mobile distributed system

* Nikolaos.Georgantas@inria.fr
† Daniele.Sacchetti@inria.fr
‡ Valerie.Issarny@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 —Télécopie : +33 1 39 63 53 30

# Une solution à la reconfiguration de l'architecture de communication de middleware

**Résumé :** Pour faire face aux nouvelles technologies réseaux, ainsi qu'aux exigences variées des applications et aux caractéristiques dynamiques de l'environnement, le middleware doit être reconfigurable. Cette exigence a une influence importante sur l'architecture de communication de middleware, qui comprend les mécanismes de communication du middleware et de l'architecture réseau sous-jacente. Dans ce rapport, on introduit un modèle d'architecture de communication reconfigurable, similaire à celui fourni par l'architecture middleware globale et exploité par l'application. Notre modèle définit ainsi un mécanisme complet d'interaction et d'adressage pour les objets de l'architecture de communication. De plus, la reconfiguration dynamique de l'architecture de communication est supportée au moyen de la notion de réflexion.

**Mots-clés :** middleware, architecture de communication, flexibilité, reconfigurabilité, RPC, modèle de pipeline, réflexion, système distribué mobile

# 1 Introduction

Middleware has been established as the key architectural paradigm in building distributed systems. Middleware provides to applications a high-level abstraction of the underlying system and network architecture, in terms of: (i) functionality enhancement, by adding advanced interaction semantics, reliability, fault recovery, security to the communication architecture; (ii) functionality coordination, by supporting distribution transparency, by masking system and network heterogeneity; (iii) new service functionality, by supporting object naming and location, object life-cycle, object activation/deactivation and persistence. Middleware addresses both functional properties, as in supporting communication services, and non-functional properties, as in supporting QoS requirements, e.g., performance, timeliness, security, dependability.

Middleware often has to execute in a context of varying application requirements and dynamic environmental conditions, as well as to deal with heterogeneity in existing networks and with emerging network technologies and services. Hence, there is a need for multiple specialized profiles of middleware functionality in all phases of an application's lifecycle: at development time, at deployment time and at run-time. To fulfill this requirement, middleware shall have certain structural properties that enable its reconfiguration and extensibility. Especially for the run-time phase, middleware shall, further, support mechanisms for getting application requirements and environmental conditions, and for dynamic reconfiguration. Reconfigurability addresses, among other, the *middleware communication architecture*, which consists of inherent middleware communication mechanisms and the underlying network architecture.

In this report, we introduce an architectural model for designing and developing a reconfigurable middleware communication architecture for the distributed object middleware paradigm. Our model supports the following features:

- Enhanced protocol interaction capabilities.
- Structural flexibility for protocols enabling easy restructuring and reuse.
- Static (at application deployment) and dynamic (at application run-time) configurability of the middleware communication architecture.
- Application inspection and application-driven adaptation of the middleware communication architecture.

To illustrate feasibility and effectiveness of the proposed architectural model, an implementation scenario applied on a mobile distributed system is presented.

The report is structured as follows. In Section 2, the standard middleware communication architecture is presented. Section 3 introduces the concept behind our architectural model. In Section 4, the architectural model is specified. Section 5 presents the object and protocol addressing scheme of the model, while Section 6 describes the model's reflection architecture. An implementation scenario is presented in Section 7. Section 8 discusses related work, and, finally, Section 9 concludes the report.

## 2 Middleware Communication Architecture

The architecture of a distributed system incorporating middleware can be abstracted into three structural layers: (i) the application layer, (ii) the middleware layer and (iii) the underlying network architecture.

The application layer contains application objects, which are the users of the services provided by the middleware layer. The primary functionality offered to applications by the middleware layer is a discrete (operational) interaction mechanism, collectively having the form of a generic remote procedure call (RPC) service, as opposed to a continuous (stream) interaction mechanism. We call this the *external interaction service (EIS)*, as it is offered by the middleware layer to an external layer, i.e., the application layer. This service is supported by the fundamental part of the middleware layer, which we call the *middleware core*. The EIS is the only service expected from a minimal middleware infrastructure. In the example of CORBA [8], the middleware core is represented by the ORB. Using the EIS, a client application object can communicate with a remote server object simply by object method invocation. A couple of specialized middleware core objects, called stub on the side of the client object and skeleton on the side of the server object, undertake the effective forwarding of the method invocation to the server across the network and the returning of the method result to the client. The stub is a local representative of the remote server to the client, being in effect the direct recipient of the client invocation. The skeleton represents locally the invoking client to the server object, being in effect the actual invoker of the server method. The EIS architecture is depicted in Figure 1.

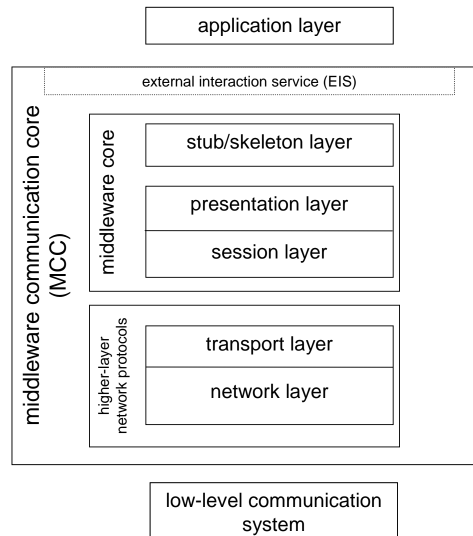


Figure 1: The external interaction service (EIS) architecture

The EIS is supported by the protocols of the middleware core, which we call *middleware core protocols*. These are commonly:

- A presentation layer protocol, which performs encoding of a method invocation to a common data format, appropriate for transmission across the network. On the receiving side, the peer protocol entity decodes the received message back to a method invocation and encodes the method result to be sent to the caller.
- A session layer protocol, which undertakes sending the encoded method invocation to a peer protocol entity and receiving the method result.

In CORBA, the General Inter-ORB (GIOP) protocol specification defines the Common Data Representation (CDR) for encoding method calls, the GIOP message formats for managing the method invocation session, and the GIOP transport adapter for mapping GIOP onto an underlying transport protocol. The commonly used mapping of GIOP onto TCP is called the Internet Inter-ORB (IIOP) protocol. Thus, for CORBA, the GIOP/IIOP stack provides the presentation and session protocols discussed above.

Underneath the middleware layer lies the network architecture, which further supports the EIS. The network architecture is commonly constituted by:

- A hierarchy of transport layer and network layer protocols, which we call *higher-layer network protocols*. In a common deployment of CORBA, TCP and IP are the higher-layer network protocols used.
- The low-level communication system. This is built independently based on low-level hardware and software architectures. For fixed IP networks the IEEE 802.3 Ethernet LAN specification, and for wireless IP networks the IEEE 802.11b wireless LAN specification represent potential low-level communication systems.

To help further discussion, we define the *middleware communication core (MCC)* to be the combination of the middleware core and the higher-layer network protocols. The MCC supports the EIS. We exclude the low-level communication system, which also supports the EIS, as it is closely associated with hardware and uses low-level communication mechanisms. In the CORBA example, the MCC contains the protocol stack of GIOP/IIOP/TCP/IP.

In addition to middleware core functionality, supplementary services may be provided by the middleware layer such as a naming service for locating server objects by name, a life cycle service for managing object creation and deletion, a security service, a transaction service, a persistence service for retaining object state. These services are deployed as application objects with respect to the middleware core — e.g., they may use the EIS, and may be incorporated by applications as organic building blocks, offering standard support of basic or advanced functionality. Further, the middleware layer may provide to applications a stream interaction service. A specialized transport protocol hierarchy for continuous media streams can be built "at application level", using the EIS for control-plane communication. For the purpose of structural clarity, we assume that this protocol hierarchy does not combine with the MCC protocols supporting the EIS. It is implemented as a full protocol stack lying

over the common low-level communication system. The entire middleware communication architecture is depicted in Figure 2.

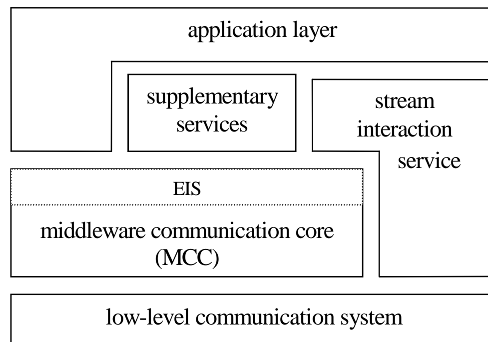


Figure 2: The entire middleware communication architecture

We conclude from the above that all building blocks of a distributed system — we ignore the low-level communication system — have access to an advanced interaction service — the EIS — with the exception of the MCC, which uses conventional protocol messages for its own interactions.

# 3 Towards a Configurable Middleware Communication Core

The main architectural principle imposed by the use of middleware is that applications are deployed as distributed organizations of objects interacting by method invocation, detached from their remote interaction mechanism. This mechanism is effectuated by the MCC. This principle offers to applications enhanced communication capabilities and structural flexibility. Nevertheless, the MCC follows a conventional static architectural model, which imposes the following features on its protocols:

- Protocols use protocol messages for their remote interactions. Protocol messages follow a fixed structure, which limits protocol interaction capabilities.

- Protocols are inflexibly dependent on each other, as each protocol conventionally uses the services provided by its underlying protocol for its remote communication.

These features lead to monolithic MCC architectures.

We are seeking to introduce flexibility within the MCC by re-applying the principle adopted for applications: The MCC shall be deployed as an organization of objects interacting by method invocation, detached from their remote interaction mechanism. This leads to

the need for another MCC, at a second level of abstraction; however, this recursion cannot practically be realized. Consequently, the unique MCC will be used. Hence, we introduce a unified interaction mechanism, which will be used by all objects of the MCC for their remote interactions, in the way the EIS is used by application objects. We call this the *internal interaction service (IIS)*. The IIS is provided by the MCC itself. Hence, MCC objects will be at the same time both providers and users of the IIS. The EIS and IIS relation to the MCC is depicted in Figure 3. As an example, we apply this concept to CORBA, assuming an object-oriented implementation of the GIOP/IIOP/TCP/IP protocol hierarchy of the MCC. An object within, e.g., GIOP may invoke, for the purpose of managing a GIOP session, a method of a remote peer object. This method invocation will be effected by the IIS, employing the whole MCC protocol hierarchy, or, possibly, a subset of this hierarchy, depending on the requirements of the calling object.

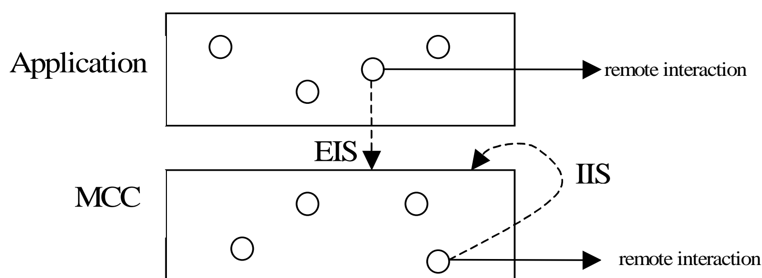


Figure 3: The EIS and IIS relation to the MCC

The next section specifies an architectural model for the design and development of a flexible MCC, and more specifically for the design and development of the supported EIS and IIS. Focus is on establishing both static and dynamic configurability for the MCC. Towards this objective, our architectural model applies the scheme of reflection [12] to certain engineering aspects of the MCC.

# 4 MCC Architectural Model

## 4.1 EIS and IIS vs. Conventional Protocol Stacks

To delimit the EIS and IIS, and to map them on well-known schemes, we provide a short description of a conventional protocol stack architecture. A communication protocol is commonly constituted by two[1] peer entities, communicating with each other through a predefined set of protocol messages. Protocols are usually layered in protocol stacks. Each protocol layer of a protocol stack uses the services of its underlying layer to transmit its

[1] It could be more than two entities, in the case of group communication protocols.

protocol messages. Hence, the layering hierarchy within the protocol stack further imposes an *interaction hierarchy.* A protocol message of a certain protocol layer may be used to carry data (a protocol message) coming from the above layer, and/or control information of the layer itself produced within the layer. Received control information may cause a change in the state of the peer protocol layer, and is consumed within this layer. Received data is forwarded by the peer layer to its above layer. Forwarded data may, ultimately, be either control information for an upper layer or application layer data.

In this well-defined protocol stack scheme, we can abstract along the protocol stack a usually single application data transmission mechanism employing the entire protocol stack, and several control transmission mechanisms. Each one of them is associated to a discrete protocol layer, and employs the subset of the protocol stack that lies below the protocol layer. The control and data transmission mechanisms may be functionally combined by sharing the capacity of protocol messages for throughput efficiency reasons.

For our architectural model, we define that the conventional application data transmission mechanism is mapped onto the EIS, while the conventional control transmission mechanisms of the several protocols are mapped onto the IIS. The IIS is made explicit and takes a form analogous to the EIS. This entails the necessity that the EIS and IIS are explicitly discrete and functionally independent. This is a deviation to the conventional protocol stack scheme described above, in which the control and data transmission mechanisms are functionally combined.

## 4.2 MCC Architectural Elements

We define the *protocol module* to be the structural element of the MCC. A protocol module can be a complete protocol or a self-contained protocol function, e.g., a retransmission or a flow-control function; we call the latter more specifically a *protocol function module.* Protocol modules are composed of objects, which we call *protocol objects.* Protocol modules can be associated with each other to produce combined functionality. Accordingly, we define that application objects are grouped into *application modules*, which can be complete applications or application components.

We define for each application module an *EIS coordinator module*, and for each protocol module an *IIS coordinator module* to be the *service access points (SAPs)* for using the EIS and IIS, respectively. These modules incorporate, among other, stub and skeleton functionality for each object of the associated module, thus allowing the transparent use of the EIS or the IIS.

The EIS and IIS are each supported by a processing sequence of protocol modules forming a *protocol pipeline.* The pipeline scheme implies that the output of a protocol module serves as input to the next module in the pipeline. By changing the configuration of the protocol pipeline, i.e., selecting alternative protocol modules, a variety of EIS and IIS services may be defined. A protocol module may participate in more than one protocol pipelines, thus supporting several EIS or IIS services. A protocol module may, as well, use more than one IIS services, depending on the requirements of its objects. There may be differentiation

per object or per interaction. Correspondingly, an application module may use several EIS services. The architectures of the EIS and the IIS are depicted in Figure 4.

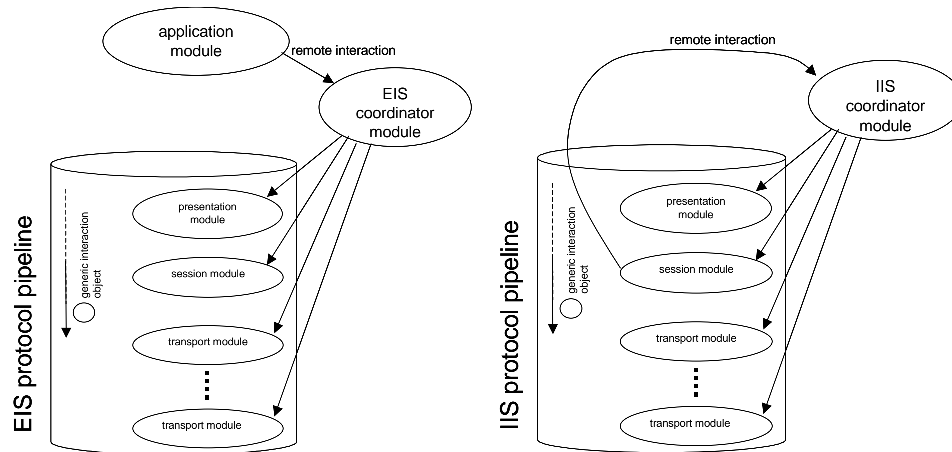


Figure 4: The EIS and IIS architectures

Along a protocol pipeline, an interaction is processed, being carried within a generic structure called the *generic interaction object*. This interaction object may take alternative forms. It may support carrying the content of an interaction in a typed form, which means that the structural properties of the interaction are maintained. In this way, individual fields of the interaction may be selectively processed by protocol modules that can apprehend the content of the interaction. From some point of the protocol pipeline downwards, the interaction object may be serialized to an untyped form within a message buffer. An untyped message buffer can only be treated as an array of bytes. These advanced processing capabilities offered by the interaction object may be supported by applying reflection. The interaction object shall expose a meta-interface enabling its manipulation. The interaction object provides standardized base-level interface and meta-interface to all protocol modules, thus contributing to the configuration flexibility of the protocol pipeline.

The pipeline configuration information for all EIS services used by an application module is stored within the associated EIS coordinator module. Accordingly, the pipeline configuration information for all IIS services used by a protocol module is stored within the associated IIS coordinator module. On the client side of an interaction, a coordinator module drives the generic interaction object downwards along the protocol pipeline, delivering it in turn to each protocol module of the processing sequence, while on the server side, a similar coordinator module executes the same task upwards. The inverse procedure is followed for the result of the interaction. For processing a generic interaction object, each protocol module of the pipeline interacts locally only with the coordinator module, and not with any other

protocol module of the pipeline. The interaction of a protocol module with the coordinator module uses a standardized interface. This offers enhanced flexibility in the configuration of a protocol pipeline, allowing for easily adding, removing or replacing protocol modules.

The choice of grouping interaction coordination responsibility for all objects of an application or protocol module within the associated coordinator module is justified by the fact, that the objects of a module commonly have similar interaction requirements, which can be collectively managed.

### 4.3 Protocol Pipeline Configuration

Following the typical middleware communication architecture, a protocol pipeline supporting the EIS or the IIS comprises:

- A presentation protocol module, which maps an interaction onto the *generic interaction object* that will be processed along the protocol pipeline. A further presentation conversion of the interaction object, e.g., from a typed form to an untyped form, may be necessary at some later stage along the protocol pipeline. However, we consider here the simple case.

- A session protocol module, which manages a session for sending the interaction content to a peer protocol entity and receiving the interaction result. Depending on the requirements of the interacting objects and on the capabilities of the underlying transport service, alternative session modules may implement or coordinate various RPC schemes such as synchronous or asynchronous (announcement, call-back, deferred synchronous), multi-cast or broadcast.

- A sequence of transport protocol modules. A transport protocol sequence establishes a certain *transport hierarchy*, meaning that each module enhances the aggregate functionality of the modules that are underneath it in the hierarchy. This hierarchy shall generally be conformed to, meaning that the relative position of the modules shall not change. Nevertheless, protocol function modules may more flexibly be placed at different positions enhancing the aggregate service provided by the hierarchy.

The configuration of the transport sequence may vary according to the required interaction service. We distinguish among the following cases for the transport sequence:

- The EIS commonly employs the entire transport hierarchy. This provides the EIS with a fully reliable, fault recovery capable interaction mechanism.

- The IIS may employ the entire transport hierarchy.

- The IIS serving a certain transport module may employ a transport sequence comprising all transport modules that lie below the module in the transport hierarchy. This is similar to the conventional protocol stack scheme. The transport service provided by this transport sequence may not be reliable. If a reliable transport service

is needed, the transport service may be enhanced with additional functionality, e.g., with a timer-based retransmission mechanism. This functionality shall be provided by an additional protocol function module placed on top of the transport sequence.

For the EIS, the protocol pipeline configuration does not present any particularity. The particularity for the IIS is twofold:

1. The IIS violates the interaction hierarchy identified for a conventional protocol stack, since an interaction of any protocol module is served by the presentation module at the top of the conventional protocol hierarchy.

2. A protocol module may both use the IIS and support it, participating in its protocol pipeline.

Regarding the first aspect, as long as the second aspect is not true, there is no engineering implication: the interactions of a protocol module are served by an independent protocol stack. Regarding the second aspect, we define an *interaction service instance* within a module, to be the individual support of an interaction service by the module. To make this aspect feasible, the protocol module shall allocate explicitly discrete and independent resources for each one of the *interaction service instances* it deploys. This means that:

- Discrete object instances within the module shall be used for supporting each interaction service instance.

- A discrete interaction object shall be used for carrying each interaction; this constraint has already been satisfied.

In this way, an object A currently supporting an interaction service instance A, may initiate a remote interaction using an independent interaction service instance B. An independent instance B of the same object will support the interaction service instance B. This scheme is depicted in Figure 5.

The notion of an interaction service instance is to be clarified within the context of the protocol module. For example, a module may:

- Deploy a new service instance for each interaction it serves.

- Maintain two service instances, one for serving its own interactions, and one for serving the interactions of other modules. This requires that the module is aware of the originator of every interaction it serves.

- Deploy a new service instance upon a new service demand, if the currently allocated service instances are in a blocked state, possibly waiting for the completion of interactions initiated by them.

A special internal interaction case for a protocol module, which shall be treated with caution, is a *resource allocation interaction.* A resource allocation interaction is performed between two peer protocol modules for *explicitly* setting up a session. This session makes

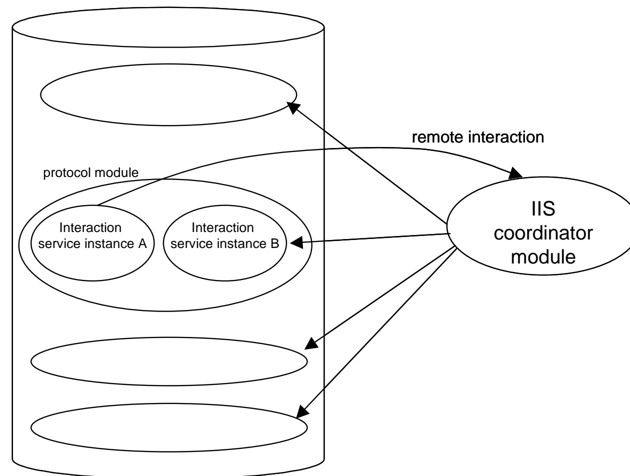


Figure 5: Using and supporting the IIS

part of the protocol functionality and will be used for supporting an interaction service instance. We consider the case of a protocol module that always explicitly allocates a session for supporting an interaction service instance. A session setup interaction initiated by this module shall use a protocol pipeline that does not include the module itself, because this would provoke an infinitely recursive session setup at the module, unless an already established session exists and can be used for supporting the new session setup.

# 5 Object and Protocol Addressing

In distributed object middleware platforms, addressing is oriented towards identifying a specific object. A remote object reference is a structure containing the address of the hosting node, which is commonly a network layer address, a communication port number, which may be a well-known one, and an object key uniquely identifying the object within the address space defined by the host address and port. For example in CORBA deployment over IP networks, an Interoperable Object Reference (IOR) contains an IP address, a TCP port and an object key. In SOAP/Web services middleware [11], an object may be identified by a HTTP Uniform Resource Locator (URL), containing the host address, an optional port number and the file system path to the object resource. Generation of an object reference is done by the middleware core, when an object is created and registered with the core for receiving remote invocations. The use of the naming service, as well as of the stub and skeleton objects allows transparent address resolution of a server object and transparent physical addressing of a remote invocation to the server.

In conventional protocol stacks addressing is oriented towards identifying a specific protocol entity. A node is usually characterized by its network layer address, e.g., an IP address. For protocol layers above the network layer, addressing is further specialized to identify the specific layer, e.g., a TCP port for HTTP. Within a protocol layer, addressing discriminates among multiple sessions between peer layers with session identifiers. When a protocol message is delivered to a protocol layer, the enclosed addressing information may not identify a specific protocol entity. In this case, the message may be dispatched in terms of its message type by a protocol layer manager entity, which possibly receives all messages of the layer. For example a session setup message may be dispatched by the manager to a new session object being instantiated to handle the new session. A node address may be resolved by use of some routing mechanism. A session identifier is determined either explicitly, by executing a session setup procedure, or implicitly, where no session setup procedure is carried out.

In our architectural model, any well-established addressing scheme, as the ones described above for distributed object middleware, could be used for the application objects. Addressing inside the MCC shall consider both distributed object middleware addressing and conventional protocol addressing.

We define, in our architectural model, a unified scheme for both application layer and MCC addressing, which follows the SOAP/Web services middleware addressing scheme. Addressing is oriented towards identifying a specific resource. This could be an object, which is most appropriate for applications, or, a module or a session, which is most appropriate for protocols. A full address of a resource is a URL containing the host address and the file system path to the resource. Within a URL, protocol modules may be simply identified by a well-known name, e.g., tcp for TCP. Protocol objects may also be identified by a well-known name or an object key, which could be a number generated upon object instantiation. Session identifiers may be numbers attributed upon session setup. Succession of identifiers in a URL path may denote encapsulation, e.g., tcp/connection_id references a connection object inside the TCP module. Following the conventional protocol addressing discussed above, an interaction referencing a protocol module and not a specific object within the module, will be addressed to a pre-determined object, e.g., the protocol module manager.

For applications, address resolution of a remote server object may use a naming service. The EIS coordinator module undertakes the physical addressing of an interaction to the server; this is transparent to applications. For protocols, address resolution may employ a routing mechanism, be the outcome of a session setup procedure, or use well-known names; address resolution is part of the protocol functionality. Thus, the address of a peer protocol entity is not opaque to a protocol module and may be resolved within the module itself. For this reason, an object of the module intending to interact with the peer entity, may itself communicate to the associated IIS coordinator module the peer entity address. By applying reflection, this can be done through a meta-interface provided by the coordinator module. In this way, an IIS coordinator module offers a base-level interface, through which its associated protocol module may transparently use the supported IIS, and a meta-interface, through which the protocol module may intervene in this transparent service support.

# 6 Dynamic Configuration of the MCC

In our discussion so far, we have already seen two applications of reflection in our architectural model: one concerning the manipulation of the generic interaction object, and one concerning the addressing of a remote protocol entity. We define here the general reflection architecture inside our architectural model.

Reflection is applied to enable inspection on the EIS and IIS. The EIS and IIS coordinator modules, which encapsulate the configuration information of the respective interaction services, are the most suitable points to provide inspection. Coordinator modules expose meta-interfaces, through which the protocol pipeline configuration of an interaction service may be inquired or modified by inserting, removing or replacing protocol modules. Further, tuning of certain protocol parameters, or intervening in the processing of a specific interaction — as by conveying addressing information — may be possible through the meta-interfaces. We introduce two meta-levels of reflection. We distinguish between a meta-level accessed by applications and a meta-level accessed by MCC protocols. We call the former the *external meta-level* and the latter the *internal meta-level*. The external meta-level is populated by EIS coordinator modules meta-interfaces, while the internal meta-level is populated by IIS coordinator modules meta-interfaces.

A protocol module may access the meta-interface provided by its associated IIS coordinator module to configure the IIS it uses. This provides a MCC self-inspection. An application module may access the meta-interface provided by its associated EIS coordinator module to configure the EIS it uses. Further, an EIS coordinator module provides indirect access to the IIS coordinator modules associated to the protocol modules employed by the EIS. In this way, an application may configure the IIS services used by the protocols it employs. A full inspection of the application over the underlying MCC is in this way possible. To realize this scheme, pre-configured, possibly minimal, IIS and EIS may be established at system bootstrapping. Applications that are network-aware may use the default EIS to negotiate the reconfiguration of the EIS and IIS. The reflection architecture specified by our architectural model is depicted in Figure 6.

# 7 Implementation Scenario

Our architectural model enables both static and dynamic configurability within the middleware communication architecture. We are presenting an initial implementation scenario to illustrate support for static configurability. We apply our architectural model to a wireless distributed system, where there is a high demand for flexibly integrating and supporting enhanced features within the middleware communication architecture. We experiment with enhancing the wireless distributed system with wide-area mobility.

The baseline network is an IEEE 802.11b wireless local network executing in infrastructure-based mode, which allows wireless terminals to access a wide-area wired IP network through a base station, which interfaces both the wireless and the wired network. The

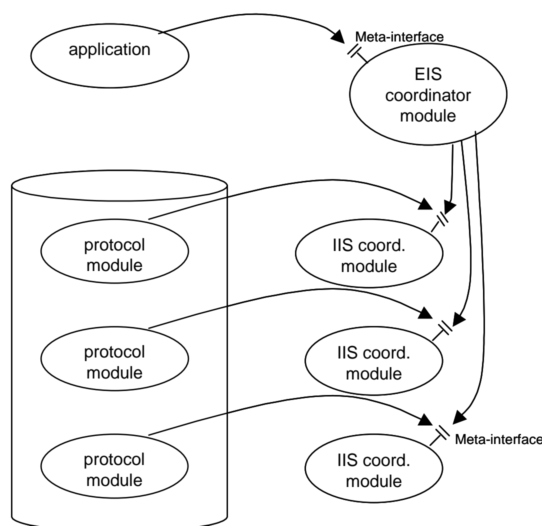


Figure 6: Reflection within the architectural model

middleware technology employed over this IP infrastructure is SOAP/Web Services [11]. A SOAP/HTTP protocol stack is used to convey remote object method invocations.

We build our system in Java, using existing open source Java implementations of SOAP and HTTP. We are deploying *Apache AXIS 1.0*, which is a SOAP implementation by Apache Software Foundation. We are deploying *Jigsaw 2.2.2*, which is a Web server platform by W3C, providing a sample HTTP 1.1 implementation among a variety of other features. For presentation conversion of Java object method invocations to XML data types carried in a SOAP message, we use the mapping defined in the *JavaTM API for XML-Based RPC (JAX-RPC)*. We use a J2SE 1.3 platform over Linux.

Accommodating wide-area terminal mobility within the Internet has long been an active field of research. Mobile IPv4 and mobile IPv6 protocols by IETF provide a mobility support solution at IP layer, making mobility transparent to higher layers. While this provides certain advantages, such as, that higher-layer protocols and applications intended for fixed IP networks may as well be deployed in mobile IP networks, it deprives higher-layer protocols and applications from being mobile-aware and optimized for mobility. A second solution, oriented to mobile distributed systems, introduces mobility management at the middleware layer. We are adopting this solution. We adopt a model of mobility support based on the *Wireless Access and Terminal Mobility in CORBA Specification* [9], which is a specification by OMG, heavily affected by the work of the ACTS DOLMEN [10] project. This specification enhances the CORBA model with wide-area mobility, which is added as an external feature

based on the CORBA bridging concept. This scheme makes mobility completely transparent to stationary ORBs. We are adapting this work to the SOAP/Web Services middleware.

In the initial, stationary configuration of our implementation, the MCC comprises a merged presentation/SOAP module, a HTTP module, and a TCP connection management module interfacing to the OS implementation of TCP. The open source implementations of SOAP and HTTP are adapted to our architectural model, so that they follow the EIS scheme. In the initial configuration there is no need for an IIS, because the presentation/SOAP and HTTP modules do not execute any remote control interactions.

For the mobile-capable configuration, the adopted wireless CORBA model requires introduction of three new architectural entities: the Terminal Bridge integrated into the wireless terminal's MCC, the Access Bridge integrated into the base station's MCC, and the Home Location Agent integrated into a stationary terminal's MCC. This requires extra protocol functionality to be inserted into the MCC. A new protocol module implementing a specialized session protocol, called the HTTP Tunneling Protocol (HTTP-TP), is introduced into the wireless terminal's and base station's MCCs. The HTTP-TP follows the specification of the GIOP Tunneling Protocol defined in the wireless CORBA model. The HTTP-TP module executes a number of remote control interactions, which means an IIS shall be implemented. Easily and effectively integrating the extra protocol functionality will illustrate the support of our architectural model for static configurability of the MCC.

# 8 Related Work

Our work relates to work in various areas. Regarding the area of middleware internal architecture engineering, in [1], design patterns are applied within the internal architecture of TAO, which is a CORBA-compliant open-source ORB providing QoS and real-time support, to enable extensibility and dynamic configurability. Reference [2] further provides a survey on meta-programming mechanisms for ORB middleware, allowing middleware to adapt to changing application requirements or environmental conditions during application deployment or runtime. These meta-programming mechanisms have been applied to TAO. Of special interest is the pluggable protocols mechanism, which decouples transport protocols of an ORB from its higher-level component architecture, allowing new protocols to be easily added. With respect to work in [1] and [2], which applies design and implementation optimizations to certain individual system and network aspects of an ORB, our work introduces a more abstract, unifying model for ORB communication architecture.

Regarding the area of communication architectures, the specification and implementation of an end-system communication middleware architecture, called Da CaPo++, is reported in [3]. Da CaPo++ is not built on any middleware technology, exposes, however, many of their properties. It supports a rich set of communication capabilities. It incorporates collections of modular protocols, allowing their flexible configuration. In [4], RPC semantics are studied, and a configurable group RPC service is specified and constructed by means of micro-protocols, within the x-kernel communication programming environment. Micro-protocols implement individual properties. A dependency graph defines the relationship

between micro-protocols imposing constraints in the configuration of the RPC service. In [5], building communication architectures from components, called micro-protocols, within the Ensemble framework is presented. By using a formal verification tool, a certain communication architecture configuration is checked against a specification. Further, formal optimization is applied to this configuration. Our work adopts several architectural principles from the work presented in [3]-[5]. However, our work complies with the powerful middleware paradigm, which extends the scope of our solution to a wide area of applications.

Regarding the area of reflection, the Flexinet [6] middleware platform, developed in Java by researchers at APM, incorporates reflection in many aspects of its functionality. An RPC mechanism, other than the standard Java RMI, is built from scratch. Dynamic configuration of the RPC protocol hierarchy is supported through a meta-interface. The Java Core Reflection mechanism is used for dynamic manipulation of a generic interaction object. Stubs as well as remote object references, called name objects, are dynamically manipulated. Application inspection on several internal middleware mechanisms is supported through meta-interfaces. The OpenORB [7] reflective middleware platform combines reflection and component frameworks (CF) technology. OpenORB is structured as a set of configurable component frameworks, each one undertaking a domain of concern, e.g., protocol CF within the communications layer or buffer management CF within the resource layer. Reflection is used to discover the current structure and behavior, and to enable selected changes at runtime. In this way, OpenORB can be specialized to different domains, such as multimedia or real-time systems. In our future work we plan to further specialize the reflection architecture of our model. To this end, the above work, in [6]-[7], will certainly provide us with valuable knowledge.

# 9 Conclusion

In this report, we have addressed the issue of middleware configurability, focusing on the middleware communication architecture. We have introduced an architectural model, which enables both static and dynamic reconfiguration of the MCC. This model re-applies to the MCC the fundamental principle introduced by the distributed object middleware paradigm: "Applications are deployed as distributed organizations of objects interacting by method invocation, detached from their remote interaction mechanism". Providing a remote method invocation capability to MCC objects, and decoupling them from their remote interaction mechanism, enables enhanced interaction capability and structural flexibility for the MCC. The latter is further leveraged by decoupling MCC protocols from each other.

We have specified two interaction mechanisms with similar semantics, the EIS serving application interaction, and the IIS serving MCC interaction. We have also established a unified scheme for both application object and protocol addressing. Further, we have established a reflection architecture, which enables both self-inspection and application inspection of the MCC. Last, we have presented an implementation scenario illustrating static configurability, in which a wireless distributed system is enhanced with wide-area mobility.

In our future work, we consider specifying and developing an architectural framework based on our model, which will integrate standard interfaces and functionality classes for the entities of the model. We will further specialize the reflection architecture and will implement dynamic reconfigurability.

## References

[1] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware", *IEEE Commun. Mag.*, vol. 37, pp.54-63, Apr. 1999.

[2] N. Wang, K. Parameswaran, D. Schmidt and O. Othman, "Evaluating Meta-Programming Mechanisms for ORB Middleware", *IEEE Commun. Mag.*, vol. 39, pp.102-113, Oct. 2001.

[3] B. Stiller, C. Class, M. Waldvogel, G. Caronni, and D. Bauer, "A Flexible Middleware for Multimedia Communication: Design, Implementation, and Experience", *IEEE J. Select. Areas Commun.*, vol. 17, pp. 1580-1598, Sept. 1999.

[4] M. A. Hiltunen and R. D. Schlichting, "Constructing a Configurable Group RPC Service", in *Proc. Int.Conf. on Distributed Computing Systems*, Vancouver, Canada, May 1995

[5] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, R. Constable, "Building Reliable, High-Performance Communication Systems from Components", in *Proc. ACM SOSP'99*, South Carolina, Dec. 1999

[6] R. Hayton and ANSA Team, "FlexiNet Architecture", APM Ltd., Cambridge (UK), 1999.

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

[8] Object Management Group, *Common Object Request Broker Architecture (CORBA/ IIOP)*, 3.0.2 ed., 2002.

[9] Object Management Group, *Wireless Access and Terminal Mobility in CORBA Specification*, 2002.

[10] S. Trigila, K. Raatikainen, B. Wind, and P. Reynolds, "Mobility in Long-Term Service Architectures and Distributed Platforms", *IEEE Pers. Commun.*, vol. 5, pp. 44-55, Aug. 1998.

[11] SOAP Version 1.2 Part 0: Primer, http://www.w3.org/TR/soap12-part0.

[12] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *A System of Patterns (Pattern-Oriented Software Architecture)*, John Wiley & Sons Ltd, England, 1996.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803