ACT: An Adaptive CORBA Template to Support Unanticipated Adaptation

Technical Report MSU-CSE-03-22

S. M. Sadjadi and P. K. McKinley
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824
Email: {sadjadis,mckinley}@cse.msu.edu

August 2003

***Abstract*— This paper proposes an Adaptive CORBA Template (ACT), which enables run-time improvements to CORBA applications in response to unanticipated changes in either their functional requirements or their execution environments. ACT enhances CORBA applications by transparently weaving adaptive code into their object request brokers (ORBs) at run time. The woven code intercepts and adapts the requests, replies, and exceptions that pass through the ORBs. ACT itself is language- and ORB-independent. Specifically, ACT can be used to develop an object-oriented framework in any language that supports dynamic loading of code and can be applied to any CORBA ORB that supports portable interceptors. Moreover, ACT can be integrated with other adaptive CORBA frameworks and can be used to support interoperation among otherwise incompatible frameworks. To evaluate the performance and functionality of ACT, we implemented a prototype in Java to support unanticipated adaptation in non-functional concerns, such as quality-of-service and system-resource management. Our experimental results show that the overhead introduced by the ACT infrastructure is negligible, while the adaptations offered are highly flexible.**

Keywords: middleware, CORBA, dynamic adaptation, interoperability, request interceptor, dynamic weaving, proxy, quality-of-service, mobile computing.

I. INTRODUCTION

CORBA applications comprise autonomous programs typically hosted on heterogeneous platforms and distributed over heterogeneous networks. Although an application may be targeted at a particular type of execution environment when originally developed, over its lifetime the application is likely to be ported to new environments. Indeed, a key benefit of CORBA and other middleware platforms is that they mask the distribution of resources across a network and hide differences among computing platforms and networks. However, the need to achieve acceptable quality-of-service over different underlying technologies has given rise to extensive research and development in adaptive middleware [1]–[9]. Moreover, the potential diversity of platforms and networks hosting a given CORBA application increases the likelihood that the application will be required to accommodate situations not anticipated during the original development. In these cases, new adaptive code needs to be introduced to the application after it is deployed. Examples include code to enhance the fault-tolerance of critical application components, to detect and respond to new security attacks, and to mitigate variable channel conditions and frequent disconnections that arise when an application is ported to a wireless network. However, adding new adaptive functionality to an extant application is complicated when (1) the source code of the application is unavailable, (2) the source code is available but modifying it directly is undesirable, or (3) the application is required to run continuously and cannot be easily taken off-line for upgrade.

In this paper, we propose the Adaptive CORBA Template (ACT), which supports such "unanticipated" adaptation in CORBA applications. ACT enables dynamic improvements to CORBA applications in response to changes in their functional requirements or in non-functional concerns, such as quality-of-service, fault-tolerance, and security. We refer to ACT as a framework *template*, because it provides a generic model for constructing and enhancing adaptive CORBA frameworks. Several such frameworks have been developed recently to support quality-of-service [4], [10], real-time processing [8], [9], and fault tolerance [11], [12]. As depicted in Figure 1, ACT-based frameworks can be implemented in different programming languages such as Java and C++, and ACT can used to extend existing adaptive CORBA frameworks such as QuO [4]. Moreover, ACT can be used to enable interoperation among otherwise incompatible frameworks, such as OpenORB [2] and TAO [8].

An ACT-based framework can be integrated with a CORBA application transparently at run time: new types of adaptation can be added without recompiling the application. The key insight into how to achieve this transparency is the concept of a *generic interceptor*, which is a particular type of CORBA portable request interceptor [13]. Although a generic interceptor must itself be registered with the ORB of a CORBA application at startup time, its presence enables registration of *specific* request interceptors to be postponed until run time. In this manner, a generic interceptor can dynamically weave new adaptive code into the ORB as the application executes. The adaptive code can intercept and adapt requests, replies, and exceptions that pass through the ORB. In addition to a generic interceptor, ACT also defines a rule-based interceptor,

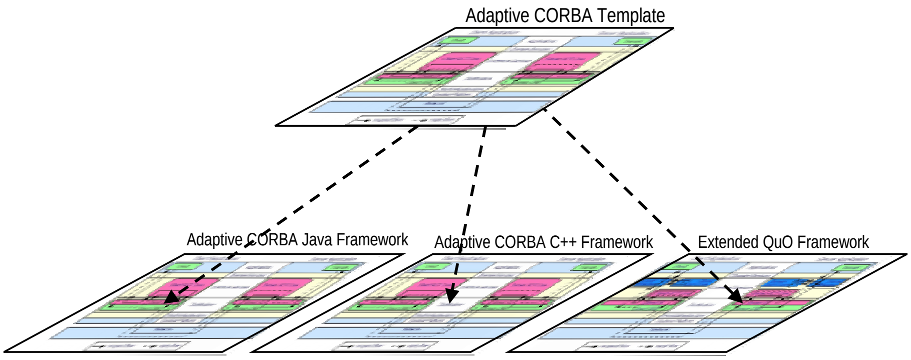


Fig. 1. ACT as a template for adaptive CORBA frameworks.

which adapts intercepted requests according to a set of rules that also can be loaded dynamically at run time.

ACT can be used to develop an object-oriented framework in any programming language that supports dynamic loading of code and can be applied to any CORBA ORB that supports portable interceptors [13]. We developed a Java prototype of ACT as well as a set of administrative consoles that enable manual adaptation of applications at run time. The prototype uses ORBacus [14], a Java ORB from IONA Technologies. To demonstrate the seamless interaction of ACT with other adaptive CORBA frameworks, we coupled ACT with the QuO framework [4] developed at BBN Technologies. The resulting framework is able to weave quality-of-service (QoS) aspects (referred to as *qoskets* in QuO terminology [15]) into CORBA applications both at compile time and at run time. To evaluate the functionality and performance of this hybrid framework, we used it to enhance an existing image retrieval application [16] as it executes. The results at this case study show that ACT introduces negligible overhead to an application while supporting transparent and flexible adaptation at run time.

The remainder of this paper is organized as follows. Section II discusses background and related work. Section III describes the ACT architecture and its prototype implementation. Section IV describes the case study in which we coupled ACT with QuO. Finally, Section V concludes the paper and discusses future work.

II. BACKGROUND AND RELATED WORK

In this section, we review two relevant topics: middleware layers as defined by Schmidt [17], and CORBA portable interceptors as defined by OMG [13]. We also describe how ACT relates to other adaptive middleware projects.

A. Middleware Layers

Schmidt [17] decomposes middleware into four layers: host-infrastructure, distribution, common-services, and domain-services. Figure 2 illustrates these layers. Since the operation of ACT involves all four, we provide a brief overview here.

The host-infrastructure layer resides directly atop the operating system and provides a higher-level API that hides the heterogeneity of hardware platforms, operating systems and, to some extent, network protocols. The host-infrastructure layer provides generic services to the upper middleware layers by encapsulating functionality that would otherwise require many tedious, error-prone, and non-portable code, such as socket programming and thread communication primitives. ACE [18], Rocks [19], MetaSockets [20], and Eternal [12] are examples of adaptive middleware in this layer. ACT can be used to load and instantiate such components in running systems so as to enhance communication among the constituent ORBs.

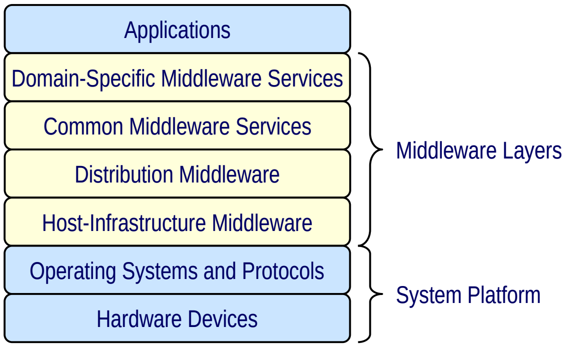


Fig. 2. Middleware layers defined by Schmidt [17].

The distribution layer resides atop the host-infrastructure layer and provides high-level programming abstractions, such as remote object operations, to the developer. Using the distribution layer, a developer can write a distributed application in a similar way to a stand-alone application. In addition, this layer hides the heterogeneity of network protocols and, in some cases, the heterogeneity of operating systems and programming languages. CORBA [21], DCOM [22], and Java RMI [23] are the main solutions to distribution middleware. Adaptive ORBs, which reside in this layer, include TAO [8], DynamicTAO [1], CIAO [24], ZEN [9], UIC [25], OpenORB [2], mChaRM [26], Electra [27], and FlexiNet [28]. By intercepting and modifying communication services, ACT can be used to enable interoperation among ORBs that support adaptability using different mechanisms.

The common-services layer resides atop the distribution layer and provides services such as fault tolerance, security, load balancing, event propagation, logging, persistence, real-time scheduling, and transactions. Examples of adaptive CORBA frameworks that provide such services include QuO [4], IRL [11], FRIENDS [29], and TAO load balancing [30]. ACT enables distributed applications to dynamically incorporate such adaptive services transparently.

The domain-specific layer resides atop the common-services layer and is tailored to a specific class of distributed applications. Unlike the common-services layer, the services in this layer can be reused only for a specific domain of applications. The Boeing Bold Stroke architecture [31] is an example of adaptive middleware in this layer that benefits from the capabilities of real-time CORBA ORBs and supports configurable and reusable avionics services.

B. CORBA Portable Request Interceptors

CORBA Portable Request Interceptors, defined by OMG [13], provide a transparent mechanism to intercept messages (defined as requests, replies, and exceptions) inside the ORBs of a CORBA application. According to the specification, a request interceptor is considered as part of an ORB and must be registered with the ORB at its initialization time (notably, a request interceptor cannot be registered with the ORB at run time). Figure 3 shows the flow of a CORBA request/reply sequence with interceptors present. This application comprises two autonomous programs hosted on two computers connected by a network. Let us assume that the client has a valid CORBA reference to the CORBA object realized by the servant. The client's request to the servant is first received by the stub, which represents the CORBA object at the client side. The stub marshals the request and sends it to the client ORB, where the request is intercepted by the client request interceptor. The interceptor can inspect requests, create new requests, and raise exceptions. For example, the ForwardRequest exception can be used to forward a particular request to a *different* CORBA object. However, to ensure portability, interceptors are not allowed to reply to intercepted requests or to modify the parameters [13]. This restriction limits the ability of request interceptors alone to adapt the behavior of CORBA applications.

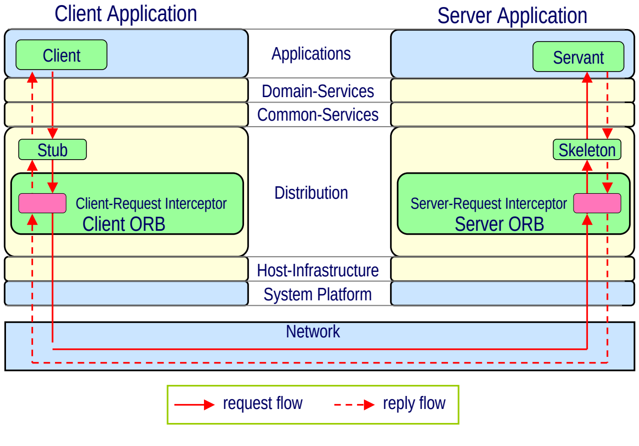


Fig. 3. A simple CORBA application with request interceptors.

Continuing the example, let us assume that the client-request interceptor in Figure 3 simply passes the request unmodified. In this case the client ORB sends the request to the server ORB, where it is intercepted by the server-request interceptor. Again, let us assume that the request is passed unmodified, in which case it is delivered to the servant by way of a skeleton, which unmarshals the request. The servant replies to the request, by way of the server ORB, where the reply also is intercepted. Eventually, the reply will be received by the client ORB and is intercepted by the client-request interceptor before it reaches the client.

As we shall discuss in Section III, the generic interceptors in ACT are in fact CORBA portable interceptors. The interceptors provide "hooks" into the interaction between clients and servants. Moreover, they use the ForwardRequest exception to deliver requests to a *proxy*, a CORBA object that is not prohibited from replying to or modifying the request.

C. Relationship between ACT and other projects

ACT is intended to complement adaptive middleware frameworks and to support interoperation among incompatible frameworks. Specifically, ACT can be used to dynamically load components of one adaptive framework into an existing CORBA application that was developed using a different framework. By transparently intercepting requests and replies, ACT enables such applications to exploit adaptive functionality defined in other frameworks. We refer to such a system as a *framework gateway*. Next, we discuss several adaptive middleware frameworks and their relationship to ACT. We group the frameworks into three categories: aspect-oriented middleware, reflective middleware, and interception-based middleware.

Aspect-Oriented Middleware. Aspect-oriented middleware enables separation of functional aspects from its non-functional aspects (*e.g.*, quality-of-service, security, and fault-tolerance) of a distributed application. One of the most extensive projects in this area is Quality Objects (QuO) [4], which provides an adaptable framework to support QoS in CORBA applications. QuO weaves QoS aspects, referred to as *qoskets*, into the applications at compile time by wrapping stubs and skeletons with specialized *delegates*, which intercept requests and replies for possible modifications [4]. In Section IV, we show how ACT can interact with QuO transparently to enable unanticipated adaptation by dynamically weaving new qoskets into the application at run time. AspectIX [5] is an aspect-oriented distribution middleware that is based on the distributed object model [32], in which an object comprises multiple fragments distributed across nodes. AspectIX enables dynamic weaving of non-functional aspects into object fragments. Although AspectIX is CORBA compliant, AspectIX dynamic adaptation cannot be used if AspectIX interoperates with other CORBA compliant non-AspectIX ORBs. To solve this problem, ACT could be used as a framework gateway that hosts fragments of a distributed object at the non-AspectIX ORBs. Squirrel [6] is an adaptive distribution middleware, specialized for streaming data, that supports QoS for multimedia applications. Again, ACT could be used as a gateway that enables interoperation among non-Squirrel and Squirrel ORBs. Specifically, ACT can enable non-Squirrel ORBs to accept and use *smart proxies* [33] transparently so that they could better communicate with Squirrel ORBs. Finally, in an earlier study by our group [34], we proposed a two-step approach to dynamic adaptation using aspects: (1) preparing hooks in the source code of an application at compile time using rule-based aspects developed in AspectJ version 1.0 [35] and (2) loading Java rules at run time to support dynamic adaptation. ACT adds domain- and language-independence to this method and enables run-time weaving of rule-based aspects.

Reflective Middleware. Reflective middleware uses computational reflection to enables inspection and modification of middleware dynamically during application execution [36]. DynamicTAO [1] and UIC [25] are CORBA-compliant reflective ORBs that employ the component-configurator pattern [37] to support dynamic adaptation. OpenORB [2] is a reflective ORB that provides explicit binding of remote objects and enables unanticipated dynamic adaptation using structural and behavioral reflection [38]. ZEN [9] is a Java ORB that use Java reflection and the virtual component pattern [39] to provide a minimal-footprint ORB that loads ORB components on demand. To exploit the adaptive features provided by these ORBs, one must use the same ORB in all the autonomous programs that constitute the CORBA application. ACT could be used as a gateway between a non-reflective CORBA-compliant ORB and a reflective ORB, as well as between two reflective ORBs of different types, to enable interoperation while exploiting the adaptive features of the reflective ORBs. To do so, ACT can host different reflective ORBs transparently while intercepting all CORBA requests, replies, and exceptions and passing them to the appropriate reflective ORB.

Interception-Based Middleware. The concept of transparently intercepting CORBA requests and replies has been used in several projects. Friedman et al. [40] use CORBA portable interceptors [13] to enhance the client side of a CORBA application by introducing proxies that can cache data and forward requests to other servants. This work is among the first to exploit CORBA portable interceptors for transparent adaptation. In the IRL project, Baldoni et al. [11] use portable interceptors to transparently introduce their implementation of fault-tolerant CORBA [13] to CORBA-compliant ORBs. Moser et al. [12] also use an interception-based approach to transparently introduce their implementation of fault-tolerant CORBA (Eternal [12] over Totem [41]) to CORBA applications. Eternal, however, employs an operating-system interception-based approach instead of CORBA portable interceptors. In general, the above projects focus on modifying program behavior in a particular way, for example, to enhance fault tolerance. In contrast, ACT uses the concept of generic interceptors to enable adaptation of different types (security, fault tolerance, QoS) in ways that were not anticipated at application development time. Moreover, generic interception enables ACT to be used as a framework gateway.

III. ACT ARCHITECTURE AND OPERATION

The Adaptive CORBA Template (ACT) is intended to support the construction and enhancement of adaptive CORBA frameworks. ACT enables CORBA applications to support unanticipated adaptation at run time without the need to modify, recompile, and relink the application source code. We introduce ACT by defining its core components and by describing their interaction with the rest of the system.

A. ACT Core Components

Figure 4 shows the flow of a request/reply sequence in a simple CORBA application using ACT. For clarity, details such as stubs and skeletons are not shown. ACT comprises two main components: a generic interceptor and an ACT core. A *generic interceptor* is a specialized request interceptor that is registered with the ORB of a CORBA application at startup time. The client generic interceptor intercepts all outgoing requests and incoming replies (or exceptions) and forwards them to its ACT core. Similarly, the server generic interceptor intercepts all the incoming requests and outgoing replies (or exceptions) and forwards them to its ACT core. A CORBA application is called *ACT-enabled* if a generic interceptor is registered with all its ORBs at startup time. If, in addition to the generic interceptors, all the ACT core components are also loaded into the application, the application is called *ACT-ready*. Making the application ACT-ready can be done either at startup time or at run time.

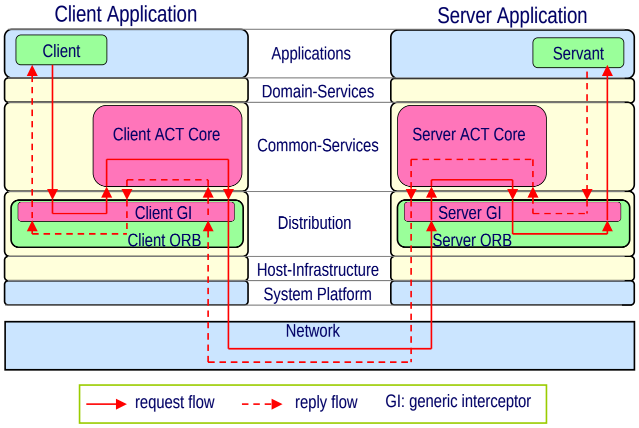


Fig. 4. ACT configuration in the context of a simple CORBA application.

Figure 5 shows the flow of a request/reply sequence intercepted by the client ACT core. The components of the core include dynamic interceptors, a proxy, a decision maker, and an event mediator. Each component is described in turn.

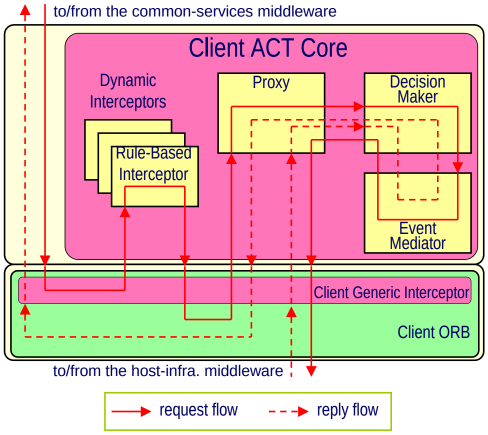


Fig. 5. ACT core components collaborating with the rest of the system.

Dynamic Interceptors. According to the CORBA specification [13], a request interceptor is required to be registered with an ORB at the ORB initialization time. The ACT core enables registration of request interceptors after the ORB initialization time (at run time) by publishing a CORBA interceptor-registration service. Such request interceptors are called *dynamic interceptors*. Dynamic interceptors can be unregistered with the ORB at run time also. In contrast, a request interceptor that is registered with the ORB at startup time is called a *static interceptor* and cannot be unregistered with the ORB during run time. We note that the code developed for a static interceptor and that for a dynamic interceptor can be identical, the difference being the time at which they are registered. In ACT, only generic interceptors are static.

A *rule-based interceptor* is a particular type of dynamic interceptor that uses a set of rules to direct the operations on intercepted requests. The rules can be inserted, removed, and modified at run time. A *rule* consists of two objects: a condition and an action. To determine whether a rule matches a request, a rule-based interceptor consults its condition object. Once a match is found, the interceptor sends the request to the action object of the rule. Since it is part of a CORBA portable interceptor, the action object cannot itself reply to the request or modify the request parameters [13]. The action object can, however, send new requests, record statistics, or raise a `ForwardRequest` exception, causing the request to be forwarded to another CORBA object such as a proxy.

Proxies. A *proxy* is a surrogate for a CORBA object that provides the same set of methods as the CORBA object. Unlike a request interceptor, a proxy is not prohibited from replying to intercepted requests. A proxy can reply to the intercepted request by sending a new request (possibly with modified arguments) to either the target object or to another object. Alternatively, a proxy can reply to the intercepted requests using local data (*e.g.,* cached replies).

Decision Makers. A *decision maker* assists proxies in replying to intercepted requests as depicted in Figure 5. A decision maker receives requests from a proxy and, similar to a rule-based interceptor, uses a set of rules to direct the operation on the intercepted requests. However, unlike a rule-based interceptor, a decision maker is not prohibited from replying to the requests.

Event Mediators. An event mediator is a CORBA object that decouples event generators from event listeners. We adopted this concept from the work by Bacon et al. [42]. An event mediator publishes a listener service, enabling registration of CORBA objects as event listeners. The event mediator is informed of events through a notification service. An event mediator forwards a copy of a new event to all listeners that have registered interest in this type of event.

B. Interaction among ACT Components

To describe the interactions among the ACT components, we provide a detailed sequence diagram [43] in Figure 6. The diagram shows the flow of a request/reply sequence in an ACT-ready application. The configuration shown in Figures 4 and 5 is used as the basis for this particular sequence diagram. Here, we consider only the activities on the client side and, for clarity, stubs and skeletons are not shown.

First, the request from the client to the servant is forwarded to the proxy (messages #1 to #11). After the request is received by the client ORB (#1), it is intercepted by the client generic interceptor (#2), where it is forwarded to the client rule-based interceptor (#3). The client rule-based interceptor checks its active rules. In this scenario, we assume it finds a rule that matches the request. The rule raises a `ForwardRequest` exception, which is passed to the client generic interceptor (#4) and then to the client ORB (#5), where a new request targeting the proxy is created (#6). Before the new request is sent to the proxy, it is intercepted again by the client generic and rule-based interceptors (#7 and #8), but this time no exception is raised (#9 and #10), and the calls simply return. The proxy receives the request (#11).

Next, the proxy processes the request and forwards it to the servant (messages #12 to #21). The proxy consults the decision maker (#12), where an event may be raised to handle an unknown situation (#13 and #14). The decision maker may adapt the client application by modifying the request parameters, sending new requests to other objects, or directing the proxy to reply to the request (*e.g.,* using cached replies). We assume that in this scenario, the decision maker modifies the request parameters and directs the proxy to send the modified request to the servant (#15) via the client ORB (#16). The modified request is also intercepted by the client generic and rule-based interceptors (#17 and #18) but again no exception is raised (#19 and #20). Therefore, the modified request is sent to the server ORB (#21).

The reverse sequence of actions occurs at the server application (not shown) and the reply to the modified request is returned to the client ORB (#22). The reply is intercepted by the client generic and rule-based interceptors (#23 and #24), where no exception is raised (#25 and #26). The reply is sent back to the proxy (#27), where it is forwarded to the decision maker (#28) for possible modifications and possible event raising (#29, #30, and #31).

Finally, using the reply from the servant and the direction given by the decision maker, the proxy replies to the client's request (#32). The reply is intercepted by the client generic and rule-based interceptors (#33 and #34). Again no exception is raised (#35 and #36), and the client ORB sends the reply back to the client (#37).

The extensive redirecting of messages in ACT raises the issue of performance overhead. We deem such overhead as necessary to provide flexibility and transparency. Moreover, our experimental results, described in Section IV, indicate that the overhead is actually quite small.

C. ACT Prototype

We have developed an ACT prototype in Java and tested it over ORBacus [14], a CORBA-compliant ORB distributed by IONA Technologies. ORBacus [14], like JacORB [44], TAO [8], and many other CORBA ORBs, supports CORBA portable interceptors [13], the only requirement for using ACT.

To make a CORBA application ACT-ready at the application startup time, we need to resolve the following bootstrapping issues. First, we need to register a generic interceptor with the application ORB. Like many other ORBs, ORBacus [14] uses

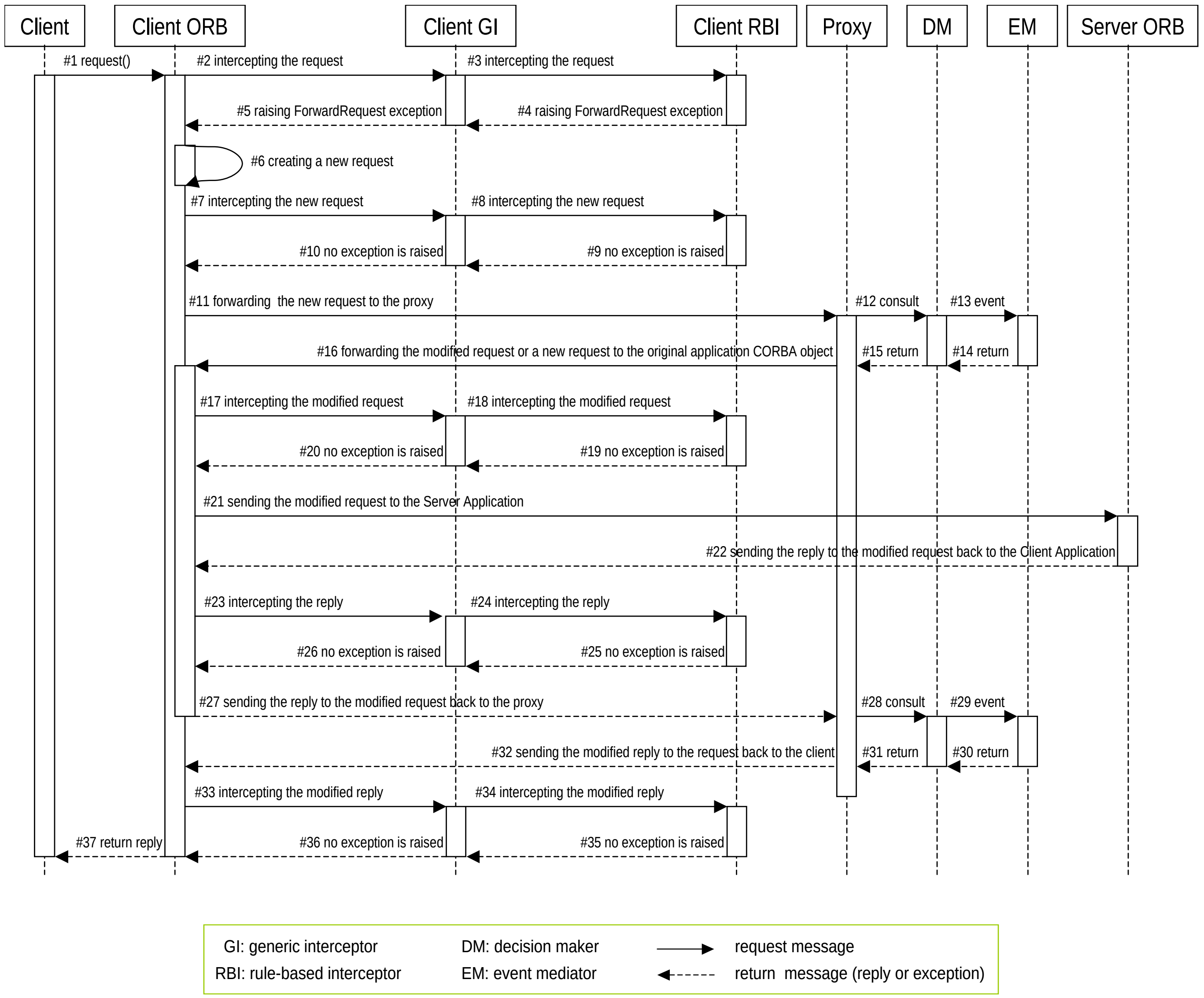


Fig. 6. Request/reply sequence in the client side of an ACT-ready application.

a configuration file that enables an administrator to register a CORBA portable interceptor with the application ORB. JacORB [44] and TAO [8] use a similar approach. Second, since the components in the ACT core are also CORBA objects, they require an ORB to support their operation (registration of services, and so on). Therefore, we need either to obtain a reference to the application ORB for this purpose, or to create a new ORB. ORBacus does provide such a reference, although the CORBA specification does not support this feature. To implement ACT over an ORB that does not provide such a reference, we simply create a new ORB, although its use introduces additional overhead.

To test the operation of our ACT prototype, we developed two administrative consoles: the Interceptor Registration Console and the Rule Management Console. *The Interceptor Registration Console* enables a user to manually register a dynamic interceptor. This console first obtains a generic interceptor name from the user and checks if the generic interceptor is registered with the CORBA naming service. Next, the user can register a dynamic interceptor with the generic interceptor. *The Rule Management Console* allows a user to manually insert rules into rule-based interceptors.

IV. CASE STUDY: COUPLING ACT AND QUO

To investigate the integration of ACT with an existing CORBA framework, we combined our ACT prototype with the Quality Objects (QuO) framework [4], developed by BBN Technologies and released under an open-source license. QuO is a powerful adaptive framework that supports dynamic adaptability in CORBA and Java RMI applications. ACT and QuO can work together in two major ways. First, ACT enables legacy CORBA applications to incorporate and benefit from QuO functionality, without modifying the source code of the application (indeed, even if the the source code is unavailable). Such a need may arise if the application is to be executed in an environment where conditions might be quite different than originally planned. Second, combining QuO and ACT enables weaving of adaptive code into distributed applications at both compile time and run time; we describe a specific example later in this section. We begin a brief overview of QuO, for completeness, followed by a discussion of how ACT and QuO interact and a description of an experiment in which they were combined to enhance an extant application.

A. QuO Background

QuO employs aspect-oriented programming [45] to separate the non-functional (systematic) aspects from the functional aspects of an application. Figure 7 illustrates a very simple QuO application. The client wrapper (or *delegate*) is the main point of contact between the client and the QuO core. The client wrapper is generated from a program written in the aspect-oriented structural description language (ASL) [15]. The QuO core comprises a contract and several system conditions. A *contract* is written in the contract-description language (CDL) [15] and defines acceptable regions of operation. *System conditions* can be considered as software "sensors" that record values representing the state of the execution environment. QuO combines the code for the QuO core and the code for wrapper into a package called a *qosket*. Using an aspect weaver called `quogen`, QuO weaves a qosket into an application at compile time.

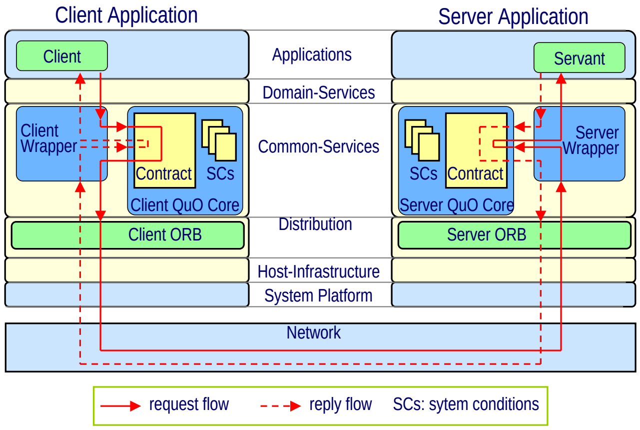


Fig. 7. A simplified depiction of the QuO architecture.

As shown in Figure 7, a request from the client is first received by the client wrapper. In a typical CORBA application, a client has a reference to a CORBA object stub. In QuO, however, the application developer explicitly creates the client wrapper, which wraps the stub (not shown). The client wrapper consults the contract in the client QuO core. The contract evaluates the current acceptable region of operation according to the details of the request and the status of the system as monitored by the system-condition objects. Once the current region of operation is identified, the actions specified in the contract are carried out. These actions might include returning a cached reply to the client, sending a request different than the original, forwarding the request with modified parameters, or redirecting the request to another CORBA object. If the reply is not generated locally, the request (or a modified request) is passed to the client ORB. The request is then sent to the server side of the application, where the reverse sequence of actions occurs. The reply generated by the servant, possibly modified by the server QuO core, will eventually reach the client ORB, where it is passed to the client wrapper. The client wrapper consults the client QuO core again for possible modifications and, finally, returns the reply to the client.

B. Dynamic Weaving of Qoskets Using ACT

Combining ACT with QuO enables transparent weaving of new qoskets into applications at run time. We identify three types of applications may benefit from such a capability. First, dependable applications are required to operate continuously without interruption; code for handling newly discovered faults can be added to these applications as they execute. Second, embedded applications are required to provide very small footprints; a minimal adaptive core can be compiled with the application, and optional adaptive code can be swapped in and out as needed during run time. Third, the source code for some legacy CORBA applications may be unavailable, or modifying the source code may be undesirable. Such applications can be adapted transparently using ACT and QuO, without modifying or even recompiling the application source code.

Figure 8 shows a request/reply sequence in a simple CORBA application using both QuO and ACT. The client and server generic interceptors are registered with the client and server ORBs, respectively, at startup time. To weave a new qosket into the application at run time, a new rule can be inserted in the client rule-based interceptor. The new rule can direct the rule-based interceptor to load the code for a proxy and a decision maker. The proxy in this case is simply a modified QuO wrapper, and the decision maker is exactly the contract defined in the new qosket. The rule then intercepts all incoming and outgoing requests/replies and forwards them to the proxy, where they are processed as if the qosket had been woven in to the application at compile time.

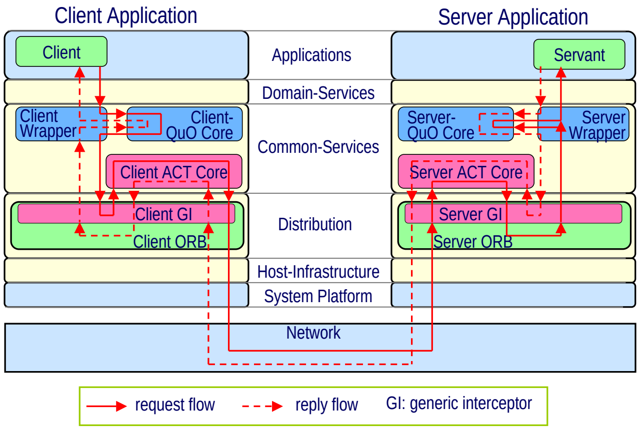


Fig. 8. Coupling ACT and QuO.

C. Example: Supporting Unanticipated Adaptation

To evaluate the performance and functionality of the hybrid ACT/QuO architecture described above, we used it to insert new adaptive functionality into an existing QuO application at run time. The application [16], a distributed image retrieval system, was developed by BBN Technologies and is distributed with the QuO framework. The application has two parts, a client that requests and displays images, and a server that stores the images and replies to requests for them. This application supports several different types of qoskets, which are woven into the application at startup time. A particular qosket called "UserAdapt" enables a user to modify the application interactively by directing it to retrieve different versions of the images. For example, selecting small instead of large versions of images can be used to reduce bandwidth consumption and delay.

First, we incorporated ACT into this application by introducing generic interceptors. To do so, we started the application with a command-line parameter directing it to an ORBacus configuration file defining how to load, create and register a generic interceptor with the application ORB. At this point the application is ACT-enabled. Figure 9 compares the round-trip delay for retrieving images of varying size, using both the original application and the ACT-enabled version. As shown, this overhead is negligible.

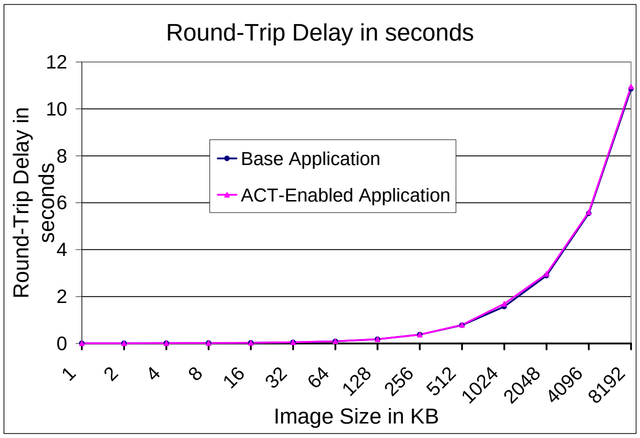


Fig. 9. Round-trip delay in ACT/QuO application.

Next, we developed a new qosket called UserAdaptFrameIntervel to weave to the application at run time using ACT. This qosket enables the user to interactively control the rate at which images are retrieved. Figure 10 and 11 show the code define the contract (in CDL) and the wrapper for the new qosket (in ADL), respectively. We defined three regions of operations Fast, Normal, and Slow in the contract, enabling the user to control the frame rate, for example, to conserve bandwidth. As illustrated in Figure 11, this control is accomplished by inserting appropriate delays. For the Fast region, we did not insert any delay, but for the Normal and Slow regions, we inserted 50 and 100 milliseconds frame-interval delay, respectively. We used the quogen utility to compile the new qosket.

To demonstrate the interaction between ACT and QuO, we ran an experiment that involves both static and dynamic weaving of qoskets into this application. The experiment is intended to represent run-time upgrading of a surveillance system (implemented using the image retrieval application) to add a new feature that controls the frame rate. Figure 12 shows a sample image from a camera in an instructional laboratory.

```
contract UserAdaptFrameRate ( syscond quo::ValueSC
  quo_sc::ValueSCImpl userFrameRate )
{
  region Fast (userFrameRate == 2) {}
  region Normal (userFrameRate == 1) {}
  region Slow (userFrameRate == 0) {}
};
```

Fig. 10. Code for the contract of the new qosket written in CDL.

```
behavior UserAdaptFrameRate ()
{
  void slide::SlideShow::read(in long gifNumber,
    out string size, out octetArray buf)
  {
      before METHODCALL
      {
       region Fast {}
       region Normal { ... Thread.sleep(50 ); ... }
       region Slow { ... Thread.sleep(100); ... }
      }
    }
  ...
  }
```

Fig. 11. Excerpted code for the wrapper of the new qosket written in ASL.

We executed the server on a desktop computer connected to a 100 Mbps wired network and the client on a laptop computer connected to an 11Mbps 802.11b wireless network; both systems are running the Linux operating system. At startup the "UserAdapt" qosket is woven into the application by specifying the wrapper class as a command-line parameter. Later, at run time, we used our Interceptor Registration Console to weave the "UserAdaptFrameRate" qosket into the application. Figure 13 shows screen dumps of the application as it displays large and small versions of an image, respectively.

Figure 14 shows a trace of the rate at which frames are displayed at the client application. During the experiment, a user modifies the application as follows. When application starts, large versions of frames (the default option) are retrieved from the server as fast as possible. The size of these images, combined with the limited bandwidth of the wireless network, produces a frame rate of approximately 2 images

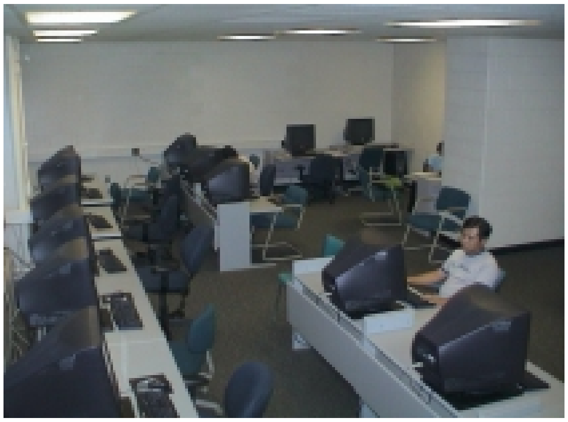
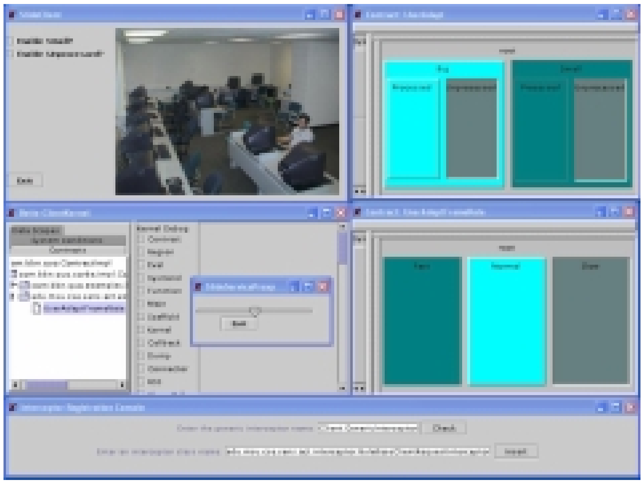
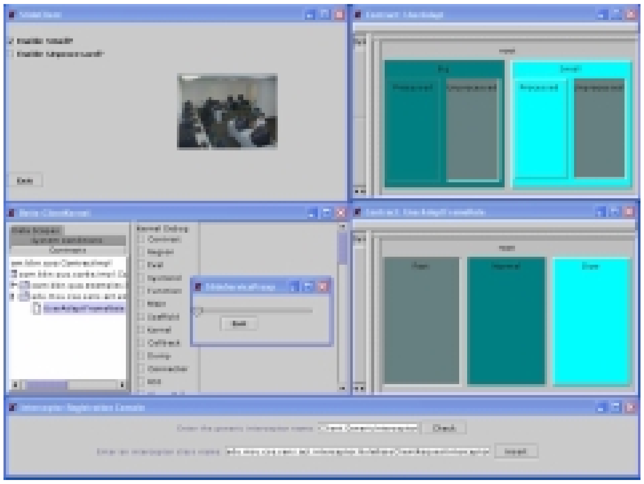


Fig. 12. Sample image of a monitored instructional laboratory.



(a)



(b)

Fig. 13. Screen captures of ACT/QuO image retrieval application: (a) 252 KB version of image displayed (b) 19 KB version of image displayed.

per second for the first 30 seconds of this experiment. At this point, the user selects the small-images option by way of the GUI in the "UserAdapt" qosket, thereby increasing the frame rate to approximately 14 images per second.

At 60 seconds into the experiment, the user dynamically weaves the UserAdaptFrameRate qosket into the application, using the interactive administration utilities described in Section III-C. Figure 14 shows a short, downward spike in the frame rate caused by the delay for weaving the new qosket. We consider such a one-time delay to be acceptable for this type of application. Immediately after the qosket is inserted, an interactive console is displayed by the qosket, enabling the user to choose from the three options (Fast, Normal, and Slow) interactively at run time. The Fast option is the default. At 90 seconds into the experiment, the user selects the Normal option; the additional 50 msec delay reduces the frame rate to approximately 7.5 images per second. At 120 seconds, the user chooses the Slow option (100 msec delay), which reduces the frame rate to approximately 5.5 images per second. At 150 seconds, the user chooses the Fast option again, which increases the frame rate to 14 images per second.

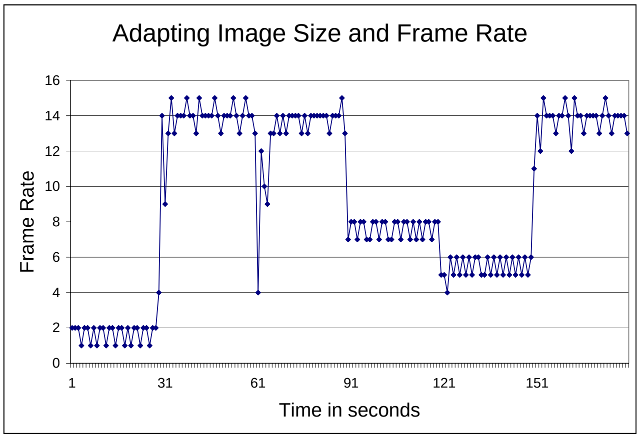


Fig. 14. Dynamic adaptation in a ACT/QuO hybrid application.

This experiment illustrates how ACT can be used to dynamically incorporate new behavior (in this case, a new QuO qosket) into a CORBA application at run time. The process is transparent to the application, in that we did not modify the application code or the QuO code. We simply started the application with generic interceptors registered with the application ORB.

V. CONCLUSION AND FUTURE WORK

In this paper, we proposed an adaptive CORBA template (ACT), which can be used to develop new adaptive CORBA frameworks and to enhance existing frameworks with unanticipated adaptive functionality and interoperability features. ACT can adapt legacy CORBA applications at run time without the need to modify or recompile their source code. The only requirement is that the application use a CORBA ORB that supports portable interceptors [13]. We developed an ACT prototype in Java and conducted a case study in which we integrated ACT with QuO. Our experiments show that the overhead introduced by ACT is negligible. We also showed that ACT can enable transparent integration of new adaptive code into extant QuO applications.

Further Information. A number of related papers and technical reports of the Software Engineering and Network Systems Laboratory can be found at the following URL: `http://www.cse.msu.edu/sens`.

Acknowledgements. The authors would like to thank John Zinky, Richard Schantz, Steve Vinoski, and Douglas Schmidt for their insightful early feedback on this work. We would also like to thank BBN Technologies and IONA Technologies for providing us with the source code for QuO and ORBacus, respectively. This work was supported in part by the U.S. Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744. This work was also supported in part by U.S. National Science Foundation grants CCR-9912407, EIA-0000433 and EIA-0130724.

REFERENCES

[1] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, and R. H. Campbell, "Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2000)*, (New York), April 2000.

[2] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas, "An architecture for next generation middleware," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, (The Lake District, England), September 1998.

[3] M. Astley, D. C. Sturman, and G. A. Agha, "Customizable middleware for modular distributed software," *Communications of the ACM*, vol. 44, no. 5, pp. 99–107, 2001.

[4] J. A. Zinky, D. E. Bakken, and R. E. Schantz, "Architectural support for quality of service for CORBA objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, 1997.

[5] M. Geier, M. Steckermeier, U. Becker, F. J. Hauck, E. Meier, and U. Rastofer, "Support for mobility and replication in the AspectIX architecture," Tech. Rep. TR-I4-98-05, Univ. of Erlangen-Nuernberg, IMMD IV, 1998.

[6] R. Koster, A. P. Black, J. Huang, J. Walpole, and C. Pu, "Thread transparency in information flow middleware," in *Proceedings of Middleware'01 (IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing)*, Springer Verlag, Nov. 2001.

[7] N. Venkatasubramanian, M. Deshpande, S. Mohapatra, S. Gutierrez-Nolasco, and J. Wickramasuriya, "Design and implementation of a composable reflective middleware," in *Proceedings of the 21th International Conference on Distributed Computing Systems (ICDCS-21)*, April 2001.

[8] D. C. Schmidt, D. L. Levine, and S. Mungee, "The design of the TAO real-time object request broker," *Computer Communications*, vol. 21, pp. 294–324, April 1998.

[9] R. Klefstad, D. C. Schmidt, and C. O'Ryan, "Towards highly configurable real-time object request brokers," in *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, April - May 2002.

[10] C. R. Becker and K. Geihs, "MAQS: Management for adaptive QoS-enabled services," in *Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, 1997.

[11] R. Baldoni, C. Marchetti, A. Termini, "Active software replication through a three-tier approach," in *Proceedings of the 22th IEEE International Symposium on Reliable Distributed Systems (SRDS02)*, (Osaka, Japan), pp. 109–118, October 2002.

[12] L. Moser, P. Melliar-Smith, P. Narasimhan, L. Tewksbury, and V. Kalogeraki, "The Eternal system: An architecture for enterprise applications," in *Proceedings of the Third International Enterprise Distributed Object Computing Conference (EDOC'99)*, July 1999.

[13] Object Management Group, Framingham, Massachusett, *The Common Object Request Broker: Architecture and Specification Version 3.0*, July 2003. Available at http://doc.ece.uci.edu/CORBA/formal/02-06-33.pdf.

[14] "ORBacus for C++ and Java version 4.1.0," 2001. http://www.iona.com/products/orbacus_home.htm.

[15] R. Schantz, J. Loyall, M. Atighetchi, and P. Pal, "Packaging quality of service control behaviors for reuse," in *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-time distributed Computing*, (Washington, DC), April 2002.

[16] J. Zinky, J. Loyall, and R. Shapiro, "Runtime performance modeling and measurement of adaptive distributed object applications," in *Proceedings of International Symposium on Distributed Object and Applications (DOA 2002)*, (Irvine, California), October 2002.

[17] D. C. Schmidt, "Middleware for real-time and embedded systems," *Communications of the ACM*, vol. 45, June 2002.

[18] D. C. Schmidt, "The ADAPTIVE Communication Environment: An object-oriented network programming toolkit for developing communication software," *Concurrency: Practice and Experience*, vol. 5, no. 4, pp. 269–286, 1993.

[19] V. C. Zandy and B. P. Miller, "Reliable network connections," in *ACM MobiCom 2002*, (Gorgia), September 2002.

[20] S. M. Sadjadi, P. K. McKinley, and E. P. Kasten, "Architecture and operation of an adaptable communication substrate," in *the Ninth International Workshop on Future Trends of Distributed Computing Systems (FTDCS '03)*, May 2003.

[21] Object Management Group, Framingham, Massachusett, *The Common Object Request Broker: Architecture and Specification Revision 2.2*, Feb. 1998.

[22] Microsoft Corporation, *Microsoft COM Technologies - DCOM*, 2000. http://www.microsoft.com/com/dcom.asp.

[23] Java Soft, *Java Remote Method Invocation Specification, revision 1.5, JDK 1.2*, Oct. 1998.

[24] N. Wang, D. C. Schmidt, and M. Kircher, "Towards an adaptive and reflective middleware framework for QoS-enabled CORBA component model applications," *IEEE Distributed System Online, Special Issue on Reflective Middleware*, 2003.

[25] M. Roman, F. Kon, and R. H. Campbell, "Reflective middleware: From your desk to your hand," *IEEE Distributed Systems Online*, vol. 2, no. 5, 2001.

[26] W. Cazzola and M. Ancona, "mChaRM: A reflective middleware for communications-based reflection," Tech. Rep. DISI-TR-00-09, Universita degli Studi di Milano, May 2000.

[27] S. Maffeis, "Adding group communication and fault-tolerance to CORBA," in *Proceedings of the Conference on Object-Oriented Technologies*, pp. 135–146, 1995.

[28] R. Hayton, *FlexiNet Open ORB Framework*. APM Ltd., Oct. 1997.

[29] J. C. Fabre and T. Perennou, "A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach," *IEEE Transactions on Computers*, vol. 47, no. 1, pp. 78–95, 1998.

[30] O. Othman, "The design, optimization, and performance of an adaptive middleware load balancing service," Master's thesis, University of California, Irvine, 2002.

[31] D. Sharp, "Reducing avionics software cost through component-based product line development," in *The Software Technology Conference*, (Salt Lake City, Utah), April 1998.

[32] M. van Steen, P. Homburg, and A. S. Tanenbaum, "The architectural design of Globe: A wide-area distributed system," Tech. Rep. 422, Vrije Universiteit, Amsterdam, The Netherlands, March 1997.

[33] N. Wang, D. C. Schmidt, O. Othman, and K. Parameswaran, "Evaluating meta-programming mechanisms for ORB middleware," *IEEE Communications Magazine, Special Issue on Evolving Communications Software: Techniques and Technologies*, October 2000.

[34] Z. Yang, B. Cheng, R. Stirewalt, J. Sowell, S. M. Sadjadi, and P. K. McKinley, "An aspect-oriented approach to dynamic adaptation," in *Proceedings of the ACM SIGSOFT Workshop On Self-healing Software*, Nov. 2002.

[35] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," *Lecture Notes in Computer Science*, vol. 2072, pp. 327–355, 2001.

[36] F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The case for reflective middleware," *Communications of the ACM*, vol. 45, pp. 33–38, June 2002.

[37] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture*, vol. 2. John Wiley, 2001.

[38] G. Blair, G. Coulson, and N. Davies, "Adaptive middleware for mobile multimedia applications," in *Proceedings of the Eighth International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pp. 259–273, 1997.

[39] A. Corsaro, D. Schmidt, R. Klefstad, and C. O'Ryan, "Virtual component a design pattern for memory constrained embedded applications," in *Proceedings of the Ninth Conference on Pattern Language of Programs (PLoP 2002)*, 2002.

[40] R. Friedman and E. Hadad, "Client side enhancements using portable interceptors," in *Proceedings of the Sixth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS'01)*, January 2001.

[41] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulis, and T. P. Archambault, "The Totem system," in *Proceedings of the 25th International Symposium on Fault Tolerant Computing*, (Pasadena, California), pp. 61–66, 1995.

[42] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri, "Generic support for distributed applications," *IEEE Computer*, vol. 33, no. 3, pp. 68–76, 2000.

[43] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[44] G. Brose and N. Noffke, "JacORB 1.4 documentation," tech. rep., Freie Universitt Berlin and Xtradyne Technologies AG, August 2002.

[45] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlag LNCS 1241, June 1997.