Monitoring, Security, and Dynamic Configuration with the *dynamicTAO* Reflective ORB[⋆]

Fabio Kon[⋆⋆], Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
{f-kon,mroman1,pingliu,jinamao,yamane,magalhae,rhc}@cs.uiuc.edu
http://choices.cs.uiuc.edu/2K/dynamicTAO

Abstract. Conventional middleware systems fail to address important issues related to dynamism. Modern computer systems have to deal not only with heterogeneity in the underlying hardware and software platforms but also with highly dynamic environments. Mobile and distributed applications are greatly affected by dynamic c hangesof the environment c haracteristicssuch as security constraints and resource availability. Existing middleware is not prepared to react to these c hanges.
In many cases, application developers know when adaptive c hanges in communication and security strategies would improve system performance. But often, they are not able to benefit from it because the middleware lacks the mechanisms to support monitoring (to detect when adaptation should take place) and on-the-fly reconfiguration.
dynamicTAO is a CORBA-compliant reflective ORB that supports dynamic configuration. It maintains an explicit representation of its own internal structure and uses it to carry out runtime customization safely. After describing *dynamicTAO*'s design and implementation, we discuss our experience on the development of t wo systems benefiting from the reflective nature of our ORB: a flexible monitoring system for distributed objects and a mechanism for enforcing access control based on dynamic security policies.

There is nothing permanent except change.
Heraclitus of Ephesus (535-475 BC)

1 Introduction

One of the major motivations for the development of middleware is the high degree of hardware and software heterogeneity encountered in existing systems. Middleware systems like CORBA are able to hide the specifics of the underlying platform and provide a uniform high-level interface for application developers.

[⋆] This research is supported by NSF grants 98-70736 and 99-70139.
[⋆⋆] Fabio Kon is supported in part by a grant from CAPES-Brazil, proc.#1405/95-2.

However, the diversity in modern computer systems is not limited to differences in the underlying hardware and operating system. One must not forget that even machines with the same hardware type and operating system may be configured with extremely different resources (e.g., Ethernet versus ATM networking, different amounts of RAM and disk space) and with different software packages.

Besides this "diversity in space", we also find a huge "diversity in time", i.e., a single machine typically experience drastic variations in CPU, memory, disk, and network availability. Mobile computers experience changes in connectivity, bandwidth, and error patterns as they move from one area to another. Laptops are subject to different security policies as they are connected to different domains.

Existing middleware systems are not ready to deal with these two kinds of diversity. They are usually optimized to a particular architecture and to a particular configuration. But, computing environments are getting increasingly dynamic; if the next generation middleware is not capable of managing the dynamic variations in the environment properly, a large amount of computing resources will be wasted and application performance will be greatly affected.

In order to cope with these variations and still maintain a good performance level, middleware and application components must be able to detect changes in the environment and reconfigure themselves to optimize their performance under the new conditions. We addressed this problem by adding support for reconfiguration and runtime extensibility within TAO, an open source CORBA Object Request Broker (ORB).

2 *dynamicTAO*

In order to deal with the highly dynamic environments described in the previous section, our group is developing the *2K* distributed operating system [16], which is based on a dynamically configurable middleware layer compatible with CORBA. Rather than implementing a new ORB from scratch, we realized that it would be more productive to modify an existing ORB to add the dynamism we needed.

After carefully studying existing ORBs, we came to the conclusion that the TAO ORB [29] would be the best starting point for developing our infrastructure. TAO is a portable, flexible, extensible, and configurable ORB based on object-oriented design patterns. It is written in C++ and uses the *Strategy* design pattern [6] to separate different aspects of the ORB internal engine. A configuration file is used to specify the strategies the ORB uses to implement aspects like concurrency, request demultiplexing, scheduling, and connection management. At ORB startup time, the configuration file is parsed and the selected strategies are loaded.

TAO is primarily targeted for static hard real-time applications such as Avionics systems. Thus, it assumes that, once the ORB is initially configured, its strategies will remain in place until it completes its execution. There is very little support for on-the-fly reconfiguration.

The *2K* project, on the other hand, seeks to build a flexible infrastructure to support adaptive applications running on dynamic environments. On-the-fly adaptation is extremely important for a wide range of applications including the ones dealing with multimedia, mobile computers, multiple security domains, and other kinds of dynamically changing environments. We achieved the desired level of configurability with *dynamicTAO*, our extension of TAO that enables on-the-fly reconfiguration of its strategies.

2.1 A Reflective ORB

dynamicTAO is our first complete implementation of a CORBA reflective ORB. As pointed out in [31, 32], a *reflective* system is a system that gives a program access to its definition and evaluation rules, and defines an interface for altering them. In an ORB, client requests represent the "program" to be evaluated by the system. The ORB implementation represents the "evaluator", and "evaluation" is simply remote method invocation. A reflective ORB makes it possible to redefine its evaluation semantics.

dynamicTAO is a reflective ORB because it allows inspection and reconfiguration of its internal engine. It achieves that by exporting an interface for (1) transferring components across the distributed system, (2) loading and unloading modules into the ORB runtime, and (3) inspecting and modifying the ORB configuration state. The infrastructure can also be used for dynamic reconfiguration of servants running on top of the ORB and even for reconfiguring non-CORBA applications.

Reification in *dynamicTAO* is achieved through a collection of entities known as *component configurators* [12, 13]. A component configurator holds the dependencies between a certain component and other system components. Each process running the *dynamicTAO* ORB contains a component configurator instance called `DomainConfigurator`. It is responsible for maintaining references to instances of the ORB and to servants running in that process. In addition, each instance of the ORB contains a customized component configurator called `TAOConfigurator`.

`TAOConfigurator` contains hooks to which implementations of *dynamicTAO* strategies are attached. Hooks work as "mounting points" where specific strategy implementations are made available to the ORB. We currently support hooks for different kinds of strategies such as Concurrency, Security, Monitoring, and the like. The association between hooks and component implementations can be changed at any time, subject to safety constraints.

Figure 1 illustrates this reification mechanism in a process containing a single instance of the ORB. If necessary, individual strategies can use component configurators to store their dependencies upon ORB instances and other strategies. These configurators may also store references to client connections that depend on the strategies. With this information, it is possible to manage strategy reconfiguration consistently as we explain in section 2.3.

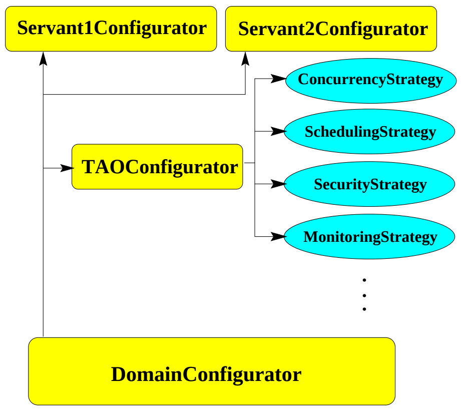


Fig. 1. Reifying the *dynamicTAO* structure

Component implementations are shipped as dynamically loadable libraries, so they can be linked to the ORB process at runtime. They are organized in *categories* representing different aspects of the ORB internal engine (which are associated with *dynamicTAO* hooks) or different types of servant components. In future implementations, we intend to support category type-checking using ANSI C++ runtime type information (RTTI).

The *dynamicTAO* architectural framework is depicted in figure 2. The *Persistent Repository* stores category implementations in the local file system. It offers methods for manipulating (e.g. browsing, creating, deleting) categories and the implementations of each category. Once a component implementation is stored in the local repository, it can be dynamically loaded into the process runtime.

A *Network Broker* receives reconfiguration requests from the network and forwards them to the *Dynamic Service Configurator*. The latter contains the `DomainConfigurator` (shown in figure 1) and supplies common operations for dynamic configuration of components at runtime. It delegates some of its functions to specific component configurators (e.g., *TAOConfigurator* or a certain *ServantConfigurator*).

We minimized the changes to the standard ACE/TAO distribution by delegating some of the basic configuration tasks to components of the ACE framework such as the `ACE_Service_Config` (used to process startup configuration files and manage dynamic linking) and the `ACE_Service_Repository` (to manage loaded implementations) [9].

This architectural framework enables the development of different kinds of persistent repositories and network brokers to interact with the Dynamic Service Configurator. Thus, it is possible to use different naming schemes when storing category implementations and different communication protocols for remote configuration as described below.

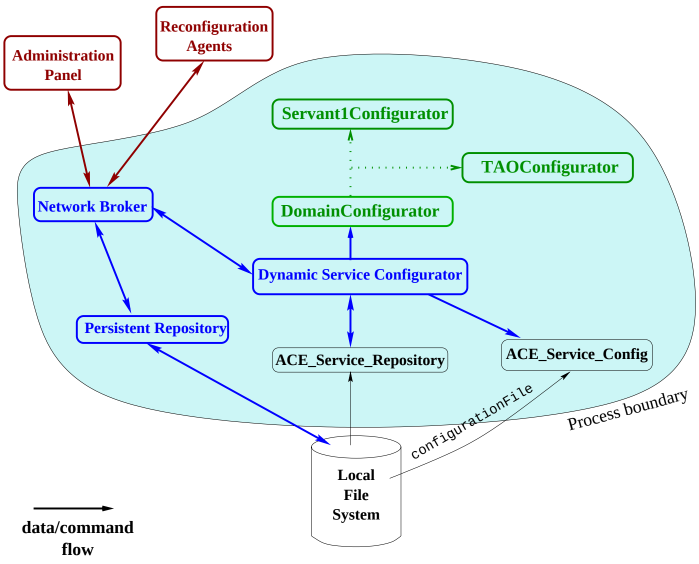


Fig. 2. *dynamicTAO* Components

We built the *dynamicTAO* components using the ACE wrappers [5] for operating system services. Thus, *dynamicTAO* runs on the several different platforms to which ACE was ported.

2.2 Reconfiguration Interface

dynamicTAO supports three distinct forms of reconfiguration interfaces. In general terms, they all provide the same functionality but each of them has characteristics that makes it more or less appropriate for certain situations. A description of the interfaces follows.

1. The **DCP Broker** is a customized subclass of the *Network Broker* shown in Figure 2. It listens on a TCP port, waiting for connection requests from remote clients. Once a connection is established, a client can send inspection and reconfiguration commands using DCP, our Distributed Configuration Protocol [11]. This interface is particularly good for debugging and for fast interaction with an ORB since the user can access the configuration interface simply by establishing a telnet connection to the DCP Broker.
2. The **Reconfiguration Agent Broker** is also a customized subclass of the *Network Broker*, it is useful for configuring a distributed collection of ORBs as we describe in section 2.4.
3. The **DynamicConfigurator** is a CORBA object that exports an IDL interface with operations equivalent to the ones offered by the DCP protocol. It is the most convenient of the three interfaces for programmatic interactions since all the communication aspects are hidden by the CORBA middleware.

We now use the DynamicConfigurator IDL specification presented in figure 3 to explain the functionality of the *dynamicTAO* reconfiguration interfaces[1].

```
interface DynamicConfigurator
{
 typedef sequence<string> stringList;
 typedef sequence<octet> implCode;

 stringList list_categories ();
 stringList list_implementations (in string categoryName);
 stringList list_loaded_implementations ()
 stringList list_domain_components ();
 stringList list_hooks (in string componentName);
 string     get_impl_info      (in string implName);
 string     get_comp_info      (in string componentName);
 string     get_hooked_comp    (in string componentName,
                                in string hookName);
 string     get_latest_version (in string categoryName);

 long load_implementation (in  string categoryName,
                           in  string impName,
                           in  string params
                           in  Configuration::Factory factory,
                           out Configuration::ComponentConfigurator cc);
 void hook_implementation (in string loadedImpName,
                           in string componentName,
                           in string hookName);

 void suspend_implementation   (in string loadedImpName);
 void resume_implementation    (in string loadedImpName);
 void remove_implementation    (in string loadedImpName);
 void configure_implementation (in string loadedImpName,
                                in string message);

 void upload_implementation   (in string categoryName,
                               in string impName,
                               in implCode binCode);
 void download_implementation (in string categoryName,
                               inout string impName,
                               out implCode binCode);
 void delete_implementation   (in string categoryName,
                               in string impName);
};
```

Fig. 3. The *DynamicConfigurator* interface

The *DynamicConfigurator* interface specifies the operations that can be performed on *dynamicTAO* abstractions, namely, categories, implementations, hooks, and configurable components. The first nine operations in the interface are used to inspect the dynamic structure of that domain and retrieve information about the different abstractions. A *category* represents the type of a component; each category typically contains different *implementations*, i.e., dynamically loadable code stored in the Persistent Implementation Repository. For example, a category called `Concurrency` contains the three threading models

[1] To make figure 3 more clear, we omitted the exceptions that each operation can raise.

that *dynamicTAO* currently supports: `Reactive_Strategy`, `Thread_Strategy`, and `Thread_Pool_Strategy`.

Once an implementation is loaded into the system runtime, it becomes a *loaded implementation* and can be associated with a logical *component* in the ORB domain. Finally, components have *hooks* that are used to represent inter-component dependence; if a component A depends upon component B then this dependence is represented by attaching B to a hook in A.

`load_implementation` dynamically loads and starts an implementation from the persistent repository. `hook_implementation` attaches it to a hook in one of the components in the domain.

The next four methods allow operations on loaded implementations. It is possible to suspend and resume their main threads, remove them from the process, and send them component-specific reconfiguration messages.

`upload_implementation` allows an external entity to send an implementation to be stored in the local Persistent Repository, so that it can be linked to a running process and attached to a hook. Conversely, `download_implementation` allows a remote entity to retrieve an implementation from the local Persistent Repository. Finally, `delete_implementation` is used to delete implementations stored at the ORB Persistent Repository.

Consider now the scenario in which a user wants to change the threading model at runtime by using an implementation of the Concurrency strategy called *Thread_Pool_Strategy.* Assuming that the user wants to start with a thread pool of size 20, the required configuration steps are the following.

1. Load the implementation into memory:
   ```
   version = load_implementation("Concurrency","Thread_Pool_Strategy","20",
   0, cc)
   ```
2. Attach the implementation to the *Concurrency* hook in TAO:
   ```
   hook_implementation("Concurrency":version,"TAO","Concurrency_Strategy")
   ```

After the new implementation is attached, the ORB starts using it. In section 2.3, we discuss what happens if a different concurrency strategy is in use.

Figure 4 shows C++ code that uses the Dynamic Configurator to retrieve and print some information about the ORB internal configuration. The code obtains a reference to the *DynamicConfigurator* object through the ORB's *resolve_initial_references()* method.

To facilitate interactive configuration, we developed *Doctor*, a Dynamic ORB Configuration Tool. As shown in figure 5, *Doctor* is a Java graphical user interface that lets users manipulate both the ORB persistent repository and the runtime configuration interactively. The tool establishes a connection to the ORB DCP Broker and let users send DCP messages by using the mouse.

2.3 Consistency

Reconfiguring a running ORB while it is servicing client requests is a difficult task that requires careful consideration. There are two major classes of problems.

```
CORBA::Object_var          dcObj;
DynamicConfigurator_var    dynConf;
CORBA::ORB_var             orb;

orb     = CORBA::ORB_init (argc, argv);
dcObj   = orb->resolve_initial_references ("DynamicConfigurator");
dynConf = DynamicConfigurator::_narrow (dcObj.in ());

stringList *list = dynConf->list_implementations ("Concurrency");

printf ("Available concurrency strategies:");
printStringList (list);

char *ret = dynConf->get_hooked_comp ("TAO", "Concurrency_Strategy");

printf ("Now, using the <%s> concurrency strategy.", ret);
```

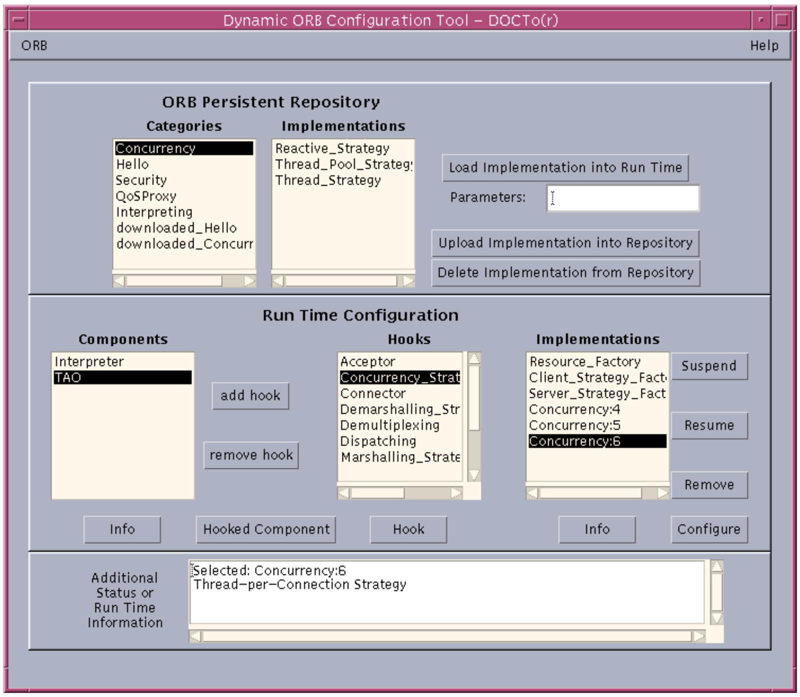
Fig. 4. Inspecting the ORB internal state

Consider the case in which *dynamicTAO* receives a request for replacing one of its strategies (S_{old}) by a new strategy (S_{new}). The first problem is that TAO strategies are implemented as C++ objects that communicate through method invocations; thus, before unloading S_{old}, the system must be sure that no one is running S_{old} code and that no one is expecting to run S_{old} code in the future. Otherwise, the system could crash. Thus, it is important to assure that S_{old} is only unloaded after the system can guarantee that its code will not be called.

The second problem is that some strategies need to keep state information. When a strategy S_{old} is being replaced by S_{new}, part of S_{old}'s internal state may need to be transfered to S_{new}. Both problems can be addressed with the help of the *TAOConfigurator*.

Consider, for example, the three concurrency strategies supported by *dynamicTAO*: single-threaded reactive, thread-per-connection, and thread-pool. If the user switches from the reactive or thread-per-connection strategies to any other concurrency strategy, nothing special needs to be done. *dynamicTAO* may simply load the new strategy, update the proper *TAOConfigurator* hook, unload the old strategy, and continue. Old client connections will complete with the concurrency policy dictated by the old strategy. New connections will utilize the new policy.

However, if one switches from the thread-pool strategy to another one, we must take special care. The thread-pool strategy we developed maintains a pool of threads that is created when the strategy is initialized. The threads are shared by all incoming connections to achieve a good level of concurrency without having the runtime overhead of creating new threads. A problem arises when one switches from this strategy to another strategy: the code of the strategy being replaced cannot be immediately unloaded. This happens because, since the threads are reused, they return to the thread-pool strategy code each time a connection finishes. This problem can be solved by a *ThreadPoolConfigurator* keeping information about which threads are handling client connections and destroying them



Selected: Concurrency:6
Thread–per–Connection Strategy

Fig. 5. The *Doctor* configuration tool

as the connections are closed. When the last thread is destroyed the thread-pool strategy signals that it can be unloaded.

Another problem occurs when one replaces the thread-pool strategy by a new one. There may be several incoming connections queued in the strategy waiting for a thread to execute them. The solution is to use the *Memento* pattern [6] to encapsulate the old strategy state in an object that is passed to the new strategy. An object is used to encapsulate the queue of waiting connections. The system simply passes this object to the new strategy which then takes care of the queued connections.

2.4 Reconfiguration Agents

After implementing the first version of *dynamicTAO* we noticed that a significant limitation it presented was that, in order to configure a particular ORB, it required a point-to-point connection between the administration node (e.g. running *Doctor*) and the ORB process. Thus, if a system administrator needed to upgrade a certain component of an on-line service composed of ten replicas located in different countries, it was necessary to connect to each replica separately, upload the new implementation of the component, and reconfigure the replica. This process was extremely laborious and tiresome.

Our group had experience with the deployment of a large-scale Multimedia Distribution System to broadcast live video and audio through a network of more than 30 multimedia servers spread across five continents. The system ran 24 hours per day for more than three months and delivered multimedia streams to more than one million users in dozens of different countries [14]. The difficulty in carrying out that experiment (managing more than 30 application nodes in a wide-area network) exposed the extreme necessity of flexible mechanisms for efficient runtime reconfiguration of long-running, large-scale systems. We believe that this kind of application will become increasingly important and numerous on the Internet in the next decade. Thus, a good infrastructure to support them would be extremely useful.

As a first solution to the problem we considered implementing a management front-end that would allow administrators to type sequences of DCP commands that would be sent to a list of ORBs. Although this approach would simplify the work of the administrator, it would not solve the problem of bandwidth waste, i.e., sending large amounts of duplicated information across long-distance Internet lines.

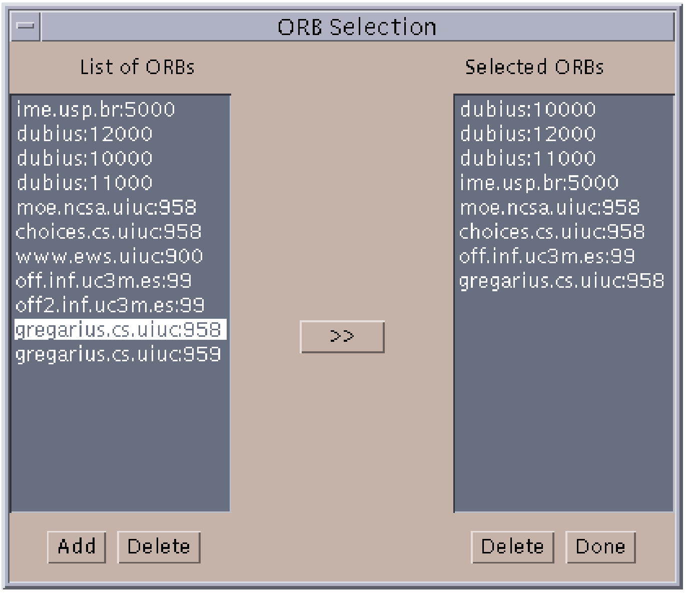
The solution we adopted was to allow administrators to organize the nodes of their Internet systems in a hierarchical manner for reconfiguration purposes. The administrator specifies the topology of the distributed application as a directed graph and creates a mobile *reconfiguration agent* which is injected into the network. The reconfiguration agent then visits the nodes of this graph of interconnected ORBs. In each ORB, the agents are received by the Reconfiguration Agent Broker. The broker first replicates and forwards the agent to neighboring nodes, then processes the DCP commands locally, and finally, collects the reconfiguration results, sending them back to the neighboring agent source.

Using this approach, the administrator can organize the reconfiguration hierarchy to optimize the data flow between distant application nodes. The reconfiguration commands are executed in parallel in the various nodes, improving response time. If desired, the graph may contain different levels of redundancy so that the system can tolerate the failure of some of the nodes in the reconfiguration network.

Administrators use a Java graphical administrative front-end for specifying reconfiguration graphs and for assembling and sending reconfiguration agents. Given the large variations on Internet line speeds, administrators should have an approximate idea of the available bandwidth in each edge of the reconfiguration graph. With this information it is possible to organize the graph to minimize the transmission over low-bandwidth and congested Internet lines.

The administrator selects those ORBs that will be part of the reconfiguration graph (see figure 6) and draws directed edges connecting the graph nodes (see figure 7). Each time a new ORB is selected from the list on the left-hand side of figure 6, a new node is added to the graph in figure 7.

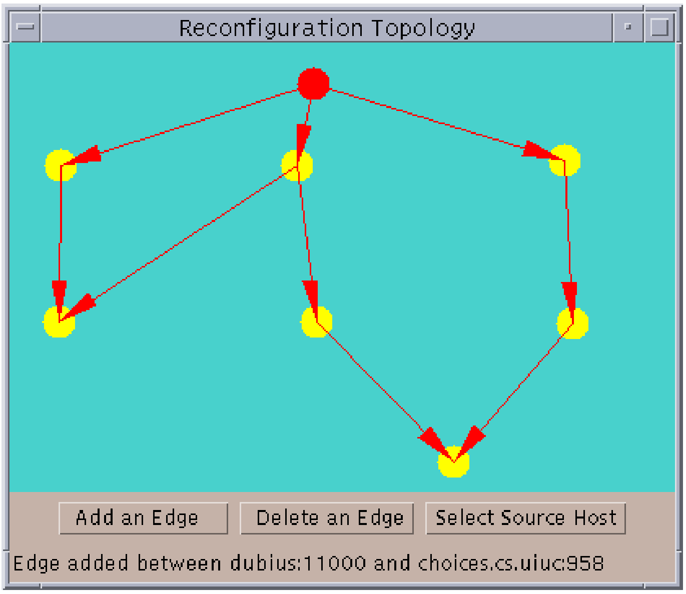
Once the reconfiguration graph is defined, a new window assists the administrator to build a list of DCP commands that are codified into a reconfiguration agent. Finally, the administrator instructs the graphical front-end to send the



ime.usp.br:5000
dubius:12000
dubius:10000
dubius:11000
moe.ncsa.uiuc:958
choices.cs.uiuc:958
www.ews.uiuc:900
off.inf.uc3m.es:99
off2.inf.uc3m.es:99
gregarius.cs.uiuc:958
gregarius.cs.uiuc:959

dubius:10000
dubius:12000
dubius:11000
ime.usp.br:5000
moe.ncsa.uiuc:958
choices.cs.uiuc:958
off.inf.uc3m.es:99
gregarius.cs.uiuc:958

Fig. 6. Selecting the nodes of the reconfiguration graph



Edge added between dubius:11000 and choices.cs.uiuc:958

Fig. 7. Defining the reconfiguration topology

agent to an initial node in the graph. Figure 8 shows the composition of an agent with three DCP commands: `list_categories`, `list_loaded_implementations`, and `list_implementations`.

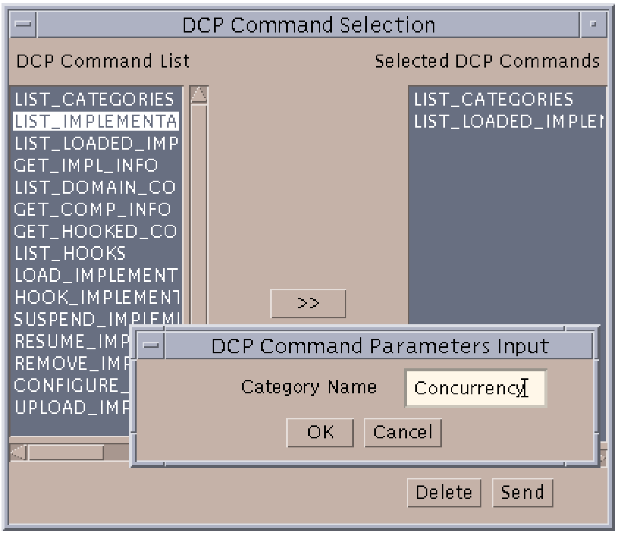


Fig. 8. Composing a reconfiguration agent

Securing Dynamic Configuration The initial implementation of *dynamicTAO* did not provide security either in the DCP Broker or in the Reconfiguration Agent Broker. In other words, if these interfaces were enabled, any user could contact one of the brokers and inject inspection and reconfiguration agents freely. In order to solve this problem, we implemented a flexible security architecture described in section 4. It relies on a Reference Monitor that allows for very fine-grain control over the access to the *DynamicConfigurator* operations.

We are also working on a security mechanism for both the DCP and the Reconfiguration Agent Brokers. We have a preliminary prototype supporting encryption, authentication, and access control.

More details about the implementation and the issues related to reconfiguration agents can be found in [15]. In the following sections we describe how the *dynamicTAO* infrastructure was used to implement Monitoring and Security services.

3 Monitoring Object Interactions

To support the construction of effective adaptable applications and systems, the middleware must provide a way to detect when adaptation should take place. In the previous section, we showed **how** *dynamicTAO* could be used to adapt an application. In this section, we show how an application can know **when** it is time to adapt.

We built the *2K* Monitoring Service [21] as a dynamically loadable component that can be attached to and detached from *dynamicTAO* at any time by using the configuration interfaces described in the previous section. It is able to collect and consolidate information about the interactions (i.e., method invocations) among CORBA objects in the distributed system. By using the Monitoring Service in conjunction with the *2K* Resource Manager (which provides dynamic information about hardware resource availability), a program can be completely aware of the dynamics of the environment in which it is inserted.

By knowing the nature and magnitude of the interactions between components, a system can reconfigure itself in order to adapt to different situations and improve its performance. Moreover, if the information about component interaction is exported to applications, they become capable of implementing their own adaptation policies. Finally, exporting this information to system administrators and users in a way that they can easily understand, might help them to identify bottlenecks in their system. For example, by showing that applications spend most of their time waiting to access the local file system might indicate that the administrator should install a faster hard disk or that the system should adopt a more effective caching policy.

We developed this service following having two major goals in mind: minimum performance degradation and minimum interference. First, the Monitoring Service should not slow down any part of the system significantly. Second, it should not change the dependency relations among other system and application components. That means that when the service is deactivated, it should be as if the service did not exist. And when the service is activated, the system and application components should not be aware of the service unless it needs to use it.

3.1 Architecture

The Monitoring Service uses the reflective ORB *DynamicConfigurator* interface for dynamically loading (and unloading) its modules. Once the service is loaded, it is inserted into the invocation path by using a request-level interceptor[2]. Unloading it from memory or suspending its execution temporarily causes its removal from the invocation path. When the service is not active, the overhead for the interceptor is negligible (simply checking the nullity of a pointer).

Our architectural framework, shown in Figure 9, is composed of the *Monitoring Interceptor*, which collects information about selected client requests and one or more *Storage Servers*, which are responsible for saving the data into a persistent store and for processing queries about the stored data. In addition, the *dynamicTAO* *DynamicConfigurator* is used to dynamically configure the interceptor behavior. The Monitoring Service user depicted in Figure 9 is either

[2] The interceptor mechanism is defined in the CORBA specification, chapter 18 [22].

a computer program responsible for detecting special conditions in the environment or a programmer or system administrator using a text-based or graphical front-end.

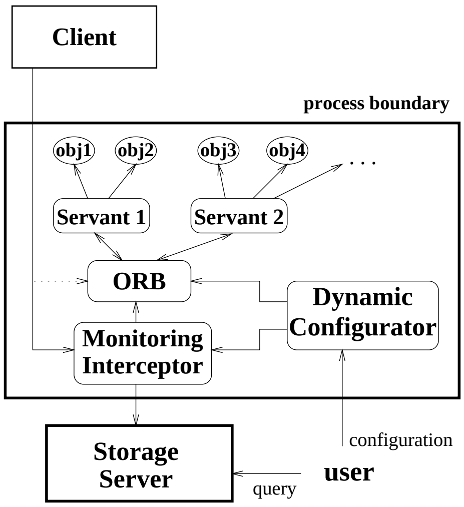


Fig. 9. The Monitoring Service Architecture

Upon initialization, the interceptor contacts the Name Server to locate a Storage Server in the network. Users can then configure the monitoring process through the *configure* method of the *dynamicTAO DynamicConfigurator* interface. It is possible to specify (1) the name of the objects that should be monitored, i.e., which objects should have their requests information sent to the Storage Server, (2) which operations of each object should be monitored, and (3) some interceptor internal parameters such as how often it sends the collected information to the Storage Server.

Every time the interceptor detects a client request that should be sent for storage, it creates a record containing five kinds of information about that request: client machine address, target object name, target operation name, timestamp, and server-side duration. The records are grouped in a local buffer and a different thread sends them to the Storage Server periodically.

The Storage Server stores its data either in a file system or in a database management system and exports two interfaces. The first is used by the monitoring interceptor to publish the collected information and the second is used by users to send queries about the collected information. The query interface provides support for a wide range of query types. Users can ask, for example, "When was the last call to operation A on object X?", "What is the average completion time for calls to operation A on object X from host P?", "How many times did object X receive requests between time t_0 and t_1?", and so on.

3.2 Performance Measurements

We measured the latency on calls to a CORBA object in three different stages. In stage one, we measured the latency on each of the method calls without using the Monitoring Service. In stage two, we measured the latency after the Monitoring Service was loaded and attached to the interceptor, but without having it monitoring this particular object. In the last stage, we measured the latency when the object is being monitored by the Monitoring Service. For each stage, we tested both local and remote method calls. Table 1 shows the average of the results for 50 experiments ran between a Sun Ultra2 and Ultra60 machines running Solaris 2.6 and connected by fast Ethernet. Each experiment consists of measuring the round-trip time for a call on a *getHello()* operation that simply returns a 12-character CORBA string to the client.

getHello calls	Without Monitoring	Monitoring Disabled	Monitoring Enabled
Local	0.781	0.803	0.941
Remote	1.252	1.277	1.379

Table 1. Monitoring Service overhead (in *ms*)

As can be seen from Table 1, the overhead of the Monitoring Service on an object that was being monitored was 20% and 10% for local and remote calls, respectively. The overhead was reduced to 2.8% and 2.0% when the Monitoring Service was active but not monitoring that particular object.

We are certain that there are still opportunities for optimizations that would make the overhead smaller. However, it is important to notice that the *getHello* operation is almost the worst case scenario because it has no parameters and its returned value is very small. In common cases, CORBA operations carry a large number of arguments that must be marshalled and demarshalled. In those cases, the relative overhead of the Monitoring Service would be much smaller.

4 Dynamic Security

The second service we implemented on top of the *dynamicTAO* infrastructure was the Reference Monitor [17], a flexible mechanism for enforcing access control based on dynamic security policies. This work consisted on deploying the Cherubim security framework [2] in the *dynamicTAO* environment and adding support for audit logging and caching of security decisions.

As we described in section 2.1, the *TAOConfigurator* contains a hook to which security strategies can be dynamically attached. When this happens, the new security strategy has the opportunity to add message-level interceptors (to encrypt/decrypt the message contents and authenticate communication peers) and request-level interceptors (to control the access to CORBA objects).

When using our *Cherubim_Security_Strategy*, applications are able to choose from a large range of security models including Discretionary Access Control (DAC), Double Discretionary Access Control (DDAC), and Mandatory Access Control (MAC) [28]. Cherubim adopts the general CORBA Security Reference Model and the OMG Security Service interfaces [23].

We are currently extending our implementation to support Role-Based Access Control (RBAC) [26] and message-level authentication and encryption. The new security system resulting from this effort will be the basis for security in the *2K* distributed environment.

4.1 Architecture

The Cherubim security framework supports access control by using *Active Capabilities* [2], pieces of Java bytecode that have the same role as conventional capabilities but that carry objects instead of just data. Active Capabilities are protected by digital signatures and encryption and are generated by an administrative tool that has access to a trusted secure store. All the information about user (or principal) roles and privileges are maintained in a secure store object called a *credential*. A single active capability can carry credentials for several objects and, since it contains interpretable code, it can support dynamic, flexible security policies, making decisions based on changing attributes such as location, resource availability, and other situation-specific parameters.

When a principal wants to access an object, it must first present the active capability and then send the desired requests. In our model, clients access secured objects by first installing an active capability into the Reference Monitor and then using the objects without having to worry about security. Alternatively, the active capabilities may be installed by a third party like an administrative tool, or fetched transparently by the Reference Monitor so that the application can be totally unaware of security. In our experiments, we adopted the last approach, which works with security-unaware applications.

Figure 10 shows the major components of our Reference Monitor architecture. If the *Cherubim_Security_Strategy* is attached to the *Server_Security_Strategy* hook in the *TAOConfigurator*, then all client requests are intercepted and delivered to the *Reference Monitor* module.

Before forwarding the call to the ORB, the Reference Monitor must check if the principal associated to the client sending the request is allowed to call that particular operation, on that particular object, with those particular arguments. The Reference Monitor first checks whether that security decision is available in the *Authorization Cache*. If the decision is cached, then it either forwards the call to the ORB (if the security decision is to grant access) or throws a CORBA *NO_PERMISSION* exception.

If the security decision is not available in the Cache, the Reference Monitor contacts the *Active Capability Evaluator*. If necessary, the Active Capability Evaluator contacts the *Policy Server* to fetch the active capability from the *Secure Store*. After the active capability is evaluated, the security decision is stored in the *Authorization Cache* for future use. If any credential in an active

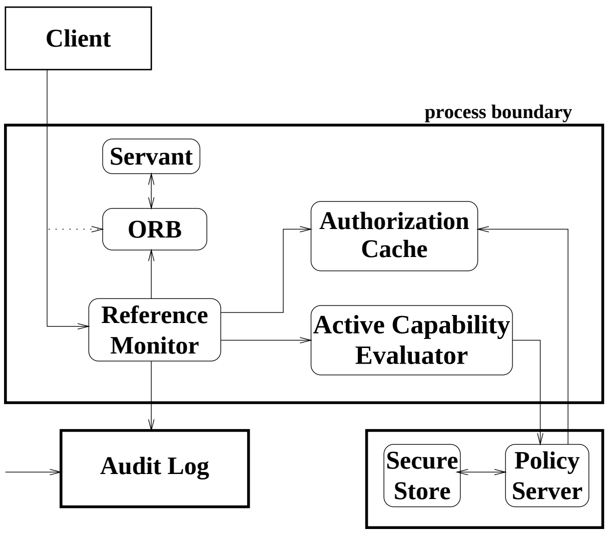


Fig. 10. The Reference Monitor Architecture

capability is revoked, the Policy Server contacts the Authorization Cache to update its list of security decisions.

If desired, security decisions can be stored in the *Audit Log*. The log can be used as a record of all security-sensitive operations performed in the system, assisting in the detection of attempted security violations.

The possibilities for dynamically configuring the security subsystem that *dynamicTAO* provides are very useful for a wide range of applications in several situations. As an example, consider a mobile computer moving from a corporate intranet towards a wireless satellite network. It may be acceptable to use light-weight encryption and soft access control in the intranet but it may be required to apply strong encryption and very tight access control policies when switching to the wireless network.

Each rectangle in figure 10 is a separate component that may be running on a separate machine. Thus, the system is subject to network partitions and to failures in individual components. Our current implementation requires that all the components be available, otherwise, it denies the access by throwing a security exception. The service could be extended to support different behaviors in the presence of network outages such as relying on a local versions of the policy server with limited functionality, and logging the events locally while the remote Audit Log is unreachable.

5 Componentizing the ORB

After our experience in developing applications with both open source and commercial ORBs, we came to the conclusion that typical applications utilize just a very small fraction of the services and functionalities provided by common ORBs. Besides, one of the criticism that CORBA often receives is that it is too big and heavy-weighted to be used in small devices and embedded systems. Although *dynamicTAO* can be configured dynamically, its memory footprint is never less than a Megabyte. It would be extremely difficult, if not impossible, to run it on a PDA such as the PalmPilot III. This motivated us to develop *LegORB*, which can be customized dynamically to adapt to resource availability and to accommodate the requirements of different applications and devices at different moments.

Mies van der Rohe's dictum "Less is more" is *LegORB*'s major tenet. It is a component-based ORB that can be configured at runtime so that it loads just enough components to provide the middleware services required by each application. Applications can select from a range of different implementations for each ORB component category and, as in *dynamicTAO*, replace components on-the-fly. To achieve minimal code size and high performance, we are writing the whole ORB source code from scratch, having small devices and componentization as our fundamental goals.

Unlike *dynamicTAO*, *LegORB* was designed having componentization and dynamic reconfiguration as a fundamental premise. We had embedded systems and PDAs in mind since the very beginning, which allowed us to achieve surprising results in terms of code size. A minimal configuration of LegORB containing just the basic infrastructure and a simple IIOP client engine that is able to send CORBA requests to standard ORBs occupies only around 6Kbytes on the PalmOS operating system for the PalmPilot. The server side includes extra functionality to receive and process client requests. Still, its size can be limited to around 10 to 20Kbytes. These LegORB instances are able to interoperate with traditional ORBs such as Orbix, ORBacus, and Washington University's TAO.

LegORB has a basic skeleton with a set of hooks to which infrastructure components are attached. These components then collaborate to offer ORB functionality. The set of hooks can be extended to accommodate situation specific functionality like real-time processing. Even though the categories are already defined, each category has different implementations. Combining different kinds of categories leads to different ORB behaviors.

The current implementation of the client side of the LegORB defines seven different categories: Invocation Interface, Connector, GIOP, IIOP, MIOP, Marshaler, and Demarshaler. Each category defines a standard interface that implementations of that category must provide. In addition to that, each implementation can add more functionality by offering a more detailed interface to be used by components that are aware of it.

One of the scenarios in which we are applying *2K* is in the context of *active spaces* such as smart rooms. These rooms contain computers, printers, video cameras, projectors, microphones, digital white boards, as well as other kinds of electric and electronic devices. In our preliminary experiments, we "CORBArized" some devices by implementing IDL interfaces that control video cameras, light switches, and even a microwave oven. By using well-defined interfaces and CORBA as a common communication substrate, we were able to integrate all these highly heterogeneous devices into the distributed system and interact with them not only by using powerful workstations running full CORBA implementations but also by using hand-held PalmPilot computers running our minimal ORB.

6 Related Work

Recent research in middleware have identified limitations on existing CORBA implementations, which led to ORB extensions for dealing with specific aspects such as real-time [7], group communication [20], and fault-tolerance [19]. Our goal, on the other hand, is to provide a generic infrastructure in which different kinds of customizations can be performed using reflection [18].

Other research groups have addressed the problem of middleware customization by using different approaches. The Operating Systems group at the Friedrich-Alexander University of Erlangen-Nürnberg is developing AspectIX [8], a configurable middleware architecture based on the fragmented object model. AspectIX clients would interact with a fragment of the global object (the fragment implementation) by using an interface (the fragment interface). The global object could be configured by using "profiles" which in turn specify "aspects" that must be supported by the fragment implementations. AspectIX Aspects can be compared to *dynamicTAO* category implementations with the difference that *dynamicTAO* implementations can be added on-the-fly. The AspectIX group plans to implement a prototype of their model where each object running within a single ORB would be able to specify its own policies and protocols. In *dynamicTAO*, a similar effect could be achieved by using different ORBs inside a single process and configuring each of the ORBs in a different way. In the *LegORB* model, on the other hand, the ORB can be configured to support any of the two approaches.

The Distributed Multimedia Research Group at the Lancaster University has proposed a reflective architecture for next generation middleware [1, 4]. They developed a prototype using the Python interpreted language in which the programmer is able to inspect and change the implementation at runtime. The level of reflection is much higher than in *dynamicTAO* since, in their Python system, it is possible to add or remove methods from objects and classes dynamically and even change the the class of an object at runtime. Their research has emphasized dynamic configurability through a well-defined *open binding* model which allows multiple reflective levels. In contrast, our research concentrates on a simpler reflective model, focusing on high performance. In our model, the reflective mechanisms are not included in the normal flow of control, they are only invoked when needed.

The Distributed Adaptive Run-Time (DART) [25] provides a framework where applications can modify their internal behavior as well as the behavior of services that they are using. It distinguishes between internal application adaptation (*Adaptive Methods*) and adaptation of the application's environment (*Reflective Methods*). In the case of Adaptive Methods, applications offer several implementations of each method. A special entity called *selector* chooses the most effective one at each invocation. In its turn, reflective methods allow adaptation of the runtime environment. When calling a reflective method, the call is redirected to a set of meta-level objects that manage run-time services. A *DART manager* (which can be compared to the *dynamicTAO* `DomainConfigurator`) stores adaptation information and references to applications and policies. Reconfiguration is triggered and controlled by using events that are also used to maintain consistency. Entities known as *policies* have the knowledge required to reconfigure applications. Policies use the DART manager to access applications as well as the meta-objects associated with them.

COMERA [34] (COM Extensible Remote Architecture) provides a framework based on Microsoft COM that allows users to modify several aspects of the communication middleware at run-time. It relies on the *Custom Marshaler* interface exported by COM, as well as the componentized architecture design that allows the use of user-specified components. By using COMERA, system developers can customize the middleware according to application requirements.

Previous work in system instrumentation and monitoring developed significant contributions that could be applied in the context of CORBA. The Pablo research group at the University of Illinois has developed a powerful framework for performance analysis and visualization [27]. In this framework, raw performance data is processed by performance visualization, correlation, evaluation, and interaction tools. Data is then correlated with appropriate network and computation components, both hardware and software, in order to highlight performance problems in meaningful ways.

The Distributed Object Visualization Environment (DOVE) [10] supports monitoring and visualization of applications and services in heterogeneous distributed systems. DOVE implements a flexible framework where DOVE-enabled applications use *application proxies* to send collected information to DOVE *agents*, which monitor and publish the information to DOVE-enabled browsers.

Unfortunately, most of the existing tools require that the applications be modified to include calls to the instrumentation libraries or monitoring agents. In our reflective approach, the monitoring system can be dynamically loaded into *dynamicTAO* and start to collect information selectively according to the user needs. There is absolutely no change required either in the application code or in the Monitoring Service code.

Our work on security builds on previous and ongoing work in standards for encryption, authentication, and access control [28, 26, 33]. Commercial products providing security for CORBA systems are starting to appear. However, to the best of our knowledge, no other implementation of the CORBA Security Service provides the degree of flexibility and dynamic configurability that our security architecture provides.

7 Future Work

We are currently developing new components for *LegORB*. Our long-term goal is to support full CORBA functionality through a component-based ORB. Fortunately, the *LegORB* architecture allows us to have working versions of the ORB from its early stages. Now, our work is to add new components incrementally until we achieve the complete functionality we desire. We are currently working on *LegORB* components supporting quality of service for multimedia applications [35] and fault-tolerance in real-time systems [30].

Our Monitoring Service currently does not provide support for visualizing the data that it captures. We will investigate the possibility of utilizing existing tools, like some of the DOVE components, to provide an interactive graphical interface to visualize the data and to configure the monitoring process without loosing the benefits of our system, namely, transparency, flexibility, and dynamic configurability.

Finally, we are extending the security architecture to add support for encryption and role-based access control for the *2K* distributed system by using UIUC Sesame [3] and scalable, dynamic security mechanisms [24].

8 Conclusions

Computing devices tend to become more and more pervasive in our society. Users will no longer tolerate having to adapt to different environments each time they interact with a computer. On the contrary, users expect the computer software to adapt itself to provide the service they need.

These highly dynamic environments with mobile computers, mobile software, and mobile users require a new paradigm for software development and deployment. Heraclitus argued change is the only constant. Middleware systems must be ready to adapt to change.

The ideas and architecture introduced by *dynamicTAO* provide a solid base for supporting safe dynamic reconfiguration of scalable, high-performance distributed systems. We are convinced that our reflective approach to middleware design provides the agility that modern applications require. Even though we are still far from having a complete solution for every aspect of the problem, preliminary results indicate that we are moving in the right direction.

The complete source code for *dynamicTAO* can be obtained from the *2K* web site at `http://choices.cs.uiuc.edu/2k/dynamicTAO`.

References

1. Gordon Blair, Geoff Coulson, Philippe Robin, and Michael Papathomas. An Architecture for Next Generation Middleware. In *Proceedings of Middleware '98*, Lake District, England, November 1998.

2. Roy Campbell and Tin Qian. Dynamic Agent-based Security Architecture for Mobile Computers. In *Proceedings of the Second International Conference on Parallel and Distributed Computing and Networks (PDCN'98)*, pages 291–299, Australia, December 1998.
3. Monika Chandak. Implementation of Sesame in Java. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.
4. Fabio Costa and Gordon Blair. A Reflective Architecture for Middleware: Design and Implementation. In *Proceedings of the ECOOP'99 Workshop for PhD Students in Object Oriented Systems*, Lisbon, June 1999.
5. Schmidt Douglas C. The ADAPTIVE Communication Environment. In *Proceedings of the Sun User Group Conference*, San Jose, California, December 1993.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Object-Oriented Software.* Addison-Wesley, 1995.
7. Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Object Event Service. In *Proceedings of the OOPSLA*. ACM, October 1997.
8. F. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, and M. Steckmeier. AspectIX: A Middleware for Aspect-Oriented Programming. In *Object-Oriented Technology, ECOOP'98 Workshop Reader, LNCS 1543*, pages 426–427. Springer-Verlag, 1998.
9. Prashant Jain and Douglas C. Schmidt. Dynamically Configuring Communication Services with the Service Configuration Pattern. *C++ Report*, 9(6), June 1997.
10. Michael Kircher and Douglas C. Schmidt. DOVE: A Distributed Object Visualization Environment. *C++ Report*, 11(3):42–51, March 1999.
11. Fabio Kon. Distributed Configuration Protocol. Project home page: `http://choices.cs.uiuc.edu/2k/DCP`, June 1998.
12. Fabio Kon and Roy H. Campbell. Supporting Automatic Configuration of Component-Based Distributed Systems. In *Proc. 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, pages 175–187, San Diego, CA, May 1999.
13. Fabio Kon and Roy H. Campbell. Dependence Management in Component-Based Distributed Systems. *IEEE Concurrency*, 2000. To appear.
14. Fabio Kon, Roy H. Campbell, See-Mong Tan, Miguel Valdez, Zhigang Chen, and Jim Wong. A Component-Based Architecture for Scalable Distributed Multimedia. In *Proceedings of the 14th International Conference on Advanced Science and Technology (ICAST'98)*, pages 121–135, Lucent Technologies, Naperville, April 1998.
15. Fabio Kon, Binny Gill, Roy H. Campbell, and M. Dennis Mickunas. Secure Dynamic Reconfiguration of Scalable CORBA Systems with Mobile Agents. Technical Report UIUCDCS-R-99-2131, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1999.
16. Fabio Kon, Ashish Singhai, Roy H. Campbell, Dulcineia Carvalho, Robert Moore, and Francisco J. Ballesteros. 2K: A Reflective, Component-Based Operating System for Rapidly Changing Environments. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, July 1998.
17. Ping Liu. The Design and Implementation of a Reference Monitor for the 2K Operating System. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1999.
18. P. Maes and D. Nardi, editors. *Meta-Level Architectures and Reflection.* North-Holland, 1987.

19. Silvano Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. In *Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies*. The USENIX Association, June 1995.
20. Silvano Maffeis and Douglas C. Schmidt. Constructing reliable distributed communication systems with CORBA. *IEEE Communications Magazine*, 14(2), February 1997.
21. Jina Mao. Monitoring and Analyzing Method Invocations in the 2K Operating System. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1999.
22. OMG. *CORBA v2.2 Specification*. Object Management Group, Framingham, MA, February 1998. OMG Document 98-07-01.
23. OMG. Security Service Specification (revision 1.2). Technical Report ptc/98-01-02, The Object Management Group, November 1998.
24. Tin Qian. *Dynamic Authorization Support in Large Distributed Systems*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, November 1999.
25. P.-G. Raverdy and R. Lea. DART: A Distributed Adaptive Run-Time. In *Work-in-progress presented at the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, September 1998.
26. Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Chlarles E. Youman. Role-based Access Control Models. *IEEE Computer*, 29(2):38–47, February 1996.
27. Daniel A. Reed and Randy L. Ribler. *Performance Analysis and Visualization*, chapter in the book "Computational Grids: State of the Art and Future Directions in High-Performance Distributed Computing". Morgan-Kaufman Publishers, August 1998.
28. Ravi S. Sandu and Pierangela Samarati. Access Control: Principles and Practice. *IEEE Communications Magazine*, 32(9):40–48, September 1994.
29. Douglas C. Schmidt and Chris Cleeland. Applying Patterns to Develop Extensible ORB Middleware. *IEEE Communications Magazine Special Issue on Design Patterns*, 1999.
30. Lui Sha, R. Rajkumar, and M. Gagliardi. Evolving Dependable Real Time Systems. In *Proceedings of the IEEE Aerospace Applications Conference*, pages 335–346, Aspen, CO, February 1996. IEEE Computer Society Press.
31. Ashish Singhai, Aamod Sane, and Roy Campbell. Reflective ORBs: Supporting Robust Time-Critical Distribution. In *Proceedings of the ECOOP'97 Workshop on Reflective Real-Time Object-Oriented Systems*, pages 55–61, Finland, June 1997. ECOOP'97 Workshop Reader, LNCS 1357.
32. Ashish Singhai, Aamod Sane, and Roy Campbell. Quarterware for Middleware. In *Proc. 18th International Conference on Distributed Computing Systems (ICDCS)*, pages 192–201. IEEE, May 1998.
33. M. Vandenwauver, R. Govaerts, and J. Vandewalle. Overview of Authentication Protocols: Kerberos and SESAME. In *Proceedings of the 31st Annual IEEE Carnahan Conference on Security Technology*, pages 108–113, 1997.
34. Y. M. Wang and Woei-Jyh Lee. COMERA: COM extensible remoting architecture. In *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS)*. Usenix, April 1998.
35. Dongyan Xu, Duangdao Wichadakul, and Klara Nahrstedt. Multimedia Service Configuration and Reservation in Heterogeneous Environments. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS'2000)*, Taipei, Taiwan, April 2000.