AUTOMATIC CONFIGURATION OF COMPONENT-BASED DISTRIBUTED SYSTEMS

BY

FABIO KON

B.S., University of São Paulo, 1991
M.S., University of São Paulo, 1994

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2000

Urbana, Illinois

Abstract

Recent developments in component technology enable the construction of complex software systems by assembling off-the-shelf components. However, it is still difficult to develop efficient, reliable, and dynamically configurable component-based systems. Components are often developed by different groups with different methodologies. Unspecified dependencies and behavior lead to unexpected failures.

This thesis demonstrates that the explicit representation of inter-component dependence is fundamental for the development of efficient, configurable, and reliable component-based systems. This provides a common ground for supporting fault-tolerance and automating dynamic configuration to facilitate system management and maintenance.

In this thesis, we present a generic architecture for managing dependencies in distributed component systems, describe a concrete implementation of this architecture, and discuss how it can be used to support automatic configuration. The architecture is divided in three parts: the Automatic Configuration Service, the component configurator framework, and the reconfiguration agents framework.

We describe the deployment of the architecture in centralized and distributed systems and present detailed experimental results.

To Paula

Acknowledgments

I would like to thank my advisor, Prof. Roy Campbell, for having always believed in my work and for providing insightful feedback during all the stages of the research described in this thesis. His vision was fundamental in shaping my research and I am very grateful for having had the opportunity to learn from him.

The students at the Systems Research Lab provided a precious collaborative working environment that will be hard to forget. To save trees, I must not list the names of all the DCL people that contributed to my work. Such list would certainly include Manuel Román, Christopher Hess, Dulcineia Carvalho, Tomonori Yamane, Prasad Naldurg, Vijay Gupta, Luiz Claudio Magalhães, Ashish Singhai, Miguel Valdez, Chuck Thompson, Mario Medina, Marcia Muñoz, and others from the SRG, 2K, and Vosaic teams. Anda Ohlsson and Bonnie Howard made my life at the department much easier.

The members of my thesis committee, Professors Geneva Belford, Klara Nahrstedt, and Dennis Mickunas provided valuable feedback at different phases of my research.

Prof. Francisco Ballesteros was always so readily available to help me with everything I needed that I often forgot that we were separated by 6886 Kms. Alexandre Oliva provided a safe environment for my reconfiguration agents at the University of Campinas. Prof. Dilma Menezes shared with me some of her inspiring ideas and was brave enough to actually use my code.

Paula was the best person in the world I could have had to share all the good and bad moments of my PhD. I was very lucky to have found her. Last but not least, I would like to thank my parents, ZecAnita, my siblings, LeNeRu, and all my family for their great emotional support over all these years, and Larry and family, who made us feel more at home in these distant lands.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ACE Adaptive Communication Environment

CORBA Common Object Request Broker Architecture

DCOM Distributed Component Object Model

DCP Distributed Configuration Protocol

Doctor Distributed ORB Configuration Tool

DSRT Dynamic Soft Real-Time Scheduler

OMG Object Management Group

ORB Object Request Broker

PDA Personal Digital Assistant

RMI Remote Method Invocation

SPDF Simple Prerequisite Description Format

Chapter 1

Introduction

There is nothing permanent except change.

Heraclitus of Ephesus (535-475 BC)

As computer systems are being applied to more and more aspects of personal and professional life, the quantity and complexity of software systems is increasing considerably. At the same time, the diversity in hardware architectures remains large and is likely to grow with the deployment of embedded systems, PDAs, and portable computing devices. All these architectures will coexist with personal computers, workstations, computing servers, and supercomputers. It is imperative that the scientific and industrial communities develop techniques to support seamless integration of highly heterogeneous environments. The construction of these new software systems and applications in an easy and reliable way can only be achieved through the composition of modular hardware and software.

Component technology has appeared as a powerful tool to confront this challenge. Recently developed component architectures support the construction of sophisticated systems by assembling together a collection of off-the-shelf software components with the help of visual tools or programmatic interfaces. Components will be the unit of packaging, distribution, and deployment in the next generation of software systems. However, there is still very little support for managing the dependencies among components. Components are created by different programmers, often working in different groups with different methodologies. It is hard to create robust and efficient systems if the dependencies between components are not well understood.

Mobile computing and active spaces seem to be among the next revolutions in information

technology. Mobile computing comprises systems as diverse as digital cellular telephony, road traffic information, or simply web browsing via wireless connections. Active spaces consist of physical spaces – such as offices, lecture and meeting rooms, homes, hospitals, campuses, train stations, cities – that are augmented with computing devices integrated into the environment. The objective of these devices is to provide information to users of the space, helping them to perform activities they would not be able to perform otherwise, or helping them to perform conventional activities more easily. In addition, multimedia interfaces will play a fundamental role in human-computer interaction. Applications will gradually abandon dull interfaces based on text and static graphics and move towards interfaces with a more dynamic appearance based on audio, video, and animations. We can also expect the migration of services such as radio and TV to the personal computer. Within one decade, there will probably be no distinction among the telephone, radio, TV, and data networks. In the future, there will be no distinction between computer and dedicated radio and TV receivers. Everything will be integrated into an extension of today's Internet. The dynamic nature of multimedia, with its stringent, time-sensitive resource requirements, will contribute to the changes towards more dynamic computing environments.

Until recently, these highly-dynamic environments with mobile computers, active spaces, and ubiquitous multimedia were only present in science fiction stories or in the minds of visionary scientists like Mark Weiser [Wei92, WSA+95]. But now, they are becoming a reality and one of the most important challenges they pose is the proper *management of dynamism.* Future computer systems must be able to configure themselves dynamically, adapting to the environment in which they are executing. Furthermore, they must be able to react to changes in the environment by dynamically *re*configuring themselves to keep functioning with good performance, irrespective of modifications in the environment. Applications and systems must react to variations in resource availability by dynamically adapting their algorithms, updating parts of the system, and replacing software components when needed.

Unfortunately, the existing software infrastructure is not prepared to manage these highly-dynamic environments properly. Conventional operating systems already have a hard time managing static environments based on components. Even Microsoft researchers say that they are "well aware of how difficult it is to install software on [Windows] NT and get it to work" [MB+99] because

of the management problems that components induce. In Solaris, a system administrator does not have a clean way of upgrading software that is being executed by the users. When an executable is replaced, the running versions of the old executable crash.

1.1 The Heart of the Problem

Existing software systems face significant problems with reliability, administration, architectural organization, and configuration. The problem behind all these difficulties is the lack of a unified model for representing dependencies among system and application components and mechanisms for dealing with these dependencies.

As systems become more complex and grow in scale, and as environments become more dynamic, the effects of the lack of proper dependence management become more dramatic. Therefore, we need an integrated approach in which operating systems, middleware, and applications collaborate to manage the components in complex software systems, dealing with their dependencies properly.

1.2 Thesis Contribution

To solve the problem described in the previous section, this thesis introduces an architecture that provides support for systems and applications to reason about their own internal structural organization. We call this ability *architectural-awareness* and we achieve it with a framework that supports the explicit representation of dependencies in component-based systems. This representation can then be manipulated in order to implement software components that are able to configure themselves and adapt to ever changing dynamic environments.

By reifying the interactions between system and application components, system software can recognize the need for reconfiguration to provide better support for fault-tolerance, security, quality of service, and optimizations. In addition, it gains the means to carry out this reconfiguration without compromising system stability and reliability, with minimal impact on performance.

Our research builds on previous and ongoing work on software architecture and dynamic configuration of distributed systems. But, rather than simply looking at the architectural connections among the components of a single application, our approach differs from previous work in this

area by reasoning about all the different kinds of dependencies that relate each component to other application, middleware, and system components. In addition, instead of relying on architecture descriptions defined *a priori*, our framework is prepared to deal with completely dynamic architectures that are only known at runtime.

In this thesis, we present a generic architecture for dependence representation and management and describe a concrete implementation of this architecture. The latter was deployed successfully in several distributed systems and in a centralized system. This demonstrates the effectiveness of our approach in improving the quality of component-based systems with respect to dynamic configuration, reliability, architectural organization, and management.

1.3 Thesis Contents

The remainder of this thesis is organized as follows. Chapter 2 discusses how dependence management in component-based distributed systems can help to provide reliability, quality of service, and dynamic configuration and reconfiguration.

Chapter 3 gives a general overview of our novel architecture for dependence management. The architecture is divided in three parts that are addressed in more detail by the three subsequent chapters. Chapter 4 describes the Automatic Configuration Service, Chapter 5 describes the component configurator framework, and Chapter 6 describes the reconfiguration agents framework.

Chapter 7 describes, in detail, two application scenarios where our architecture was deployed. Section 7.1 describes the *dynamicTAO* reflective ORB and Section 7.2 describes a scalable multimedia distribution system. Section 7.3 describes briefly three other ongoing projects that use the results of this thesis.

Chapter 8 presents the results of experiments with the three major parts of our architecture. It discusses the performance, impact, and overhead of dependence management in different contexts.

Chapter 9 describes related research in different Computer Science fields, discusses their limitations, and shows how they compare to the novel ideas introduced in this thesis.

Finally, Chapter 10 discusses several possibilities for extending this work in the future and Chapter 11 presents our conclusions, emphasizing this thesis's original contributions.

Chapter 2

Dependence Management

Before going into the details about how our architecture works, we present an overview of the issues related to dependence management. We discuss how the reification of dependence relationships can help support automatic configuration, dynamic reconfiguration, fault-tolerance, and adaptation.

2.1 Automatic Configuration

Users of modern computing environments have to deal with many different devices, ranging from PDAs and embedded systems to powerful workstations and servers. These devices are managed by different operating systems and depend on different interfaces for configuration. As a result, users are overwhelmed with the huge amount of manual configuration that these systems require. In recent years, many of us have had to learn to deal with poorly configured systems, looking for "work-arounds", just because it is too costly to configure them properly.

In spite of the increasing need for self-configuring systems, not enough research has been carried out in this field. We identify a few trends in this area.

1. Systems such as MS-Windows provide graphical "wizard" interfaces that help configuring applications and services as they are installed. However, since dependence management is not addressed, it is very common to experience errors in wizard operations, requiring manual intervention from users and system administrators.

2. Operating systems such as Linux, rely on a tool called *GNU Autoconf* [ME98] to detect the characteristics of the underlying hardware and software platforms and automate the configuration of new applications and services at compile-time. Autoconf is used in a large

number of free software packages that are designed to run on a wide variety of platforms. Obviously, the level of automatic configuration provided by such tools is very limited since it is restricted to compile-time configuration.

3. The Debian Linux operating system [Deb00] includes *dpkg*, a package maintenance utility that manages the dependencies between the executables, libraries, documentation, and data files associated with Debian software packages. It maintains databases of dependencies that let users install, update, and delete packages without compromising consistency. Even when libraries are shared by multiple packages, *dpkg* is able to manage them consistently. New packages can be read from a local CD or fetched from a remote FTP site.

   To support the update of software packages that are being executed at the time of the update, the utility renames the files related to the running version and installs the update using the package official file names. Then, the next time the package is restarted, the updated version is executed. This mechanism allows, for example, system administrators to update the network packages using FTP (a client of the network packages).

4. Jini [Wal98] provides simple mechanisms for enabling plug-and-play Java services and facilitating configuration. Jini does not address management of component-based applications and inter-component dependence. It does not provide a model for specifying hardware and software requirements.

5. Academic research in automatic configuration has addressed this problem within the realm of software architecture and Architectural Description Languages (ADLs). ADLs such as Darwin [MDK94, MTK97] and UniCon [SDZ96] specify application architecture as a collection of modules linked by *bindings* and *connectors*, respectively. At application startup time, the underlying system processes the application architectural description, instantiates the required components, and establishes the links among them.

   ADL descriptions are limited to the internal architecture of a particular application or service. They usually do not represent the interactions between the application and the underlying system and the interactions among applications. In addition, this approach requires predefined, fixed architectural descriptions and does not work well in dynamic environments

where the architecture is not known until runtime. We present a more detailed description of related work in software architecture and ADLs in Section 9.5.

Although we recognize the importance of the existing software tools and previous research results in software architecture, we are convinced that they are not enough to provide a definitive solution to the problem of automating configuration.

What is missing is a model that lets each system and application component specify its dependencies with respect to other components, applications, system services, and hardware resources. With such an explicit representation of dependencies, it becomes possible to implement services that support automatic configuration.

2.2 Dynamic Reconfiguration

Besides helping to establish the *initial* configuration automatically, proper dependence management also helps support dynamic reconfiguration of running applications and services. The importance of dynamic reconfiguration can be understood by considering the rapid proliferation of the Internet and the recent evolution of its supporting software.

The scope of Internet services continues to expand. It stretches to new fields, reaches increasing numbers of people, and encompasses more and more human activities in the virtual world of the World Wide Web. Different forms of electronic commerce ranging from airline ticket reservation to pizza delivery are now commonly available. Activities such as management of bank accounts, reading news, accessing financial information, filing income taxes, submitting articles and registering for scientific conferences, getting weather and natural disaster information, and consolidating votes in presidential elections or referenda stress the importance of service availability, reliability, and security. However, the rapid evolution of software requires frequent code updates and highly-dynamic environments require reconfiguration of system parameters. Maintaining the flexibility and rapid growth of these systems while ensuring the service requirements on such a scale as the Internet is a difficult problem.

Services must grow to meet increasing usage, new requirements, and new applications. However, flexibility usually conflicts with availability. In conventional systems, the service provider must often

shut down, reconfigure, and restart the service to update or reconfigure it. In many cases, it is unacceptable to disrupt the services for any period of time. Disruption may result in business loss, as in the case of electronic commerce, or it may put lives in danger, as in the case of mission critical systems delivering disaster information. Research in dynamic reconfiguration [Kra90, HWP93, End94, MTK97, BBB+98, BISZ98, OT98, SW98, PCS98, MG99] seeks solutions to this problem.

By breaking a complex system into smaller components and by allowing the dynamic replacement and reconfiguration of individual components with minimal disruption of system execution, it is possible to combine high degrees of flexibility and availability.

As pointed out by Goudarzi in [MG99], dynamic reconfiguration involves three basic issues: specification and management of change, preservation of system consistency, and minimization of disruption to the provided service.

Previous approaches to dynamic reconfiguration are based on reconfiguration scripts that must have some knowledge about the application architecture. However, as mentioned before, if the architecture of the system being reconfigured is not known until runtime, then an approach that requires *a priori* knowledge of the global system architecture is not appropriate.

Instead of dealing with global architectures, our approach focuses on inter-component dependencies and lets users, system administrators, and programs to replace components and reconfigure architectures at runtime.

The mechanisms for dependence management presented in Chapter 3 provide the necessary support for software developers to implement reconfigurable systems that preserve consistency and minimize disruption[1].

2.3 Fault-Tolerance

Distributed systems may malfunction for a variety of reasons, including network failures, node failures, software errors, and even programmed system shutdowns for maintenance. In addition, in component-based systems it is also common to have situations in which a part of the system fails while the remaining parts continue to work properly. The challenge is to develop fault-tolerant

[1] As we show in the next chapters, our architecture *per se* does not guarantee consistency and minimal disruption but it provides the system support that is required to implement it.

component-based systems that are able to detect partial failures and change themselves to achieve a new configuration where system execution can continue normally. To carry out reconfiguration in the event of failure, it is important to know which components depend on the faulty component and which components depend on the parts of the system being reconfigured.

It is hard to create robust and efficient systems if the dynamic dependencies between components are not well understood. Furthermore, it is very common to find cases, in which a component fails to accomplish its goal because an unspecified dependency is not properly resolved. Sometimes, the graceful failure of one component is not properly detected by other components leading to total system failure. This thesis shows how proper dependence management can help in the development of robust distributed systems.

2.4 Adaptation

Highly heterogeneous platforms and varying resource availability motivates the need for self-adapting software. Applications can improve their performance by using different algorithms in different situations and switching from one algorithm to another according to environmental conditions. Significant variations in resource availability should trigger architectural reconfigurations, component replacements, and changes in the components' internal parameters.

Consider, for example, the network connectivity of a mobile computer as its user commutes from work to home. As the user switches from a wired connection at the office, to a wireless WAN using a cellular phone, and finally, to a modem connection at home, the available bandwidth changes by several orders of magnitude. The movement is also accompanied by changes in latency, error rates, connectivity, protocols, and cost.

Ideally, we would like to have a system capable of maintaining an explicit representation of the dependencies among the network drivers, transport protocols, communication services, and the application components that use them. Only then, it would be possible to inform the interested parties when significant changes occur. Upon receiving the change notifications, applications and services can select different mechanisms, replace components, and modify their internal configuration to adapt to the changes, optimizing performance.

In the next few chapters, we present a novel architecture for dependence management and show how it facilitates the construction of flexible software systems for dynamic environments.

Chapter 3

Overall Architecture

In the introductory chapter, we saw that modern middleware and operating systems do not provide proper mechanisms for dependence management in component-based distributed systems. The lack of such mechanisms leads to several problems related to administration, reliability, quality of service, and dynamic configuration. To solve these problems, we present mechanisms for dependence representation in complex software systems and describe an integrated architecture that uses these mechanisms to provide dynamic (re)configuration while still supporting reliability, quality of service, customizability, and scalability.

3.1 The Big Picture

Figure 3.1 presents a schematic view of the major elements of our architecture. It shows the chapters in which each element is described in detail. *Prerequisite specifications* reify static dependencies of components towards its environment while *component configurators* reify dynamic, runtime dependencies. *Automatic configuration* aims at automating management, easing the work of users and administrators. The automatic configuration process is based on the prerequisite specifications, and constructs the component configurators. *Mobile reconfiguration agents*, in turn, improve management by making it more scalable and efficient.

The contributions of this thesis focus on the four aspects on the top of Figure 3.1, leveraging the work of others on standard CORBA Services and on QoS-aware Resource Management. Figure 3.2 depicts the relationships among the six elements in the architecture.

In the following sections of this chapter, we provide an overview of the major elements of our

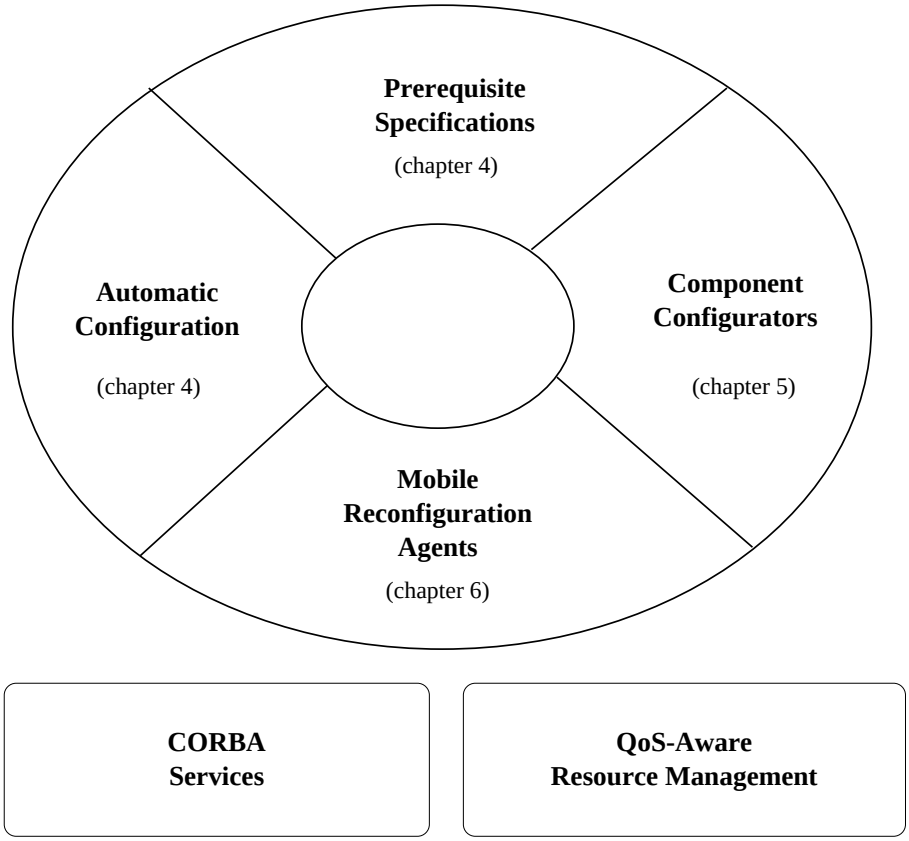


Figure 3.1: Overall Architecture for Dynamic Configuration

architecture explaining the relationships depicted in Figure 3.2. In the subsequent chapters, we give a more in-depth description of each of the aspects of the architecture.

3.2 Automatic Configuration

Software systems are evolving more rapidly than ever before. Vendors release new versions of web browsers, text editors, and operating systems once every few months. System administrators and users of personal computers spend an excessive amount of time and effort configuring their computer accounts, installing new programs, and, above all, struggling to make all the software work together[1].

In environments like MS-Windows, the installation of some applications is partially automated by "wizard" interfaces that direct the user through the installation process. However, it is common

[1]When even PhD students in Computer Science have trouble keeping their commodity personal computers functioning properly, one can notice that something is very wrong in the way that commercial software is built nowadays.

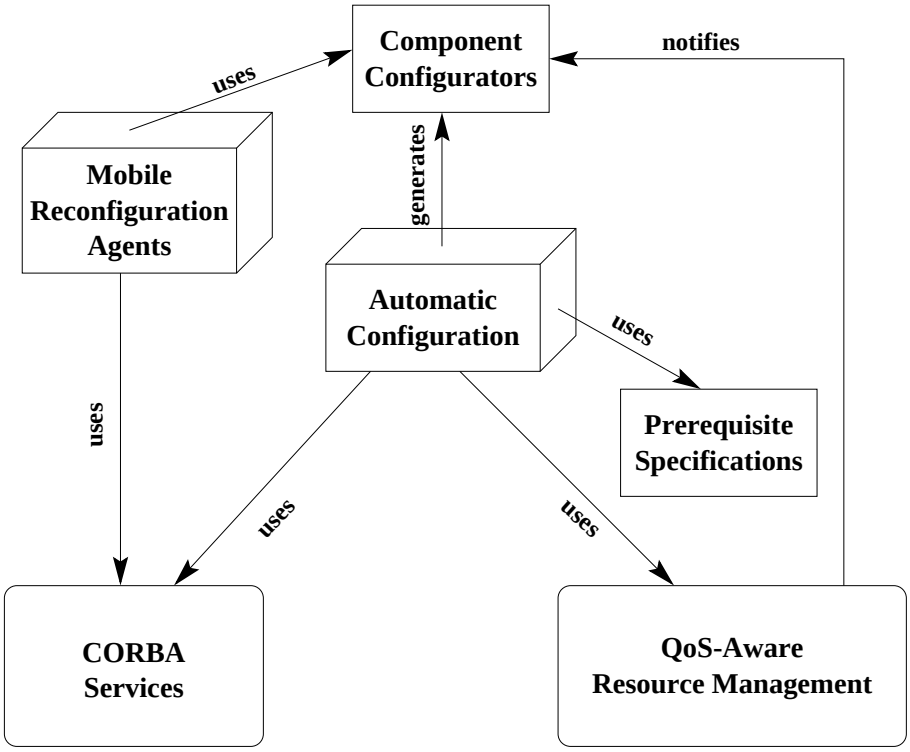


Figure 3.2: Relationships

to face situations in which the installation cannot complete or in which it completes but the software package does not run properly because some of its (unspecified) requirements are not met. In other cases, after installing a new version of a system component or a new tool, applications that used to work before the update, stop functioning. It is typical that applications on MS-Windows cannot be cleanly uninstalled. Often, after executing special uninstall procedures, "junk" libraries and files are left in the system. The application does not know if it can remove all the files it has installed because the system does not provide the clear mechanisms to specify which applications are using which libraries.

To solve this problem, we need a completely new paradigm for installing, updating, and removing software from workstations and personal computers. We propose to automate the process of software maintenance with a mechanism we call *Automatic Configuration*. In our design of an automatic configuration service for modern computer environments, we focus on two key objectives:

1. Network-Centrism and

2. an "What You Need Is What You Get" (WYNIWYG) model.

Network-Centrism refers to a model in which all entities, users, software components, and devices exist in the network and are represented as distributed objects. Each entity has a network-wide identity, a network-wide profile, and dependencies on other network entities. When a particular service is configured, the entities that constitute that service are assembled dynamically. Users no longer need to keep several different accounts, one for each device they use. In the network-centric model, a user has a single network-wide account, with a single network-wide profile that can be accessed from anywhere in the distributed system. The middleware is responsible for instantiating user environments dynamically according to the user's profile, role, and the underlying platform [CKB+00].

In contrast to existing operating systems, middleware, and applications where a large number of non-utilized modules are carried along with the standard installation, we adopt a *What You Need Is What You Get* model, or *WYNIWYG*. In other words, the system configures itself automatically and loads a *minimal* set of components required for executing the user applications in the most efficient way. The components are downloaded from the network, so only a small subset of system services are needed to bootstrap a node, leading to the construction of "network-computers".

In the Automatic Configuration model, system and application software are composed of network-centric components, i.e., components available for download from a *Component Repository* present in the network. Component code is encapsulated in dynamically loadable libraries (known as "DLLs" in Windows and "shared objects" in Unix), which enables dynamic linking.

Each application, system, or component[2] specifies everything that is required for it to work properly (both hardware and software requirements). This collection of requirements is called *Prerequisite Specifications* or, simply, *Prerequisites*.

3.2.1 Prerequisites

The prerequisites for a particular inert component (stored on a local disk or on a network component repository) must specify any special requirement for properly loading, configuring, and executing that component. We consider three different kinds of information that can be contained in a list

[2]From now on, we use the term "component" not only to refer to a piece of an application or system but also to refer to the entire application or system. This is consistent since, in our model, applications and systems are simply components that are made of smaller components.

of prerequisites.

1. The nature of the hardware resources the component needs.

2. The capacity of the hardware resources it needs.

3. The software services (i.e., other components) it requires.

The first two items are used by *QoS-Aware Resource Management Services* to determine where, how, and when to execute the component. QoS-aware systems can use these data to enable proper admission control, resource negotiation, and resource reservation. The last item determines which auxiliary components must be loaded and in which kind of software environment they will execute.

The first two items – reminiscent of the Job Control Languages of the mid-1960s – can be expressed by modern QoS specification languages such as QML [FK99b] and QoS aspect languages [LBS+98], or by using a simpler format such as SPDF (see Section 4.2.1 and [KCM+99]). The third item is equivalent to the *require* clause in architectural description languages like Darwin [MDK94] and module interconnection languages like the one used in Polylith [Pur94].

The prerequisites are instrumental in implementing the WYNIWYG model as they let the system know what the exact requirements are, for instantiating the components properly. If the prerequisites are specified correctly, the system not only executes all the necessary components to activate the user environment, but also executes a minimal set of components required to achieve that.

We currently rely on the component programmer to specify component prerequisites. Mechanisms for automating the creation of prerequisite specifications and for verifying their correctness require further research and are beyond the scope of this thesis. Another interesting topic for future research is the refinement of prerequisites specifications at runtime according to what the system can learn from the execution of components in a certain environment. This can be achieved by using QoS profiling tools such as QualProbes [LN00].

3.3 Component Configurator

The explicit representation of the dynamic dependencies is achieved through special objects attached to each relevant component at execution time. These objects are called *component configu-*

rators; they are responsible for reifying the runtime dependencies for a certain component and for implementing policies to deal with events coming from other components.

While the Automatic Configuration Service parses the prerequisite specifications, fetches the required components from the Component Repository, and dynamically loads their code into the system runtime, it uses the information in the prerequisite specifications to create component configurators representing the runtime inter-component dependencies. Figure 3.3 depicts the dependencies that a component configurator reifies.

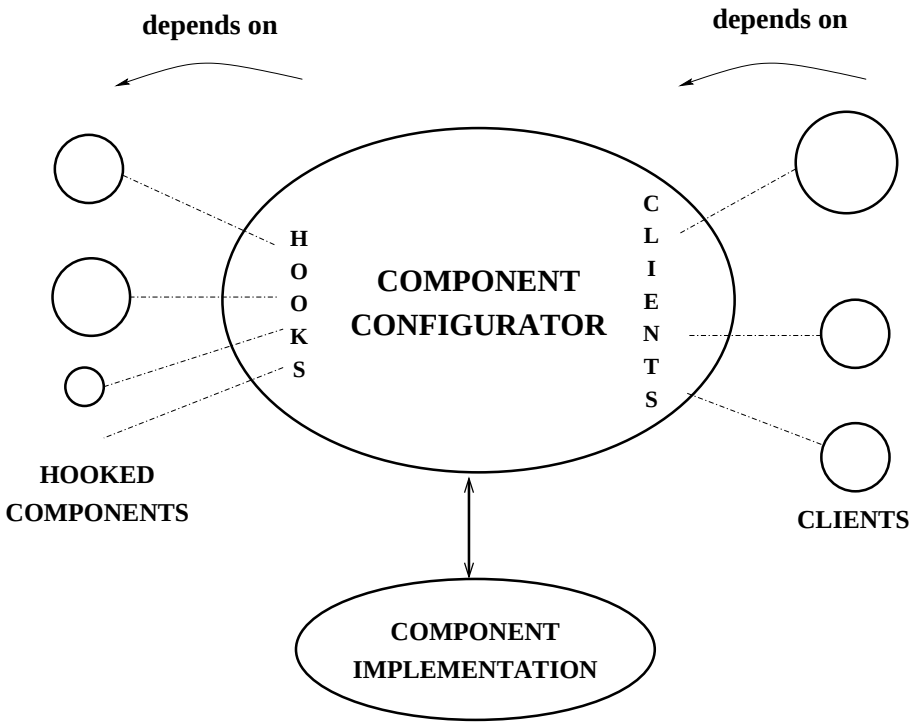


Figure 3.3: Reification of Component Dependencies

Each component C has a set of *hooks* to which other components can be attached. These are the components on which C depends and are called *hooked components*. There might be other components that depend on C, these are called *clients*. In general, every time one defines that a component C_1 depends on a component C_2, the system should perform two actions:

1. attach C_2 to one of the hooks in C_1 and
2. add C_1 to the list of clients of C_2.

As an example, consider a web browser that specifies, in its list of prerequisites, that it requires a

TCP/IP service, a window manager, a local file service, and an implementation of the Java Virtual Machine (JVM). Its component configurator should maintain a hook for each of these services. When the browser is loaded, the automatic configuration system must verify whether these services are available in the local environment. If they are not, it must create new instances of them. In any case, references to the services are stored in the browser configurator hooks and may be later retrieved and updated if necessary.

Component configurators are also responsible for distributing events across the inter-dependent components. Examples of common events are the failure of a client and destruction, internal reconfiguration, or replacement of the implementation of a hooked component. The rationale is that such events affect all the dependent components. The component configurator is the place where programmers must insert the code to deal with these configuration-related events.

Component developers can program specialized versions of component configurators that are aware of the characteristics of specific components. These specialized configurators can, therefore, implement customized policies to deal with component dependencies in an application-specific way.

For an example of how customized component configurators could help applications, consider a QoS-sensitive video-on-demand client that reserves a portion of the local CPU for decoding a video stream. The application developer can program a special configurator that registers itself with the QoS-aware resource management service. In this way, when the resource management service detects a change in resource availability that would prevent the application from getting the desired level of service, it notifies the configurator (as shown in Figure 3.2). The configurator, with its customized knowledge about the application, sends a message to the video server requesting that the latter decrease the video frame rate. Then, with a lower frame rate, the client is able to process the video while the limited resource availability persists. When the resources go back to normal, another notification allows the video-on-demand configurator to re-establish the initial level of service.

3.4 Reconfiguration and Inspection Agents

The automatic configuration mechanism described in Section 3.2 adopts a pull-based approach for code updates and configuration. In other words, the service running in a certain network node

takes the initiative to *pull* the code and configuration information from a Component Repository. In large-scale systems, this may not be enough. To support efficient and scalable management, we also need a mechanism to allow system administrators to *push* code and configuration information into the network. Our architecture achieves this by using the concept of mobile agents [KG99].

The architecture comprises of special administrative tools that are used to create mobile agents and inject them into the network. The agent traverses the network and, in each node, is able to perform three kinds of actions:

1. install the code for new components,
2. reconfigure the node, changing its working parameters and its software architecture (e.g., by adding, removing, and replacing components), and
3. collect information about the node's dynamic state.

After visiting all the nodes specified by the administrator, the agent returns to the administrator tool and displays the results of its actions along with the collected information.

At each node, the agent inspects and reconfigures the components through their component configurators so that the dependencies are still managed properly. Reconfiguration and inspection agents can be built over low-level mechanisms such as sockets and simple command interpreters. They can also use more sophisticated facilities like the CORBA Naming Service for locating the nodes to be visited, the CORBA middleware for communication, and a Java virtual machine for agent execution.

The following chapters present an in-depth description of the issues associated with each of the architectural elements presented in this chapter and discusses their implementation in more detail.

Chapter 4

Automatic Configuration Service

As described in the previous chapter, automatic configuration enables the construction of network-centric systems following a WYNIWYG model. To experiment with these ideas, we developed an Automatic Configuration Service for the *2K* operating system [KCM+00]. In this chapter, we present the framework in which the service is based and describe a concrete implementation of the service. We conclude the chapter by discussing interactions with other services and use cases.

4.1 A Flexible Framework

Different applications domains may have different ways of specifying the prerequisites of their application components. Therefore, rather than limiting the specification of prerequisites to a particular language, we created a framework in which different kinds of prerequisite descriptions can be utilized. To validate the framework, we designed SPDF, a very simple, text-based format that allowed us to perform initial experiments. In the future, other more elaborated prerequisite formats including sophisticated QoS descriptions [FK99b, LBS+98] can be plugged into the framework easily.

In addition, depending upon the dynamic availability of resources and connectivity constraints, different algorithms for prerequisite resolution may be desired. For example, if a diskless PDA is connected to a network through a 2Mbps wireless connection, it will be beneficial to download all the required components from a central repository each time they are needed. On the other hand, if a laptop computer with a large disk connects to the network via modem, it will probably be better to cache the components in the local disk and re-use them whenever is possible.

Figure 4.1 shows how the architecture uses the two basic classes of the Automatic Configuration framework: *prerequisite parsers* and *prerequisite resolvers*. Administrators and developers can plug different concrete implementations of these classes to implement customized policies.

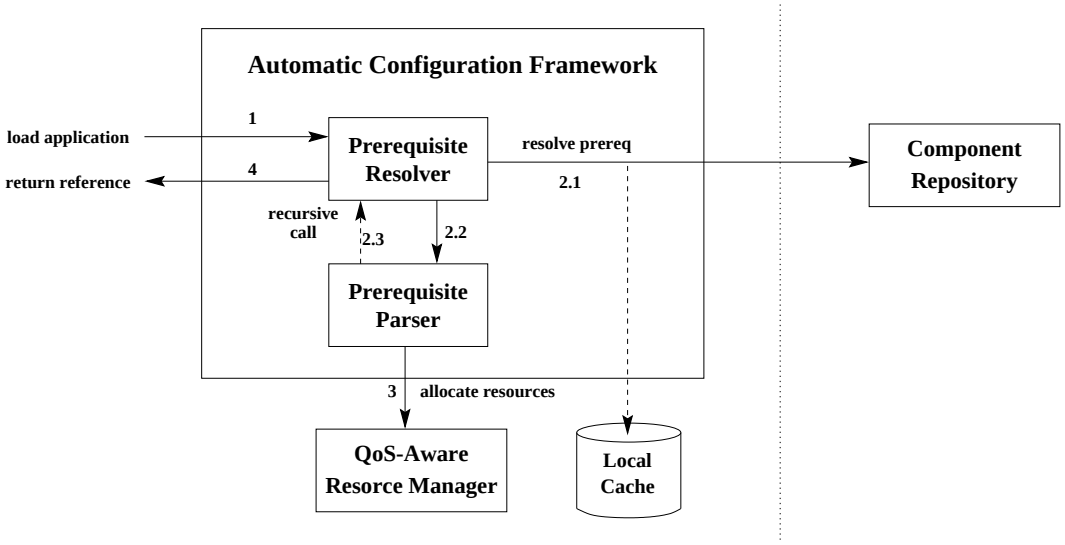


Figure 4.1: Automatic Configuration Architecture

The architecture for automatic configuration works as follows. First, the client sends a request for loading an application by passing, as parameters, the name of the application's "master" component and a reference to a component repository (step 1 in Figure 4.1). The request is received by the prerequisite resolver, which fetches the component code and prerequisite specification from the given repository, or from a local cache, depending on the policy being used (step 2.1).

Next, the prerequisite resolver calls the prerequisite parser to process the prerequisite specification (step 2.2). As it scans the specification, the parser issues recursive calls to the prerequisite resolver to load the components on which the component being processed depends (step 2.3). This may trigger several iterations over steps 2.1, 2.2, and 2.3.

After all the dependencies of a given component are resolved, the parser issues a call to the Resource Manager to negotiate the allocation of the required resources (step 3). After all the application components are loaded, the service returns a reference to the new application to the client (step 4).

4.2 A Concrete Implementation

To evaluate the framework, we created concrete implementations of the prerequisite parser and resolver. The prerequisite parser, called `SPDFParser`, processes SPDF specifications. The first prerequisite resolver, called `SimpleResolver`, uses CORBA to fetch components from the *2K* Component Repository. The second, called `CachingResolver`, is a subclass of `SimpleResolver` that caches the components on the local file system.

4.2.1 SPDF

We designed the Simple Prerequisite Description Format (SPDF) to serve as a proof-of-concept for our framework. An SPDF specification is divided in two parts, the first is called hardware requirements and the second, software requirements. Figure 4.2 shows an example of an SPDF specification for a hypothetical web browser. The first part specifies that this application was compiled for a Sparc machine running Solaris 2.7, that it requires at least 5MB of RAM memory but that it functions optimally with 40 MB of memory, and that it requires 10% of a CPU with speed higher than 300MHz.

```
:hardware requirements
machine_type    SPARC
os_name         Solaris
os_version      2.7
min_ram         5MB
optimal_ram     40MB
cpu_speed       >300MHz
cpu_share       10%

:software requirements
FileSystem    CR:/sys/storage/DFS1.0 (optional)
TCPNetworking CR:/sys/networking/BSD-sockets
WindowManager CR:/sys/WinManagers/simpleWin
JVM           CR:/interp/Java/jvm1.2 (optional)
```

Figure 4.2: A Simple Prerequisite Description

The second part, software requirements, specifies that the web browser requires four components (or services): a file system (to use as a local cache for web pages), a TCP networking service (to fetch the web pages), a window manager (to display the pages), and a Java virtual machine (to interpret Java Applets).

The line

```
FileSystem CR:/sys/storage/DFS1.0 (optional)
```

specifies that the component that implements the file system (or the proxy that interacts with the file system) can be located in the directory `/sys/storage/DFS1.0` of the component repository (`CR`). It also states that the file system is an "optional" component, which means that the web browser can still function without a cache. Thus, if the Automatic Configuration Service is not able to load the file system component, it simply issues a warning message and continues its execution.

4.2.2 Simple Resolver and Caching Resolver

The `SimpleResolver` fetches the component implementations and component prerequisite specifications from the *2K* Component Repository. It stores the component code in the local file system and dynamically links the components to the system runtime. As new components are loaded, they are attached to hooks in the component configurator of the parent component, i.e., the component that required it. In the web browser example, the `SimpleResolver` would add hooks to the web browser configurator, call them `FileSystem`, `TCPNetworking`, `WindowManager`, and `JVM`, and attach the respective component configurators to each of these hooks.

Resolvers can be extended using inheritance. For example, with very little work, we extended the `SimpleResolver` to create a `CachingResolver` that checks for the existence of the component in the local disk (cache) before fetching it from the remote repository.

4.3 Interactions with Resource Management Services

The QoS-aware Resource Management service can be provided by systems like SMART [NL97] and the Dynamic Soft Real-Time Scheduler (DSRT) [NhCN98]. In the latter case, DSRT can use the QoS requirements in the prerequisite specifications to perform QoS-aware admission control, negotiation, reservation, and scheduling.

Although our architecture permits the integration of the Automatic Configuration Service with any Resource Management scheme, a concrete implementation of this integration is beyond the scope of this thesis. This has been addressed by the research on the *2K* Resource Management Service [Yam00] and QoS Proxy [XWN00] carried out by other members of the *2K* team.

4.4 Use Cases

The Automatic Configuration Service has been used successfully in a multimedia distribution system [KCN00] described in Section 7.2. In addition, it is currently being used in the development of three other systems in our research group.

1. *2KFS*, the *2K* Distributed File System will use the Automatic Configuration Service to load data management components, called `Containers` [HBC00], to customize the system at runtime according to user requests.

2. LegORB, a component-based ORB, will use this service to load a minimal set of ORB components satisfying application requirements [RMKC00].

3. A QoS-aware video on demand system [XWN00] uses this service to load video player components on demand.

In all these applications, the Automatic Configuration Service simplifies management greatly. Whenever new components are requested, the service downloads the most up-to-date version of each component from the component repository and installs them locally. This provides several advantages including the following.

- It eliminates the need to upload components to the entire network each time a component is updated.

- It eliminates the need to keep track manually of which machines hold copies of each component because updates are automatic.

- It helps machines with limited resources, which no longer need to store all components locally.

Chapter 5

Component Configurators

As explained in Section 3.3, component configurators are responsible for holding information about the dynamic inter-component dependencies at runtime and implementing reconfiguration policies.

We begin this chapter by stressing the benefits of this model in achieving a clear separation of concerns. Next, we describe concrete implementations of component configurators in C++, Java, and CORBA. Finally, we discuss customization of component configurators.

5.1 Separation of Concerns

By combining the ideas of computational reflection [Smi84] and object-orientation, Kiczales created the concept of the *meta-object protocol* [KdRB91]. In his model, the programming language can be modified and extended by the user, who becomes a partner in language design. The protocol makes a distinction between base-level and meta-level objects. It uses meta-level objects, or *meta-objects*, to specify the behavior of the basic language operations and constructs. If the user does not alter the meta-level, the default meta-objects are used and the language is not altered. However, if the user specializes the meta-objects by using object-oriented inheritance, then the language behavior changes and can be customized to the application.

The meta-object protocol allows for a clean *separation of concerns* in the implementation of non-functional aspects of base-level objects. Aspects such as persistence, replication, and reconfiguration can be addressed by the meta-objects while the basic application objects are not modified.

Although this is an extremely flexible model that allows for a clean separation of concerns, previous implementations led to significant performance degradations since every method invocation

to an object was intercepted by its meta-object.

Kramer introduced the concept of *configuration programming* [Kra90], claiming that, to construct and manage distributed applications in a clean and effective way, it is necessary to adopt a *configuration language* independent of the component *implementation language*. He argued that the code for configuration should be separated from the component code.

Instead of relying on a different configuration language such as Darwin [MDK94, MTK97], this thesis proposes a novel model in which the basic functionality of the configuration engine is exported through a well-defined interface[1]. The actual configuration code is written in any language the developer chooses and is encapsulated in the component configurators. The latter can be reused in different contexts, but they can also be customized to each situation.

Using reflection terminology, component configurators are meta-objects, but with the advantage that their code is only executed when (re)configuration is required. They do not intercept every method call and, therefore, do not degrade application performance.

Our model is better suited for the highly heterogeneous environments of the future since the configuration engine can be defined using OMG IDL and then accessed, via CORBA, from any platform, using any programming language. Besides, the dynamic dependencies are accessible at runtime and, unlike in conventional configuration languages, can be inspected and modified at any time.

5.2 Implementation

Component configurators are normally dormant until they receive a request for reconfiguration or some other event notification. However, if the application developer desires, a certain component configurator can be active and have its own thread of execution. In this case, it could, for example, monitor system activity and send requests and event notifications to other configurators and components.

Depending on the way it is implemented, a component configurator is capable of referring to components running on a single address space, on different address spaces and processes, or even

[1] We define the configuration interfaces in C++ using abstract classes, in Java using the `interface` keyword, and in CORBA using OMG IDL.

running on different machines in a distributed system.

First, we describe a C++ implementation for a single address space. Next, we describe a Java implementation that was initially designed for a single address space but later extended to distributed systems using Java RMI. Finally, we describe a CORBA implementation for distributed systems.

5.2.1 C++

Figure 5.1 contains a simplified declaration[2] of a *ComponentConfigurator* base class in C++. Figure 5.2 shows a schematic representation of some of its method calls.

```
class ComponentConfigurator {
public:
 ComponentConfigurator(Object *implementation);
 ~ComponentConfigurator ();

 int addHook          (const char *hookName);
 int deleteHook       (const char *hookName);
 int hook             (const char *hookName, ComponentConfigurator *component);
 int unhook           (const char *hookName);
 int registerClient   (ComponentConfigurator *client, const char *hookNameInClient=NULL);
 int unregisterClient (ComponentConfigurator *client);

 int eventFromHookedComponent (ComponentConfigurator *hookedComponent, Event e);
 int eventFromClient          (ComponentConfigurator *client, Event e);

 char *name ();
 char *info ();
 DependencyList *listHooks ();
 DependencyList *listClients ();
 ComponentConfigurator *getHookedComponent (const char *hookName);

 Object *implementation ();
}
```

Figure 5.1: The *ComponentConfigurator* C++ Base Class

The class constructor receives a pointer to the component implementation as a parameter. It can be later obtained through the *implementation()* method.

The *addHook()* and *deleteHook()* methods are used to create and remove component configurator hooks. Each hook is identified by a string, representing its name.

[2]Complete implementations of the component configurator framework in C++, Java, and CORBA are available at `http://choices.cs.uiuc.edu/2k/ComponentConfigurator`.

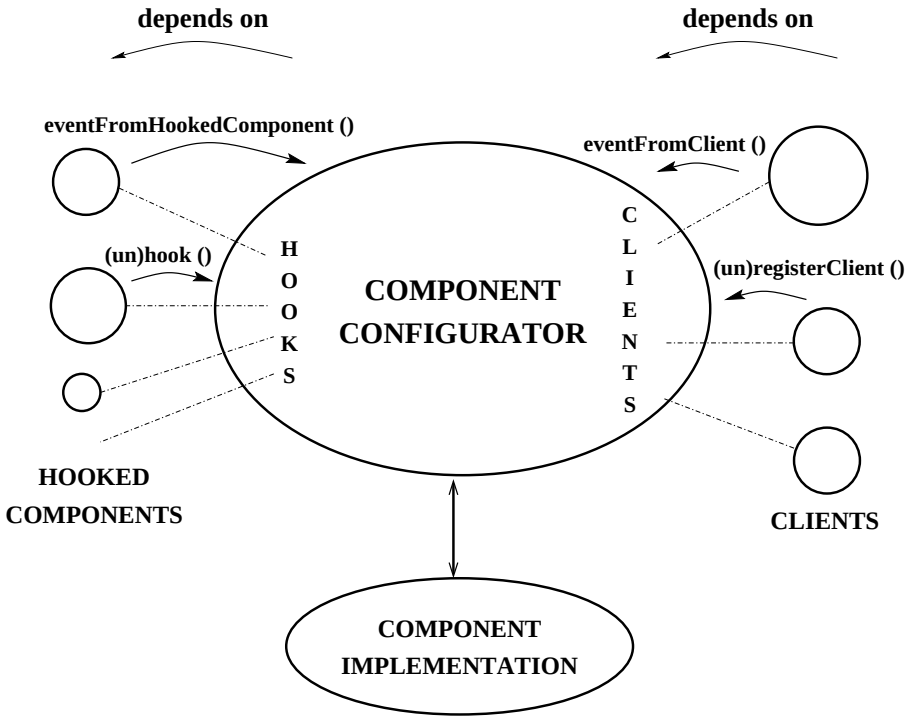


Figure 5.2: Methods for Specifying Dependencies and Sending Events

The *hook()* method is used to specify that this component depends upon another component and *unhook()* breaks this dependence. The *registerClient()* and *unregisterClient()* methods are similar to *hook()* and *unhook()* but they specify that other components (called clients) depend upon this component.

eventFromHookedComponent() announces that a component which is attached to this component has sent an event. The *ComponentConfigurator()* class is subclassed to implement different behaviors when events are reported. Examples of common events are the destruction of a hooked component, the internal reconfiguration of a hooked component, or the replacement of the implementation of a hooked component.

eventFromClient() is similar to the previous method but it announces that a client has sent an event. This can be used, for example, to trigger reconfigurations in a component to adapt to new conditions in its clients. Our reference implementation defines a basic set of events including *STARTED*, *FINISHED*, *SHUTDOWN*, *RECOVERED*, *RECONFIGURED*, *REPLACED*, *MIGRATED*, *DELETED*, and *FAILED*. Applications can extend this set by defining their own events.

name() returns a pointer to a string containing the name of the component and *info()* returns a pointer to a string containing a description of the component. Specific *info()* implementations can return different kinds of information like a list of configuration options accepted by the component, or a URL for its documentation and source code.

listHooks() returns a pointer to a list of *DependencySpecification*s. A *DependencySpecification* is a structure defined as

```
struct DependencySpecification {
 const char *hookName;
 ComponentConfigurator *component;
};
```

listClients() returns a pointer to a list of *DependencySpecification*s corresponding to the components that depend on this component (its clients) and the name of the hooks (in the client's *ComponentConfigurator*) to which this component is attached.

Finally, *getHookedComponent()* returns a pointer to the configurator of the component that is attached to a given hook.

5.2.2 Java

The Java implementation is very similar to the C++ one. Its declaration as a Java interface is shown in Figure 5.3. There are two minor differences with respect to the C++ version. The first is the *destroyComponentConfigurator()* method. Unlike C++, Java does not have explicit destructors. So, instead of calling the *delete* operator, as done in C++, Java programmers can request a component configurator to release its resources and prepare for its destruction by calling the *destroyComponentConfigurator()* method. Concrete implementations of the *ComponentConfigurator* interface can also choose to call this method on their implementation of the *finalize ()* method, which is called by the JVM garbage collector.

The second difference is the addition of exceptions. When we first implemented the C++ *ComponentConfigurator*, not every C++ compiler supported exceptions properly[3]. The C++ compilers that do support exceptions, generate a considerably larger code when exceptions are used, which

[3] Nowadays, all major C++ compilers do support exceptions.

```
public interface ComponentConfigurator {

  public void destroyComponentConfigurator ();
  public void addHook (String hookName) throws ElementExists;

  public void deleteHook (String hookName) throws NotFound;
  public void hook (String hookName, ComponentConfigurator cc)
     throws HookBusy, NotFound;
  public void unhook (String hookName)
     throws HookVacant, NotFound;

  public void registerClient (ComponentConfigurator client,
                              String hookNameInClient)
     throws ElementExists;

  public void unregisterClient (ComponentConfigurator client,
                                String hookNameInClient)
     throws NotFound;

  public void eventFromHookedComponent (ComponentConfigurator hookedComponent,
                                        ComponentEvent e);

  public void eventFromClient (ComponentConfigurator client,
                               ComponentEvent e);

  public String name ();
  public String info ();
  public Vector listHooks ();
  public Vector listClients ();
  public void name (String s);
  public void info (String s);
};
```

Figure 5.3: The *ComponentConfigurator* Java Interface

may be undesirable for small devices with limited memory. So, we decided not to add them to the base C++ class. Java compilers and virtual machines always support exceptions, so our Java *ComponentConfigurator* uses them.

5.2.3 CORBA

In CORBA, the ComponentConfigurator interface is defined using OMG IDL. Figure 5.4 shows a complete definition of the *Configuration* module which includes the *ComponentConfigurator* interface, definitions of dependencies and type structures, and a *Factory* interface. In this case, the interface can be implemented in any programming language and different implementations can communicate with each other seamlessly. We developed an implementation in C++ while other students in our department (Jennifer Jackson and Steve Zelinka) developed a Java version using the ORB that comes with the Java 1.2 distribution.

What is new here is the introduction of a factory for component configurators. The factory is a CORBA server that is responsible for creating component configurator objects that reside in the address-space in which the factory is located. Depending on where the programmer chooses to place the factories, the component configurators can be (1) co-located with their respective component implementations, (2) located in a separate process in the same machine or (3) located in a central node on the network while the component implementations are distributed. In all three cases, the communication with the component configurators is done exactly in the same way, via CORBA. No changes are required in the code to interact with local or remote component configurators.

In the CORBA implementation, a *DependencySpecification* stores a CORBA Interoperable Object Reference (IOR) so that the component configurator is able to reify dependencies among distributed components.

When a CORBA component is destroyed, the component implementation (or the ORB) must call the configurator destructor so that it can tell its clients that the destruction is taking place. If a node crashes or if the whole process containing both the component and the configurator crash, it might not be possible to execute the configurator destructor. In this case, the clients will not be informed of the component destruction. Subsequent CORBA invocations to the crashed component will raise an exception announcing that the object is not reachable or that it does not exist. This

```
module Configuration
{
  enum EventType {STARTED, FINISHED, SHUTDOWN, RECOVERED, RECONFIGURED, REPLACED,
                  MIGRATED, DELETED, FAILED, APP_SPECIFIC, UNKNOWN};
  struct Event
  {
    EventType type;
    string description;
  };
  interface ComponentConfigurator; // forward declaration
  struct DependencySpecification
  {
    string name;
    ComponentConfigurator componentConfig;
  };
  typedef sequence<DependencySpecification> DependencyList;

  interface ComponentConfigurator
    {
      exception invalidArgument{};
      exception ElementExists{};
      exception NotFound{};
      exception HookBusy{};
      exception HookVacant{};

      void destroy ();
      void addHook          (in string hookName) raises (ElementExists);
      void deleteHook       (in string hookName) raises (NotFound);
      void hook             (in string hookName, in ComponentConfigurator cc) raises (NotFound);
      void unhook           (in string hookName) raises (HookVacant, NotFound);

      void registerClient   (in ComponentConfigurator client,
                             in string hookNameInClient) raises (ElementExists);
      void unregisterClient (in ComponentConfigurator client,
                             in string hookNameInClient) raises (NotFound);
      void eventFromHookedComponent (in ComponentConfigurator hookedComponent,
                                     in Event e, in unsigned short timeToLive);
      void eventFromClient          (in ComponentConfigurator client,
                                     in Event e, in unsigned short timeToLive);

      string name ();
      string info ();
      DependencyList listHooks ();
      ComponentConfigurator getHookedComponent(in string hookName);
      unsigned short numberOfClients ();
      DependencyList listClients ();
      Object implementation ();
    };

  interface Factory
    {
      ComponentConfigurator newConfigurator (in string name, in Object implementation);
      oneway void shutdown ();
    };
};
```

Figure 5.4: The *Configuration* Module IDL interface

exception may be caught by the client component, which would be responsible for locating a new server component and updating its component configurator. Alternatively, the exception could be caught by an interceptor installed in the ORB, which could locate the new server and update the client configurator without the client's knowledge.

5.3 Customization

Our infrastructure includes a basic component configurator that provides simple implementations for the dependency management operations (e.g., *addHook()*, *registerClient()*) and a few subclasses implementing additional functionality such as mutual-exclusion (see Section 5.3.2).

Programmers can extend these basic classes in two ways. First, they can be customized to provide some generic functionality that could be reused by several applications in different contexts. This could include support for persistence, security, replication, and so on. In Section 5.3.4, we discuss how to customize a component configurator to support dynamic reconfiguration.

The second way of extending component configurators refers to application-specific extensions. For example, the programmer of a computationally-intensive distributed scientific application could include code to monitor local CPU load in the component configurator. The configurators, then, could exchange events with information about the CPU load on their machines and implement an algorithm for dynamic load balancing. The component would contain the code for carrying out the application task (scientific computation) while the component configurator would encapsulate the policies for load balancing.

We now discuss some examples of component configurator customization that are particularly interesting.

5.3.1 An Application-Specific Customization

We present here a simple example of an application-specific customization. A detailed example of a customization to support fault-tolerance in a QoS-sensitive application is presented in Section 7.2.

The dynamic dependence information enables the reconfiguration of components that are already running. Although our infrastructure does not guarantee safe reconfiguration by itself, it does provide a valuable framework for programmers to implement safe reconfiguration more easily

and uniformly.

Continuing with our web browser example (introduced in Section 4.2), the application developer could implement a *WebBrowserConfigurator* by using inheritance from the *ComponentConfigurator* class and customizing it to deal with the dynamic replacement of the system's JVM. As shown in Figure 5.5, the *eventFromHookedComponent* method can be overridden to catch *REPLACED* events coming from the JVM *ComponentConfigurator*.

```
int WebBrowserConfigurator::eventFromHookedComponent (ComponentConfigurator *cc, Event e)
{
 if (cc == JVMConfigurator)
  {
   if (e == REPLACED)
    try {
     FrozenObjs fo = currentJVM->freezeAllObjs ();
     currentJVM = JVMConfigurator->implementation ();
     currentJVM->meltObjects (fo);
    }
    catch (Exception exp)
      throw ReconfigurationFailed(exp);
  }
 else ...
}
```

Figure 5.5: Customization of the *eventFromHookedComponent* Method

When the implementation of the JVM is updated, the *JVMConfigurator* sends a *REPLACED* event to its clients. When the *WebBrowserConfigurator* receives this event, it freezes all the objects in the current JVM (using the Java object serialization mechanism), updates the current JVM with the new JVM implementation, and melts the objects in the new JVM[4].

5.3.2 Locking Configurator

The most basic version of the C++ component configurator we implemented is called *SimpleConfigurator*. It does not provide any support for controlling concurrent accesses to the configurator by multiple threads. To support that, we implemented a subclass called *LockingConfigurator*.

LockingConfigurator uses two locks. The first controls read and write accesses to the dependencies and the second, to the component implementation. Both permit either multiple concurrent readers (and no writers) or a single writer.

[4]Since the Java serialization mechanism does not record thread execution state, this must be performed with mechanisms like the ones presented in [Bou99a, Bou99b] and [Fun98].

The *listHooks()* method, for example, is implemented by first acquiring the dependencies lock in reading mode, then calling the equivalent method in the super class, and then releasing the lock.

```
inline DependencyList *listHooks ()
   {
     dependenciesLock_.acquire_read();
     DependencyList *ret = SimpleConfigurator::listHooks();
     dependenciesLock_.release ();
     return ret;
   }
```

The *registerClient()* method, on the other hand, acquires the lock in writing mode.

```
inline int registerClient (ComponentConfigurator *client, const char *hookNameInClient)
  {
    dependenciesLock_.acquire_write();
    int ret = SimpleConfigurator::registerClient(client, hookNameInClient);
    dependenciesLock_.release ();
    return ret;
  }
```

The *implementationLock* can be obtained through an accessor method. In that way, programmers can lock the implementation of a component and be sure that it will not change while it is locked.

5.3.3 Dependency Attributes

After some experience with the configurators, we noticed that, in some applications, it is desirable to assign attributes to dependencies. In that way, we can have different types of dependencies. Dependency attributes can also be used to store more information about the characteristics of that dependency. Developers can program policies that treat dependencies differently based on their attributes.

Consider, for example, a File Server that has two types of clients. The first are simple client programs activated by users. The second are backup storage devices that simply read everything from the File Server from time to time, saving its contents into tape. When the File Server is shut

down by the administrator, its configurator may need to send an event to the backup devices so that its content can be copied. In this case, the shut down process should be delayed until all the server contents is backed up to tape. By using attributes, the File Server configurator can differentiate between the clients that need to be notified synchronously (backup devices) and clients that do not need to be notified (user programs).

Figure 5.6 shows the Java *ComponentConfiguratorAttrib* interface that inherits from the basic *ComponentConfigurator* interface, adding support for attributes. *DependencyAttributes* is a list of attributes; each attribute is defined as a <*name,value*> pair, where *name* is a string and *value* is any Java object.

```
public interface ComponentConfiguratorAttrib extends ComponentConfigurator
 {
  public void addHook (String hookName, DependencyAttributes attributes)
    throws ElementExists;

  public void hook (String hookName, ComponentConfigurator cc,
                    DependencyAttributes attributes)
    throws HookBusy, NotFound;

  public void registerClient (ComponentConfigurator client, String hookNameInClient,
                              DependencyAttributes attributes)
    throws ElementExists;
 };
```

Figure 5.6: The *ComponentConfiguratorAttrib* Interface Supporting Attributes

Our implementation of the *ComponentConfiguratorAttrib* interface was used in a decentralized information system [ESS+00] described in Section 5.4.

5.3.4 Supporting Consistent Dynamic Reconfiguration

In a general sense, when replacing an old component by a new one it may be necessary to transfer the state from the former to the latter. This process can be automated by the reconfiguration engine by requiring every component to implement a pair of operations *get_state()* and *set_state()*. All the components of a certain type must then agree on a common external representation for the state of the components of that type. Then, the underlying engine simply transfers the state from one component to the other, without having to interpret its meaning.

To replace a component and remove the old version safely, one must make sure that no other

component will try to contact the component being removed. This can be achieved by using a combination of the following four mechanisms.

1. Using the *ComponentConfigurator* to notify all the components that have a reference to the old one.

2. Using the *ComponentConfigurator* as an indirection on calls to replaceable components.

3. Leaving a forwarding pointer in place of the old component.

4. Making every access to the old component throw an exception to be captured by the client. In this case, the client gets a reference to the new component by contacting a third party such as the Naming Service.

5. Keeping the old versions accessible to old client components and redirecting new clients to the new version. When the reference count in the old version reaches zero, it can be removed safely.

It is possible to implement these mechanisms in a specialized subclass of the *ComponentConfigurator*. This can be achieved with an implementation of the ideas that Bloom and Day discuss in [BD93], providing a more sophisticated support for dynamic reconfiguration at the middleware level and requiring less application participation. Different combinations of these mechanisms can be used in different parts of a single system.

5.4 Use Cases

The *ComponentConfigurator* model described in this chapter has been deployed in the following systems.

1. *dynamicTAO* uses the model to represent the dependencies among the internal components of a CORBA ORB and support dynamic reconfiguration (see [KRL+00] and Section 7.1).

2. SIDAM, a decentralized information system for disseminating road traffic information using mobile agents [SGE98]. In this system, Menezes used customized component configurators to manage dynamic configuration of `LocatorServers` implemented using Java RMI. In this

system, the `LocatorServers` objects are responsible for telling clients where they should get their information. When `LocatorServers` are removed from the system, the component configurators act to reconfigure the system automatically to maintain consistency.

According to experimental results, the use of the component configurator for dependence management in this system imposes an overhead of 10% at initialization time. After the initialization is completed, the component configurators only act when reconfiguration is required and, therefore, impose no overhead during normal system execution [ESS+00].

3. A CORBA Persistent Object Service implemented by Arjun Iyer as a Masters project [Iye99]. Iyer used component configurators to represent the dependencies among his `DataStore`, `PersistentDataStore`, `PersistentObjectManager`, and `PersistentObject` components. Although Iyer did not implement support for dynamic reconfiguration and fault-tolerance, his thesis discusses some of the issues involved in implementing these functionalities with the component configurators.

4. A Distributed Chess game implemented by Colin Jones as a class project for CS423. The CORBA component configurators were used to manage the dependencies among the different components of the system: robot players, human players, game observers, game logger, and the central chess server.

In addition to the completed systems described above, our model is being deployed in the following ongoing projects.

1. LegORB, a component-based CORBA ORB [RMKC00] targeted at embedded systems and devices with little resources. In contrast to full ORB implementations that require, typically, a few megabytes of memory, LegORB loads only the minimal set of components required to provide the functionality required by each application. A minimal LegORB configuration using component configurators to manage dynamic reconfiguration requires less than 150 Kbytes of memory.

2. A system for managing network-centric user environments for the *2K* operating system [CKB+00]. The model is used to keep references to user applications in the distributed system, managing them properly.

Chapter 6

Reconfiguration Agents

Management of software configuration in large scale networks is a major problem in modern computer systems. Scalability conflicts with dynamic flexibility since it is difficult to manage dynamic changes in a large number of machines concurrently. Scalability is a topic that received very little attention from previous research in dynamic configuration.

Internet services such as video on demand, on-line bookstores, and search engines must maintain a high level of parallelism in order to fulfill the millions of requests they receive every hour. World-wide services with high-bandwidth requirements such as on-line sales of software must be readily available in many different countries. Services must be replicated or split across machines in different locations, optimizing bandwidth and CPU utilization, and minimizing user response time.

Since software evolves rapidly and availability is a major requirement in these systems, the natural solution to cope with all these requirements is to use dynamic reconfiguration. The problem is that reconfiguring and updating the software in a distributed collection of servers using conventional tools is troublesome. Besides, it may lead to wasted bandwidth as duplicated information is sent through the same Internet lines.

A significant limitation of the traditional administrative tools for reconfiguration is their scalability. In existing systems supporting dynamic reconfiguration, the administrator must establish a point-to-point connection between the administration node and the process that is going to be reconfigured. Thus, if a system administrator needs to upgrade a certain component of an on-line service composed of ten replicas located in different continents, it is necessary to connect to each replica separately, upload the new implementation of the component, and reconfigure the replica. As the latency in long-distance Internet connections tends to be high, this process is extremely

laborious and tiresome.

The solution we propose is 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 nodes. Each node replicates and forwards the agent to neighboring nodes, processes the reconfiguration commands locally, and collects the reconfiguration results, sending them back to the neighboring agent source. Agents can carry not only reconfiguration and inspection commands, but also the code for new components to be distributed throughout the network.

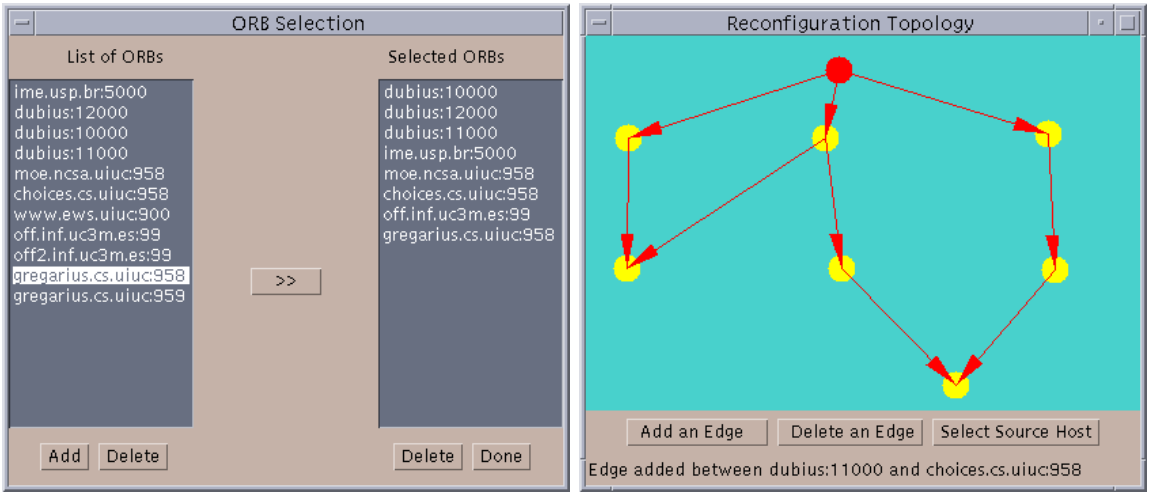
Using this approach, the administrator can organize the reconfiguration hierarchy to optimize the data flow between distant application nodes. The reconfiguration commands are executed in parallel in the various nodes, improving response time. If desired, the graph may contain different levels of redundancy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

6.1 Implementation

Our implementation is based on the *dynamicTAO* reflective ORB which is described in detail in Section 7.1. *dynamicTAO* exports two interfaces for dynamic reconfiguration: an IDL interface that can be used by CORBA clients and a simple ASCII interface based on commands send over TCP/IP. The latter can be accessed via simple telnet connections and uses the commands specified in the Distributed Configuration Protocol (DCP) (see Section 7.1 and [Kon98]). DCP includes commands for inspection, reconfiguration, and code distribution.

To support agents, *dynamicTAO* was enhanced with mechanisms for receiving, processing, and forwarding agents. The system also includes *ConfigAgentBuilder*, a graphical administrative front-end, written in Java, for specifying reconfiguration graphs and for assembling and sending reconfiguration agents [KCS+99].

The administrator selects those ORBs that will be part of the reconfiguration graph (see Figure 6.1(a)) and, using the mouse, draws directed edges connecting the graph nodes (see Figure 6.1(b)). Each time a new ORB is selected from the list on the left-hand side of Figure 6.1(a), a new node



(a) Selecting the graph nodes

(b) Creating the graph edges

Figure 6.1: The *dynamicTAO ConfigAgentBuilder* Interface

is added to the graph in Figure 6.1(b).

Once the reconfiguration graph is established, a graphical interface assists the administrator in building a script of DCP commands that are codified into a reconfiguration agent. Finally, the administrator instructs the graphical front-end to send the agent to an initial node in the graph. Figure 6.2(a) shows the composition of an agent with three DCP commands: `list_categories`, `list_loaded_implementations`, and `list_implementations`.

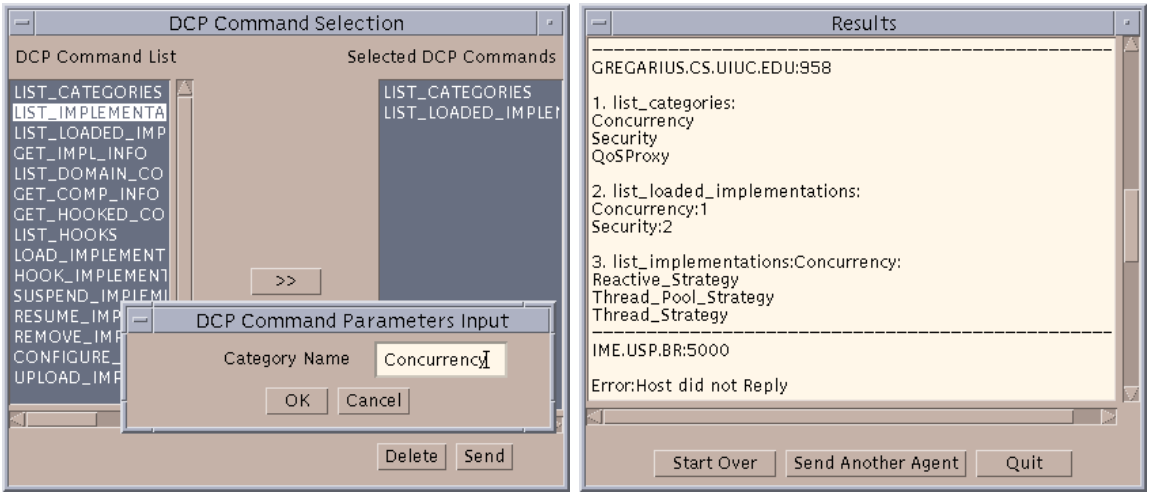
Each agent contains

1. a copy of the reconfiguration graph (so that different agents can operate on different graphs at the same time),

2. a script of DCP commands to be interpreted on each node in the graph, and

3. a unique sequence number which is used to avoid processing duplicated copies of the same agent on the same node.

Upon receiving an agent, the ORB first checks whether it has been processed before by looking at its local history of processed agents. If the agent has not been processed, it distributes the agent to its neighbors (according to the reconfiguration graph included in the agent), processes the DCP commands locally, and waits for the results of the agents sent to its neighbors. The ORB, then,



(a) Composing an agent

(b) Information collected by an inspection agent

Figure 6.2: Sending and Receiving Agents

collects the replies from its neighbors, groups them with the results of its own reconfiguration, and sends all of the information back to the node from which it received the agent. Each node waits for the results from its agents for a certain period of time. A timeout is computed taking into consideration the number of nodes the agent has already traversed.

The results returned by all the nodes in the reconfiguration graph are finally displayed to the administrator through the graphical front-end. The system also draws attention to the nodes that did not reply within a given timeout. Figure 6.2(b) shows the results returned by an agent containing three DCP inspection commands (also called an *inspection agent*). Our agent dissemination system can be seen as a simple, lightweight implementation of reliable multicast [Obr98] over TCP/IP. We intentionally adopted this simple and reliable solution for agent transmission because it lets us concentrate on the dynamic configuration aspects of the problem. However, depending upon the topology of the reconfiguration network (for example, if the nodes have many output edges), it may be worthwhile to use other underlying protocols such as IP-Multicast [DC90] for distributing agents. In that case, however, the complexity of the middleware would be much higher since it would have to guarantee reliability over a non-reliable protocol.

6.1.1 Java Agents

Our implementation supports agents with simple reconfiguration scripts based on the DCP protocol. On the one hand, this allowed us to write an extremely lightweight interpreter that processes the agents efficiently. This is very appropriate for PDAs and embedded systems with limited resources. On the other hand, it limits the expressiveness of the reconfiguration agents. By using a more powerful language for reconfiguration such as OCL [BBB+98] or a general-purpose language like Java or Emerald [JS95], the administrator would be able to write more sophisticated agents that could use the agent and ORB state to make decisions about their actions. To achieve that, we designed an infrastructure for Java agents that was implemented by Masters students in our group [KGA+00].

To support this implementation, we encapsulated Sun's Java Virtual Machine into a component that can be dynamically loaded into the *dynamicTAO* domain; this component exports an interface that contains operations for loading and interpreting Java bytecode. In addition to the reconfiguration topology graph, that now can be modified on-the-fly, the Java agent carries its dynamic state from node to node and contains Java bytecode that is interpreted inside a sand-box so that its actions can be limited. By using the *DynamicConfigurator* IDL interface (see Section 7.1.2), the Java bytecode issues inspection and reconfiguration commands to the ORB domain. Based on the result of these commands, the agent can

1. update the state it carries from node to node,
2. change its behavior to adapt to the environment and user preferences on each node, and
3. modify the reconfiguration graph, deciding to migrate to a different set of nodes.

In that manner, our framework offers both simple agents based on DCP scripts for environments with limited resources and powerful Java agents that can carry state and modify its behavior along the way.

6.2 Security

Although the introduction of mobile agents promotes significant benefits, it also brings additional security concerns [Vig98]. Luckily, the problem of mobile agents security has been studied extensively in recent years and good results are starting to appear. When using mobile agents for dynamic configuration, it is important to

1. restrict the sources of mobile code to trusted, authenticated entities [KGA+00, KGCM99],
2. limit the mobile code's resource consumption [BKR98], and
3. limit its capabilities by confining it to secured environments like sand boxes for Java byte code [Yel95] and the Janus sand box for native code [GWTB96].

[VB99] shows how to isolate agents from one another. However, another important aspect, the protection of the mobile agents from the execution environments, is still mostly unresolved. One can find partial solutions in approaches like time-limited black-box security [Hoh98] and the clueless agents [RS98] concept. The approach adopted in our group is to establish secure connections between the nodes distributing the agent using a role-based access control framework. The details are beyond the scope of this thesis and are described in [KGA+00] and [KGCM99].

6.3 Fault-Tolerance and Consistency

Our experience with distributed computer systems has shown that a wide range of applications do not require strict consistency between the components in the different nodes. When deploying the multimedia distribution system described in Section 7.2, for example, we were able to use different versions of application components in different nodes simultaneously. In many cases, even if a particular component update was successful in part of the nodes and failed in another part, the distributed system could continue working together gracefully until all the updates were achieved manually.

To keep the system small and efficient, we opted not to provide strong guarantees about the execution of the reconfiguration commands in the current implementation. On a single node, some reconfiguration commands may fail while others succeed; it is the responsibility of the agent to

catch the exceptions raised by the failed commands and treat them accordingly. Also, reconfiguration commands may succeed on some nodes and fail on others. To cope with these errors, the system detects the failures and the graphical front-end (shown in Figure 6.2(b)) displays, to the administrator, where they occurred and the respective diagnosis messages. Then, using a separate Java GUI developed by Luiz Claudio Magalhães and called *Doctor* (the Dynamic ORB Configuration Tool in Figure 6.3), the administrator can connect to the node where the failure occurred, inspect its state and reconfigure the node manually. By navigating through ORB and application components interactively, the administrator can investigate the causes of the errors, and fix them.

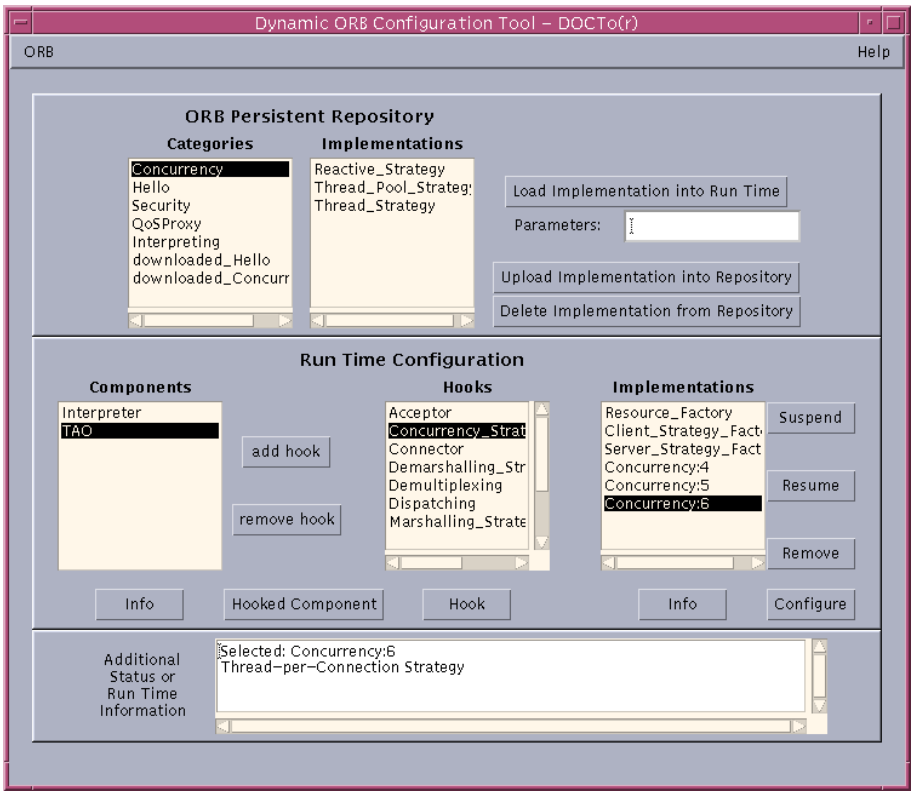


Figure 6.3: The *Doctor* Configuration Tool

On the other hand, it is possible to achieve the effects of atomic transactions using the functionality we are offering through the Java agents. On an individual node, the Java agent can be programmed to store the previous state of the system before starting to make any changes. Should an error occur during the reconfiguration process on that node, the Java code can detect the error

messages and bring the configuration back to the initial state[1].

To achieve the effects of atomic transactions on the distributed system as a whole, one can use a different technique. According to our experience, reconfiguration failures happen, typically, when a new component is loaded (and the dynamic loader cannot link it) or when the new component is initialized (and it fails to locate the functions or allocate the resources it needs). Thus, we instruct the administrator to separate its reconfiguration in two phases resembling the two-phase commit protocol for atomic transactions [GR93]. First, the administrator sends an agent to load all the required components and to initialize them by executing the operation *load_implementation()* (and, if needed, *configure_implementation()*). If an error in any node occurs, the administrator chooses between sending a new agent to unload the new components from all the nodes or using *Doctor* to fix the individual errors. If the component implementations are loaded and initialized successfully in all the nodes, the administrator sends a new agent to attach them to the proper applications and ORB components, which would correspond to the *commit* phase of the two-phase commit protocol.

We found this to be sufficient for most of the application scenarios with which we work. However, some other applications require more strict guarantees of the ACID properties for atomic transactions [GR93]. For example, if we want to reconfigure the marshalling and unmarshaling components in all the ORBs in a distributed system, it is important to guarantee an *all-or-nothing* property in the transactions. If an ORB replaces its marshalling mechanism and one of the ORBs to which it sends requests does not change its unmarshaling mechanism, they might no longer be able to communicate.

To support that, one could extend our infrastructure by using standard transaction mechanisms for distributed systems [EMS91] and recent protocols for mobile agent fault-tolerance [SPZ98]. The deployment of these mechanisms for the reconfiguration of active components is not straightforward. It is an interesting research topic. In this case, the atomic transactions must deal with active objects (i.e., data + code + running threads) rather than simply data, as is the case with more traditional uses of atomic transactions such as file systems [SW91] and distributed databases [GR93].

[1]Depending on the components being reconfigured, it may be necessary to save the component internal state before the configuration occurs so that it can be restored in case it is necessary to roll back to the initial state. This can be achieved by requiring each component to implement operations for capturing and restoring its internal state as described in Section 5.3.4.

Chapter 7

Application Scenarios

The work on this thesis was motivated initially by our research on the *2K* operating system [KCM+00]. During the early phases of that research, we identified proper dependence management as a key factor in the development of robust operating system and middleware architectures. We begin this chapter describing *dynamicTAO*, a fundamental part of *2K*, and explaining how the *dynamicTAO* internal architecture applies the ideas defended in this thesis.

Section 7.2 focuses on a Multimedia Distribution System we developed some years ago. It shows how we enhanced the system recently using the infrastructure for automatic configuration and component configurators to enable dynamic reconfiguration in the occurrence of failures.

In Section 7.3, we refer to systems being developed by other researchers that are applying some of the ideas presented in this thesis.

7.1 *dynamicTAO*

One of the major constituent elements of the *2K* distributed operating system [KSC+98, CNM98, KCM+00] is a reflective middleware infrastructure based on CORBA. After carefully studying existing CORBA Object Request Brokers, we came to the conclusion that the TAO ORB [SC99] would be the best starting point for developing this infrastructure.

TAO is a portable, flexible, extensible, and configurable ORB based on design patterns. It uses the *Strategy* design pattern [GHJV95] to encapsulate different aspects of the ORB internal engine. A configuration file is used to specify the strategies the ORB uses to implement aspects such as concurrency, request demultiplexing, scheduling, and connection management. At ORB startup

time, the configuration file is parsed and the selected strategies are loaded.

TAO is primarily targeted for static hard real-time applications such as avionics systems [HLS97a]. Thus, it assumes that, once the ORB is initially configured, its strategies will remain in place until it completes its execution. There is no support for on-the-fly reconfiguration.

On-the-fly adaptation is extremely important for a wide range of applications including those dealing with multimedia, mobile computers, and dynamically changing environments. To support this kind of dynamic adaptations, we developed *dynamicTAO* [RKC99, KRL+00], an extension of TAO that enables on-the-fly reconfiguration of its strategies. *dynamicTAO* exports an interface for loading and unloading modules into the ORB runtime, and for inspecting the ORB configuration state. The architecture can also be used for dynamic reconfiguration of user applications running on top of the ORB and even for reconfiguring non-CORBA applications.

7.1.1 A Reflective ORB

dynamicTAO is our first complete implementation of a CORBA reflective ORB. As pointed out in [SSC97, SSC98b], 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 the execution of the remote method invocation. A reflective ORB makes it possible to redefine its evaluation semantics.

dynamicTAO allows inspection and reconfiguration of its internal engine and allows ORB and application developers to specify reconfiguration policies inside customized subclasses of the *ComponentConfigurator* class. It exports 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 reification of *dynamicTAO*'s internal structure is achieved through a collection of component configurators. 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`.

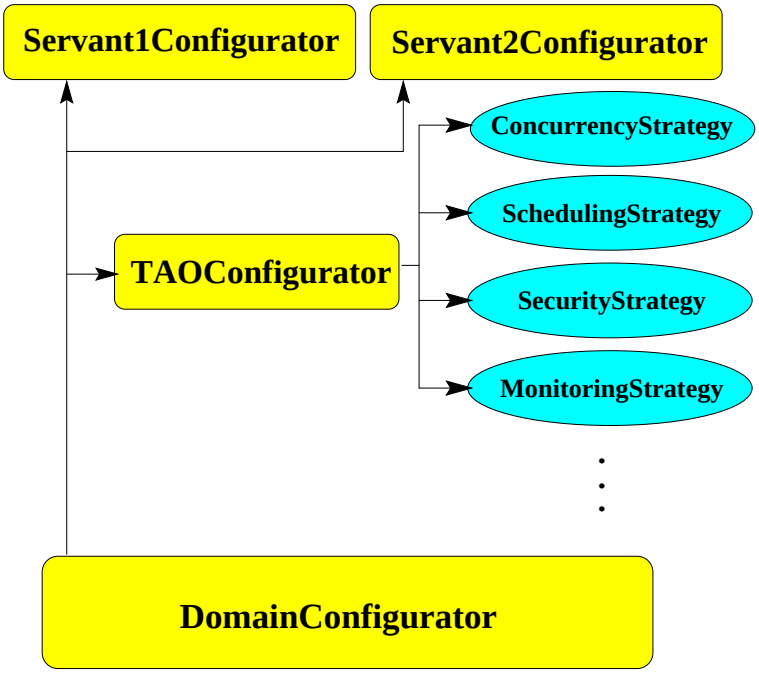


Figure 7.1: Reifying the *dynamicTAO* Structure

`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, and Monitoring. The association between hooks and component implementations can be changed at any time, subject to safety constraints.

Figure 7.1 illustrates this reification mechanism in a process containing a single instance of the ORB. If necessary, individual strategies can use component configurators to reify 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 7.1.3.

The *dynamicTAO* architectural framework is depicted in Figure 7.2. 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.

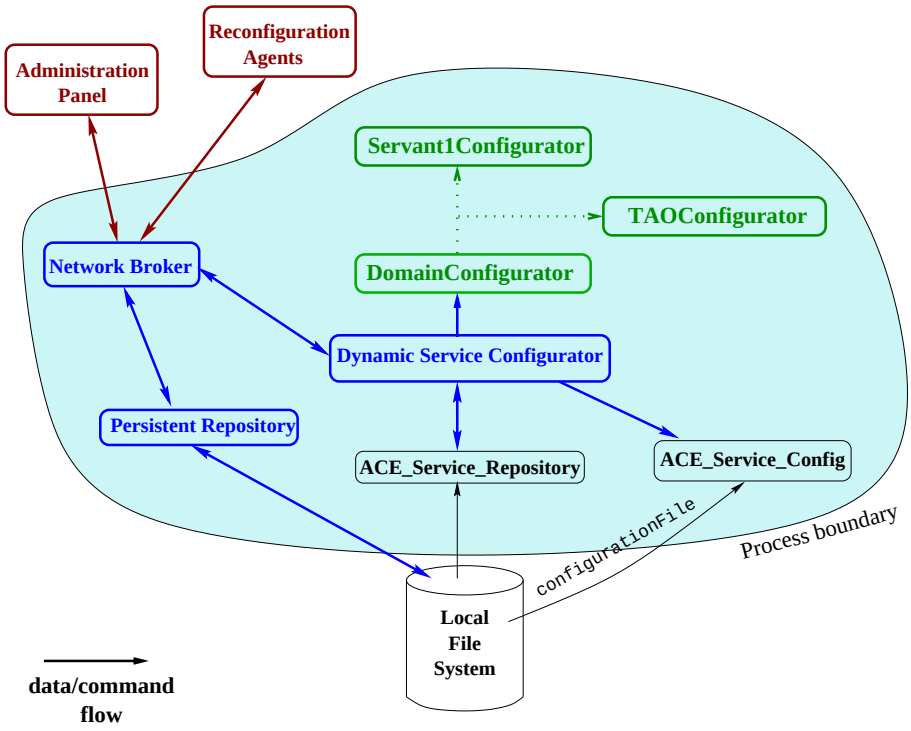


Figure 7.2: *dynamicTAO* Components

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 7.1) 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) [JS97].

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.

We built the *dynamicTAO* components using the ACE wrappers [Sch93] for operating system

services. Thus, *dynamicTAO* runs on the several different platforms to which ACE was ported.

7.1.2 Reconfiguration Interfaces

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 7.2. 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 [Kon98]. 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 described in Chapter 6.

3. The **DynamicConfigurator** is a CORBA object that exports an IDL interface with operations equivalent to those 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 7.3 to explain the functionality of the *dynamicTAO* reconfiguration interfaces[1].

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`

[1]To make Figure 7.3 more clear, we omitted the exceptions that each operation can raise.

```
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);
};
```

Figure 7.3: The *DynamicConfigurator* Interface

contains the three threading models 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 7.1.3, we discuss what happens if a different concurrency strategy is in use.

Figure 7.4 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 *Dynamic-Configurator* object through the ORB's *resolve_initial_references()* method.

```
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);
```

Figure 7.4: Inspecting the ORB Internal State

7.1.3 Consistency

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

Consider the case in which *dynamicTAO* receives a request for replacing one of its strategies (S_{old}) by a new strategy (S_{new}). TAO strategies are implemented as C++ objects that communicate through method invocations. The first problem is that, before unloading S_{old}, the system must guarantee 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 strategy, 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 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 [GHJV95] 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.

7.1.4 *dynamicTAO* and this Thesis

It is interesting to note that, although the implementation of a reflective ORB is not the main topic of this thesis, *dynamicTAO* is associated with the novel ideas presented in this thesis in many distinct ways.

The *dynamicTAO* internal structure is reified and managed with the C++ implementation of the component configurator framework. Our implementation of the CORBA version of the component configurator was developed using *dynamicTAO* as the underlying ORB. Finally, we developed the Component Repository and the Automatic Configuration Service using *dynamicTAO*'s *DynamicConfigurator* IDL interface depicted in Figure 7.3.

dynamicTAO is a good example of how the component configurator framework can be used to represent inter-component dependence and how software developers can use it to specify policies for safe dynamic configuration.

7.2 Scalable Multimedia Distribution

In [KCT+98] we describe the design and implementation of a flexible multimedia distribution system and demonstrated that it is possible to use the existing Internet to distribute low and medium bandwidth multimedia to thousands of simultaneous users. Our early experiments, however, pointed out to difficulties in managing such a large-scale system and keeping it available with an acceptable quality of service. It showed the necessity for a better support for dynamic reconfiguration, distribution of code updates, and provision of fault-tolerance for QoS-sensitive applications.

We addressed these problems in a new version of the multimedia distribution system [KCN00] built on top of the architecture presented in this thesis.

7.2.1 The Reflector

The multimedia distribution system's key element is the *Reflector*. It acts as a relay, receiving input data packets from a list of trusted sources and forwarding these packets to other Reflectors or to programs executed by end-users (also called *end-user clients*). The distribution system is composed of a network of Reflectors that collaborate with each other to distribute the multimedia data over local-area, metropolitan-area, and wide-area networks.

The Reflector is a user-level program that, at the application level, performs activities similar to those performed by network routers at the hardware level (which is where protocols such as IP-Multicast [Dee89] are implemented). Since it is implemented in software, not hard-wired into the router, the Reflector is more flexible, easy to deploy and evolve, and can be dynamically customized to users, applications, and environments.

Reflector data packets are encoded with RTP, a user-level protocol for real-time applications [SCFJ00] defined by the Internet Engineering Task Force (IETF). RTP packets can be transmitted over different kinds of low-level transport protocols such as TCP/IP, UDP/IP, and IP-Multicast.

The Reflector Network topology is determined by each Reflector's configuration. This information specifies input and output connections, access privileges, maximum allowed number of users, etc. It is stored in a database controlled by the Reflector administrator. Figure 7.5 depicts a generic Reflector network that distributes two video streams in different channels. In this figure, two capture stations send their video streams to "master" Reflectors; the streams may traverse

several "intermediate" Reflectors until they reach "public" Reflectors to which end-user clients can connect and receive the video streams[2]. All the Reflectors in the figure are initiated with exactly the same code. Using the Automatic Configuration Service described in Chapter 4, the system then customizes each Reflector according to their individual requirements.

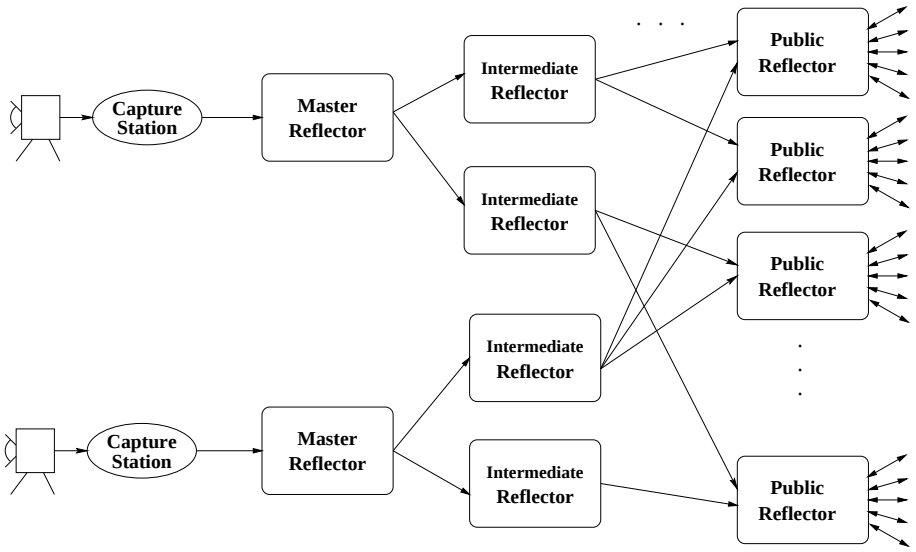


Figure 7.5: A Reflector Network Distributing Two Video Streams

7.2.2 Data Distribution Protocols

In order to support different types of inter-reflector communication protocols transparently, the Reflector framework encapsulates the concept of a network connection into an abstract C++ class named `Connection` that defines the basic interface for all types of network connections used by the Reflector. This abstract class implements some of its own methods, but the majority of the connection-related code is implemented by its subclasses: `TCPConnection`, `UDPConnection`, `MulticastConnection`, and the like.

Figure 7.6 depicts a concrete example of a highly-heterogeneous Reflector network. In this example, the network distributes two audio-visual streams. The first comes from a mobile camera, mounted on a helicopter, that sends its stream to a "master" Reflector over a wireless link. This kind of link presents a high rate of packet loss not related to congestion, which makes protocols like TCP perform poorly [BPSK96]. Thus, it is desirable to use a protocol optimized for this

[2]The classification of Reflectors as "master", "intermediate", and "public" Reflectors is merely pedagogical, they execute exactly the same program.

kind of situation like, for example, WTCP [SVSB99]. The second stream is sent to its "master" Reflector through a dedicated ISDN line. To optimize the bandwidth, one can use UDP as the communication protocol since its overhead is lower than that of TCP and the link offers a low loss rate.

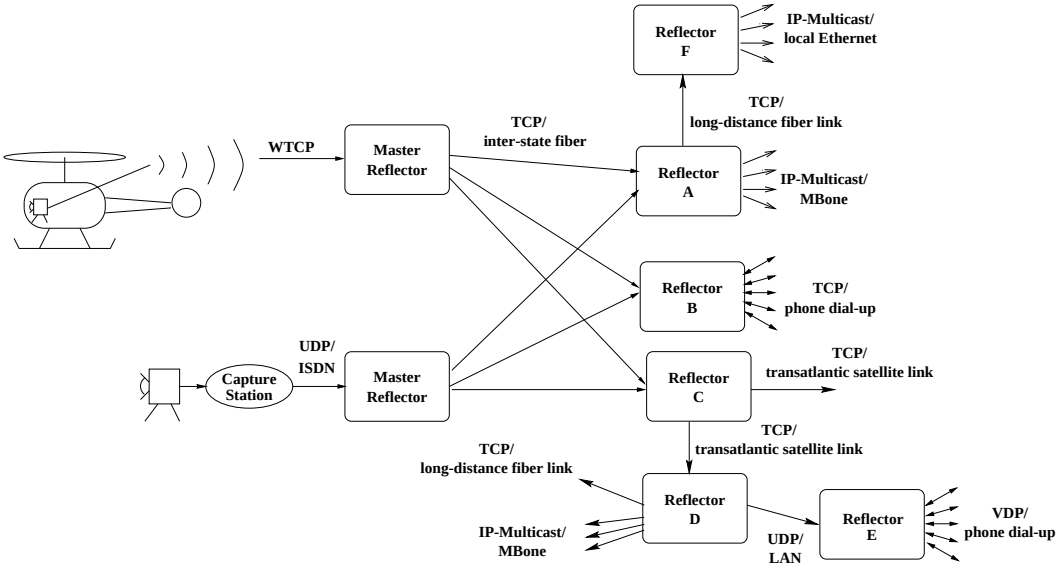


Figure 7.6: A Heterogeneous Reflector Network

Continuing with the example, Reflector C sends its streams to Reflector D over the public Internet through a transatlantic satellite link. Even though this is a high-bandwidth link, its loss rate may be high, so it is more appropriate to use TCP. Reflectors B and E offer the multimedia streams to dial-up clients over conventional phone lines; Reflector B uses TCP while Reflector E, on the other side of the Atlantic, uses the VDP adaptive algorithm. Finally, Reflectors A and D introduce the video streams into two distant points of the global MBone and Reflector F uses multicast to distribute the streams in a local Ethernet network.

Selecting the appropriate protocol for each situation, the Reflector administrators improve the quality of service offered to the end-users, optimizing the utilization of the available network bandwidth and minimizing packet losses.

Most of the Reflector's code deals with objects of type `Connection` and is not aware of the `Connection`'s underlying implementation. The actual connection type is specified when the connection is created and does not need to be known after that.

This approach allows programmers to plug in customized `Connection` subclasses by providing

their own implementation of the `Open`, `Close`, `Bind`, `Connect`, `Send`, and `Receive` methods, adding specialized functionality.

In this manner, it is possible to incorporate, into the Reflector, `Connection` subclasses that implement different transport protocols (such as the VDP QoS-aware adaptive protocol for the Internet [CTCL95] and the xbind QoS-aware reservation protocol for ATM networks [CL99]). Developers also use this mechanism to implement `Connection` subclasses that perform various operations on the data, such as encryption, transcoding, mixing, downsampling, and the like). Finally, one can create composite `Connection` types by combining existing ones. For example, one can create a `CryptoMulticast` connection type – that encrypts the data and sends them out using Multicast – by combining a `Crypto` connection with a `Multicast` connection.

7.2.3 Experience and Lessons Learned

This technology was utilized in the live broadcast of NASA's JPL Pathfinder mission [GCE+97]. During this broadcast – which lasted for several months – more than one million live video sessions were delivered to dozens of different countries across the globe by a network of more than 30 reflectors spread across five continents. The Reflectors ran in five different operating systems (Solaris, Linux, Irix, FreeBSD, and Windows) and transmitted their streams over different kinds of network links. End-users could watch the video stream simply by pointing their web browsers to selected locations, causing their browsers to download a Java applet containing the video client. The applet connected automatically to the reflector, received the video stream, decoded it, and displayed it to the user in real-time.

During this broadcast, we experienced three major problems:

1. As the code had not been tested on such a large scale and on so many different platforms, we found many programming errors both in the Reflector code and in the client applet. Surprisingly, fixing the error was, sometimes, easier than updating the code in the dozens of machines that formed the distributed system. The same problem occurred when a new version of the Reflector, with added functionality, was released. System administrators had to manually connect to dozens of machines, upload the new code, shutdown the old version, and start the new one.

2. In many cases, we had to reconfigure the reflector network by dynamically changing the distribution topology, or by setting new values to the reflector configuration parameters (e.g., maximum number of users, number of multimedia channels, etc.). The configuration information for the reflectors was stored in a centralized location. After updating this centralized database, we had to connect to each of the reflectors and instruct them to download their updated configuration. This process was tiresome and error-prone.

3. The only mechanism the Reflector provided to support fault-tolerance was to send redundant streams from different sources to the same Reflector (see [KCT+98] for details). This mechanism leads to a large waste of bandwidth. The redundant streams are always transmitted even though they are seldom used.

With this experience, we learned that a user-level Reflector system is, indeed, a powerful tool for managing large-scale multimedia distribution. It gives Reflector administrators tight control over the distribution, allowing for better control of the quality of service. It achieves that through the definition of the distribution topology, the selection of appropriate communication protocols, and the possibilities for limiting the number of clients according to the available resources.

We also learned, however, that it is important to provide better mechanisms for distributed code updates, dynamic reconfiguration, and fault-tolerance, which motivated us to develop the architecture described in this thesis.

7.2.4 Dynamic Configuration of QoS-Sensitive Systems

The synergistic relationships between dynamic configuration and QoS are clear. Dynamic configuration allows the use of the best policies for each situation. For example, a mobile computer displaying a video clip to its user could use a protocol optimized for wireless connections (e.g., WTCP [SVSB99]) when the computer is using a wireless link, but dynamically reconfigure itself to use a TCP connection when the computer is hooked to a wired Ethernet connection.

However, if the reconfiguration process itself affects the quality of service negatively, it may not be worthwhile to do any reconfiguration at all. Going back to the example, if the dynamic reconfiguration to the TCP connection is so expensive that the video is interrupted for several seconds, it is better to keep using the wireless link, even when the wired link becomes available.

Therefore, while developing the new version of the Reflector system, our major goal was to deploy our architecture for dependence management to support dynamic configuration and fault-tolerance *without* affecting quality of service negatively.

7.2.4.1 Automatic Configuration

To solve the problem of maintaining the Reflector instances up-to-date as the code of the Reflector program evolves, and to customize each Reflector according to its role, we used the Automatic Configuration Service described in Chapter 4.

The first major change we had to do in the Reflector implementation to accommodate the new design was the adoption of a component-oriented model. We reorganized the implementation of the Reflector program, breaking it into dynamically loadable components. Surprisingly, this proved to be not so difficult, thanks to the original object-oriented design of the Reflector that was based on loosely coupled objects interacting via well-defined interfaces. This component-based model facilitates the customization of the Reflector program at startup, allowing the use of the Automatic Configuration Service to select the components that are best suited for executing the Reflector in a given node. It also facilitates the dynamic reconfiguration of running Reflectors to adapt to changes in the environment and to install new versions of components on-the-fly.

Figure 7.7 presents a schematic overview of the Reflector bootstrapping process in which the Reflector configures itself with the help of our infrastructure.

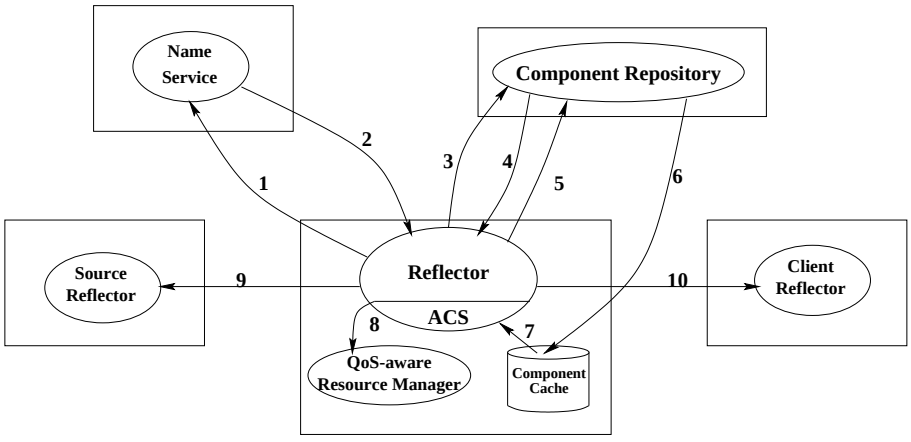


Figure 7.7: Bootstrapping a Reflector

At startup time, each Reflector contacts the CORBA Name Service to locate the Component Repository that is compatible with the operating system and hardware platform on which the Reflector is running (steps 1 and 2 in Figure 7.7). From the Component Repository, it retrieves its specific prerequisite specification file, which may contain information about its resource requirements and the list of components that must be dynamically loaded before the Reflector starts its activities.

The Automatic Configuration Service library (denoted as ACS in Figure 7.7) is linked to the Reflector process. Using the prerequisite file fetched from the Component Repository, the ACS requests the appropriate components from the remote repository (step 5), stores them in the local disk (step 6)[3], and dynamically loads them into memory using the *dynamicTAO* facilities (step 7).

Once all components are loaded, the Reflector uses the resource requirement information in the prerequisite file to request the reservation of CPU and memory from an underlying QoS-aware resource management service. This communication with the resource management service (step 8) is the only part of the bootstrapping scheme that was not implemented in our prototype. QoS-aware resource management service can be provided by systems like SMART [NL97] or the Dynamic Soft Real-Time Scheduler (DSRT) [NhCN98]. In the latter case, DSRT can use the prerequisite specification to perform QoS-aware admission control, negotiation, reservation, and scheduling.

Next, the Reflector registers itself with the Name Service, so that it can be easily located by other system entities, and opens all the input and output connections using the specified protocols (steps 9 and 10). Although our prototype reads the information about the input and output connections from a local configuration file, it could be easily modified to retrieve this information from a file stored in the Component Repository.

Administrators of the Reflector system can look at the available broadcast and videoconference sessions by using a graphical user interface that interacts with the CORBA Name Service. For example, Figure 7.8 is a screen shot that illustrates the use of this interface. It shows three independent Reflector networks: the first called `VirtualMeetingRoom` that could be used for audio and videoconferencing, possibly divided according to interest groups; the second called `News` could contain several news channels; and the third called `OlympicGames` could contain audio and video

[3]To minimize startup time and network load, the components fetched from the Component Repository may be cached locally.

broadcast channels related to Olympic events.

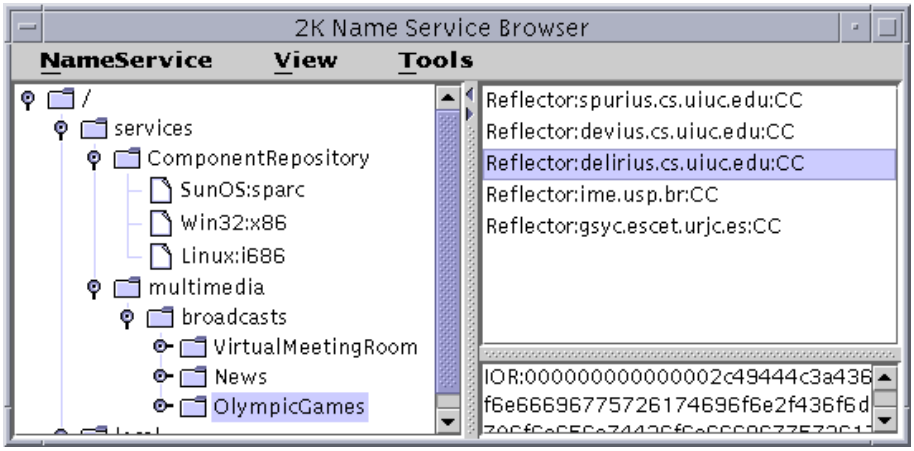


Figure 7.8: A Sample Screen Shot of the Name Service GUI

The administrator can also see that there are three available component repositories for three different kinds of architectures and can use the GUI to upload, download, and remove Reflector components from the repositories. Finally, the administrator can see that the `OlympicGames` network is composed of the five reflectors shown in the upper right-hand window. The CORBA IOR (Interoperable Object Reference) of the Reflector at `delirius.cs.uiuc.edu` is shown in the bottom right-hand window.

7.2.4.2 Fault-Tolerance

An important aspect of quality of service is that of reliability. It is difficult to ensure that the system will work properly in the presence of network failures, node failures, software failures, and temporary system shutdowns for maintenance. In addition, keeping the desired level of quality of service irrespective of all these disruptive events is a major challenge for modern QoS-sensitive systems. The Reflector architecture addresses this problem by using architecturally-aware dynamic reconfiguration based on the component configurator framework described in Chapter 5.

In multimedia distribution, the goal with respect to fault-tolerance is to maximize the system's availability without relying on redundant data transmission, which leads to waste of bandwidth. To achieve that, our prototype supports the dynamic reconfiguration of the Reflector inter-connections when failures occur. The distributed system has knowledge of its own structure and is able to es-

tablish alternative routes to deliver the multimedia streams to its users. Furthermore, whenever possible, the system performs these reconfigurations without affecting the quality of service perceived by its users.

Fault-Recovery Models In order to build an alternate distribution topology, the system must store enough information so that alternate routes can be found when failures occur. The question is: where to maintain this information?

We considered initially a solution in which all the information regarding fault recovery would be placed in a centralized database accessible by every Reflector. When a Reflector R_1 detects that one of its inputs failed or that it is silent for too long, it would contact a *configuration server* with access to the centralized database and request the address of a reflector R_2 from which it could receive the missing streams. The configuration server would return this information and contact R_2, reconfiguring it to send data to R_1. The advantage of this approach is that very little information is stored on the Reflectors, all the fault-recovery information and policies are centralized in a single location, facilitating the manipulation of this information by a single entity: the configuration server.

The second solution is to store fault-recovery information on the Reflectors themselves. That is, each Reflector would store, for each set of multimedia channels, a list of alternate Reflectors that could be used as backups. The advantage of this approach is that it does not lead to a single point of failure and does not impose an extra load on a possibly already overloaded configuration server. This solution may be more difficult to implement, but it tends to be more scalable.

We believe that the optimal solution to this problem is one that encompasses both models. On the one hand, each Reflector should have some knowledge about its local surroundings and should be able to perform reconfigurations by communicating with its immediate neighbors, without being a burden to the centralized configuration server. On the other hand, the configuration server should maintain global knowledge about the system topology. This centralized, global knowledge should be used not only as backup, in case the Reflector's localized information is not enough to keep the system functioning, but also to perform optimizations, such as dynamic changes in the network topology to improve the quality of service and promote load balancing.

Our architecture adopts the hybrid model described above. It distributes the knowledge through-

out the Reflector network and makes each Reflector aware of its dependence relationships with other Reflectors. Thus, the Reflectors are able to make reconfiguration decisions by themselves, without relying on a centralized entity. In addition to this, the global system topology is maintained in the configuration service so that a Reflector administrator or an "intelligent" software module can perform global optimizations in the distribution network.

The prototype supports fault-tolerance by using a subclass of *ComponentConfigurator* to represent the dependencies between Reflectors. When failures occur, the system uses the dependence information to locate alternate routes and keep the system functioning. The *ComponentConfigurator* implementation stores the dependencies as a list of CORBA IORs, which allows for prompt communication no matter where the objects are located.

The subclass of *ComponentConfigurator* we use is called *ReflectorConfigurator* and contains the policies for reshaping the network topology in case of failures and encapsulates all the code to deal with these reconfigurations. This approach proved to be very effective in keeping a clear separation of concerns in the Reflector code. The classes that deal with the Reflector's normal operation are totally unaware of the *ReflectorConfigurator* and of any code that deals with reconfiguration. This clean separation also makes it easy to plug different kinds of *ReflectorConfigurator*s to support different reconfiguration policies.

Triggering Reconfigurations In this implementation, four kinds of events can trigger dynamic reconfiguration:

1. A Reflector shutdown message sent by the Reflector administrator or a `kill` command executed by the local system administrator.

2. Errors (or "bugs") in the Reflector or library code that lead to a segmentation fault or bus error.

3. A reconfiguration order sent by the Reflector administrator.

4. Sudden machine crashes or network disconnections.

In the first two cases, the Reflector captures those events using signal handlers installed with the UNIX `signal` function or the Windows `SetConsoleCtrlHandler` function. In the UNIX imple-

mentation, for example, the administrator can kill a Reflector by pressing `Ctrl-C` on the terminal executing the Reflector, by sending a shutdown message to the reflector using a telnet connection, or by using the `kill` command. The Reflector captures the events generated by `Ctrl-c`, `kill`, segmentation faults, and bus errors by implementing signal handlers for the SIGINT, SIGTERM, SIGSEGV, and SIGBUS signals, respectively.

In the third case, the Reflector contacts the configuration service to retrieve its new configuration information and reprocesses it, reconfiguring its input and output connections.

Finally, the fourth case is the only one in which it is not possible to keep the client multimedia stream uninterrupted without relying on redundant streams to the same Reflector. The solution in this case is to detect when the input for a given channel has failed or is silent for too long and then locate an alternative input either by using the local list of alternatives or by contacting the configuration server.

Our current implementation focuses on supporting dynamic reconfiguration in the presence of the first two kinds of events. We now describe this process in more detail.

The Reconfiguration Process When an administrator (or other system entity) requests to kill a Reflector, the system executes a special event handler called `abandonReflectorNetwork ()`. This handler takes the following three actions.

1. Unregisters the Reflector from the Name Service.

2. Using CORBA, sends a FINISHED event to the `ReflectorConfigurator`s of all the sources (inputs) of this Reflector; the event carries a list of the clients of the finishing reflector.

3. Using CORBA, sends a FINISHED event to the `ReflectorConfigurator`s of all the clients (outputs) of this Reflector; the event carries a list of the sources of the finishing reflector.

When a `ReflectorConfigurator` receives a FINISHED event from a source Reflector, it adds all the Reflectors in the list of sources of the finishing Reflector to its list of inputs[4].

[4]This action may lead to redundant inputs for the same channel. This can be avoided by extending the code of the `ReflectorConfigurator` so that it adds all the new sources to its list of input alternatives and chooses only one of them to be its new input.

Conversely, when a `ReflectorConfigurator` receives a FINISHED event from a client Reflector, it adds all the Reflectors in the list of clients of the finishing Reflector to its output list.

Figure 7.9 shows a sample Reflector network where Reflector C has two inputs and two outputs. When C is killed and the reconfiguration process described above completes, the new configuration becomes the one in Figure 7.10.

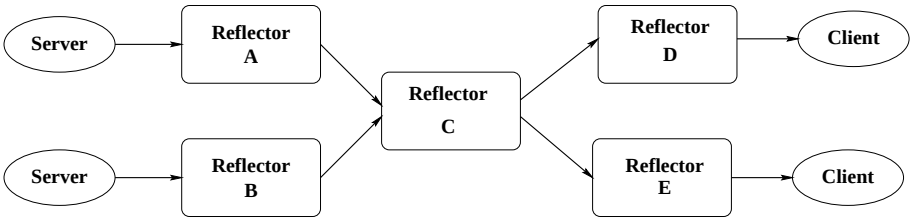


Figure 7.9: A Distribution Network with Five Reflectors

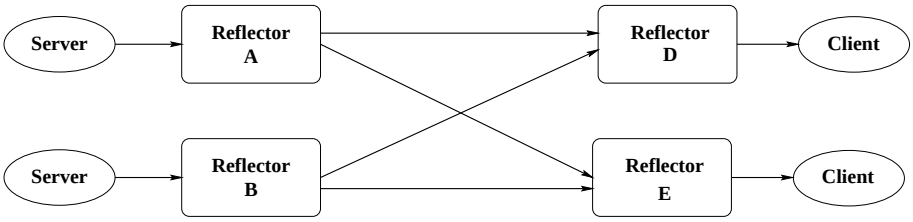


Figure 7.10: The Distribution Network After Reflector C Abandons the Network

In order to be able to carry out the reconfiguration without any glitches in the multimedia streams and without affecting the system quality of service, we had to adopt the multithreaded solution described in section 8.2.

7.2.4.3 The Recovery Process

When a Reflector starts its execution for the first time or when it is restarted after being shutdown for some reason, it executes an initialization process. In this process, in addition to performing the actions described in 7.2.4.1, it performs the following three actions.

1. Registers the Reflector with the Name Service.

2. Using CORBA, sends a STARTED event to the `ReflectorConfigurators` of all the clients (outputs) of this Reflector; the event carries a list of the sources of the new reflector.

3. Using CORBA, sends a STARTED event to the `ReflectorConfigurator`s of all the sources (inputs) of this Reflector; the event carries a list of the clients of the new reflector.

Upon receiving a STARTED event from a new source Reflector, the client Reflector opens a new input connection to the new Reflector. If it is also receiving input from one of the sources of the new Reflector, it closes that input connection as soon as the data from the new source is available. An analogous process happens upon receiving a STARTED event from a new client Reflector.

These mechanisms allow the distribution system to recover its original topology after a faulty reflector restarts. Therefore, if the system configuration is the one in Figure 7.10 and the Reflector C recovers, then the configuration switches back to the one in Figure 7.9.

Note that we do not have an automatic mechanism for restarting faulty Reflectors. This requires the administrator's intervention, which seems to be natural since a Reflector goes out of service either because of an administrator's command or because of a failure in the system. In both cases, the administrator's attention is advisable. Alternatively, if desired, the Reflector can be added to the list of daemons that are executed by the operating system when a machine boots, eliminating the need for manual intervention when a machine crashes and restarts.

In section 8.2, we present the results of several experiments with the Reflector prototype. The performance evaluation shows that, with the help of our infrastructure for dependence management, the Reflector system is able to carry out dynamic reconfiguration of the distribution topology without any negative impact on the quality of service perceived by end-users. This dynamic reconfiguration provides a solid base for fault-tolerance.

7.3 Other Applications

In addition to the use cases described in section 5.4, three ongoing projects carried out by other researchers in our group are using the products of this thesis.

1. *Gaia* [RC00], a middleware infrastructure for management of active spaces will use both the Automatic Configuration Service and component configurators to manage dynamic, heterogeneous devices in ubiquitous computing scenarios. As a traditional operating system manages the resources of a machine, Gaia manages the resources in an active space. Examples of active

spaces include active offices and active lecture rooms enhanced with various computing and multimedia devices. Interactions between the user and the devices in the space are managed by a framework inspired in the Model-View-Controller paradigm [KP88]. Gaia uses component configurators to manage the dependencies among the Model, the Controller, and the multiple Views.

2. *2KFS*, the *2K* distributed file service [HBC00] will use the Automatic Configuration Service to dynamically load components, implementing different algorithms for caching and data transcoding. The component configurator framework will be used to manage dependencies among the distributed *2KFS* components.

3. $2K^Q$ is a reconfigurable, component-based QoS framework which partitions the end-to-end QoS setup process into distributed QoS compilation and runtime QoS instantiation phases [NWX00]. Dynamic instantiation is based on *dynamicTAO* and the Automatic Configuration Service.

Chapter 8

Experimental Results

In this chapter we present the results of experiments carried out to evaluate the performance of the three major elements of our architecture: the Automatic Configuration Service, the component configurator (in this case, used to support dynamic reconfiguration of the Reflector Network), and the mobile agents infrastructure (used for reconfiguration, inspection, and code distribution).

8.1 Automatic Configuration Service

The Automatic Configuration Service is implemented as a library that can be linked to any application. A program enhanced with this service becomes capable of fetching components from a remote Component Repository and dynamically loading and assembling them into its local address-space. The library requires only 157Kbytes of memory on Solaris 7, which makes it possible to use it even on machines with limited resources such as a PalmPilot. In fact, we expect that services similar to this will be extensively used in future mobile systems to configure software automatically according to location and user requirements.

To evaluate the performance of the Automatic Configuration Service, we instrumented the Reflector application described in Section 7.2 to measure the time for fetching, dynamic linking, and configuring its constituent components.

8.1.1 Loading Multiple Components

Figure 8.1 shows the total time for the service to load from one to eight components of 19.2Kbytes each. These experiments were carried out on two Sparc Ultra-60 machines running Solaris 7 and

connected by a 100Mbps Fast Ethernet network. The Component Repository was executed on one of the machines and the Reflector with the Automatic Configuration Service on the other. Each value is the average of five runs of the experiment. The vertical bars in the subsequent graphs and the numbers in parentheses in Table 8.1 represent the standard deviation.

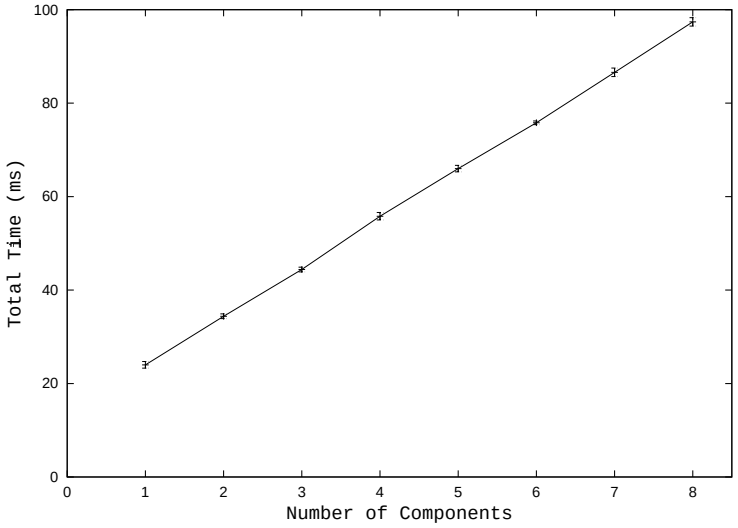


Figure 8.1: Automatic Configuration Service Performance

Table 8.1 shows, in more detail, how the service spends its time when loading a single 19.2Kbyte component. The current version of the Automatic Configuration Service fetches the prerequisites file from the remote Component Repository and saves it to the local disk. The same is done with the file containing the component code. Then, it uses *dynamicTAO* to perform the local dynamic linking of the component into the process runtime.

Action	Time (ms)	% of the total
fetching prerequisites from Component Repository	2 (0)	8
saving prerequisites to local disk	1 (0)	4
fetching component from Component Repository	4 (0)	17
saving component to local disk	1 (0)	4
local dynamic linking	5 (0)	21
autoconf protocol additional operations	11 (0.7)	46
Total	24 (0.7)	100

Table 8.1: Discriminated Times for Loading a 19.2Kbyte Component

The table also shows the additional time spent by the service (row labeled as "autoconf protocol additional operations") to detect if there are more components or prerequisite files to load, to parse

the prerequisite file, and to reify dependencies. This overhead accounts for 46% of the total time required to load the component, which suggests that it would be desirable to improve this part of the service by optimizing the implementation of the *SimpleResolver* (see Section 4.2). We believe that an optimized version of the *SimpleResolver* could lead to improvements in the order of 20% for components of this size.

In the experiments described in this section, the component code and prerequisite files were cached in the memory of the machine executing the Component Repository. When the Component Repository program needs to read both files from its local disk, there is an additional overhead of approximately 20 milliseconds.

8.1.2 Loading Components of Different Sizes

To evaluate how the time for loading a single component varies with the component size, we created a program that generates components of different sizes. According to its command-line arguments, this program generates C++ source code containing a given number of functions (which include code to perform simple arithmetic operations) and local and global variables. Using this program, we created components whose DLL sizes vary from 12 to 115 Kbytes. Figure 8.2 shows the time for the Automatic Configuration Service to load a single component as the component size increases.

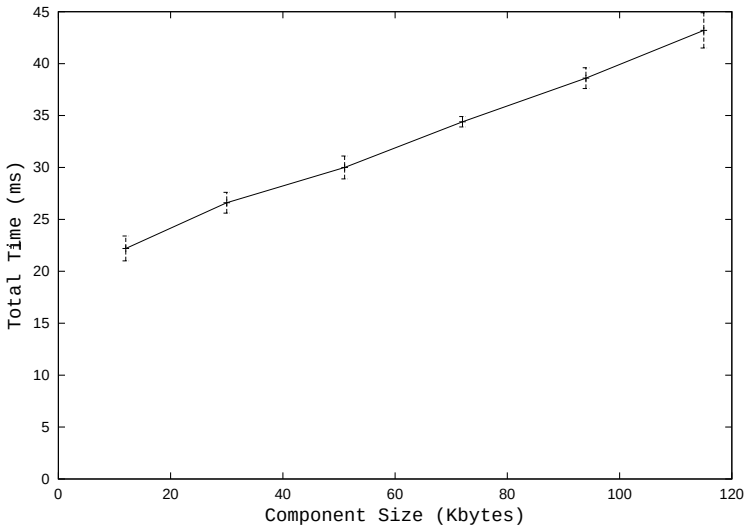


Figure 8.2: Times for Loading Components of Different Sizes

Figure 8.3 shows the absolute times spent in each step of the process[1]. We can notice that the time spent in the item labeled "autoconf protocol" is approximately constant[2]. Hence, as the component size increases, its relative importance decreases. This can be noticed in Figure 8.4, which shows the same data in a different form. In this case, the figure shows the percentage of the total time spent in each of the steps of the process.

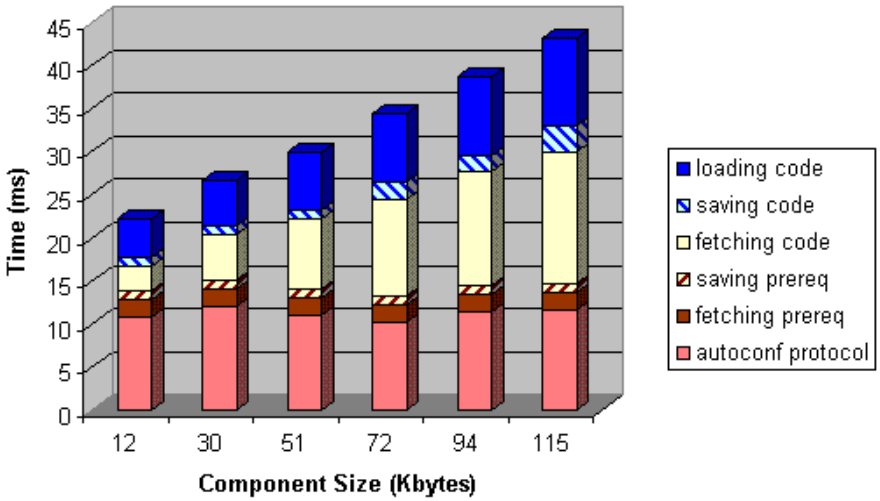


Figure 8.3: Discriminated Times for Loading Components of Different Sizes

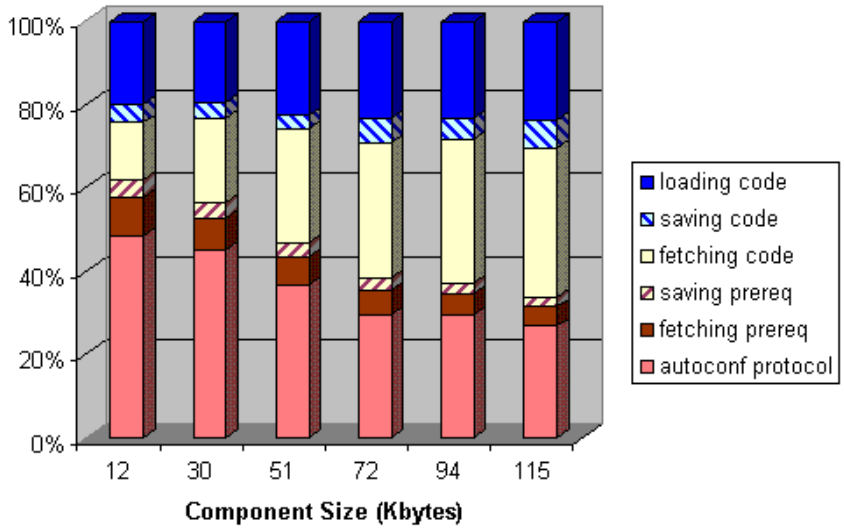


Figure 8.4: Discriminated Percentual Times for Loading Components of Different Sizes

[1]These steps are the same as those presented in Table 8.1.

[2]This is expected since the messages processed in this step do not carry component code and therefore are not affected by the size of the component.

As the size of the component increases, the time for fetching the code from the remote repository to the local machine becomes the dominant factor. It is important to remember that these data were captured in a fast local network. If the access to the repository requires the use of a lower bandwidth connection, then this step would clearly be the most important with respect to performance. This suggests that "component caching" is an important topic for future research.

Discussion

Although there is still much room for improvements and performance optimizations in the protocols used by the Automatic Configuration Service, the results presented here are very encouraging. They demonstrate that it is possible to carry out automatic configuration of a distributed component-based application within a tenth of a second, which is what we intended to prove.

8.2 Dynamic Reconfiguration Using Component Configurators

In Section 7.2.4 we explained how to use component configurators to manage dependencies and support dynamic reconfiguration of a distributed application such as the Multimedia Distribution System. In this section, we present experimental results showing that dependence management using component configurators can be performed efficiently.

We conducted experiments to evaluate the performance of dynamic reconfiguration using component configurators in the Reflector system as described in Section 7.2.4.2. In particular, we were interested in evaluating the impact of the automatic reconfiguration of the reflector network on the quality of the multimedia streams received by end-users. We carried out this set of experiments on seven Sun Ultra machines[3] connected by a 100Mbps Fast Ethernet. As depicted in Figure 8.5, three of these machines executed our Reflector program. Another machine executed TestServer, a program we created to synthesize an RTP stream [SCFJ00] with bandwidth and packet rate specified in its command line. A fifth machine executed TestClient, a program that receives the packets from a Reflector and logs the packet arrival times into a file. The sixth machine executed the CORBA Name Service and the last one, the Component Repository.

[3] Our testbed consisted of two dual-processor Sun Ultra-60 machines with 360 MHz processors and five Sun Ultra-5 machines with 333 MHz processors.

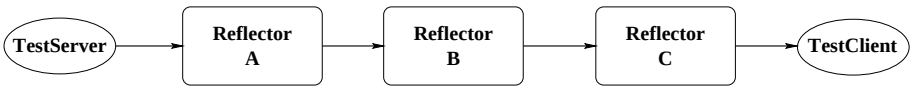


Figure 8.5: Reconfiguration Experiment Testbed

As explained in Section 7.2.4.2, when the Reflector B goes down, the system reconfigures itself automatically and adopts the new topology shown in Figure 8.6. When the Reflector B recovers, the system reconfigures itself back to the topology shown in Figure 8.5.

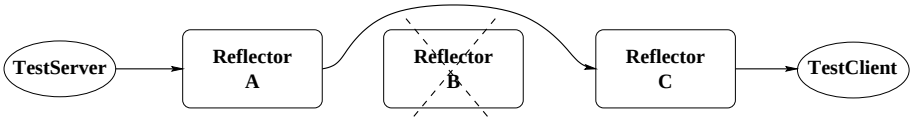


Figure 8.6: Reconfiguration when Reflector B Goes Down

We evaluate the impact of these reconfigurations on the quality of service as perceived by end-users by using the information in the log file to compute the packet inter-arrival time at the TestClient. The following figures plot packet inter-arrival time (in seconds) over time (in seconds). Each experiment lasted between 15 and 18 seconds.

Figure 8.7 shows the packet inter-arrival times at our testbed client when no reconfigurations take place. The TestServer sends an RTP/UDP stream of 1.2Mbps at 30 packets per second.

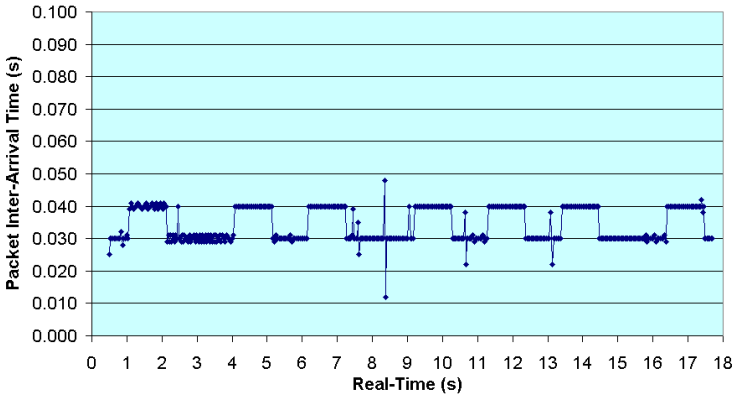


Figure 8.7: No Reconfigurations. Bandwidth = 1.2Mbps, Packet Rate = 30pps

The TestServer program uses an adaptive algorithm to keep its average output bandwidth as close as possible to the output bandwidth specified at its command line. Once every K seconds, the program checks its output bandwidth and modifies its packet inter-transmission time to adapt to the

changes in its measured output bandwidth[4]. The packet size is fixed throughout each experiment and it is computed by dividing the bandwidth by the packet rate.

In our first experiments, we set $K = 1$ and the desired packet rate to 30 packets per second. The TestServer, then, tries to send one packet every 33.3 milliseconds. However, since the Solaris clock tick is set to 10 milliseconds, it does not let a program sleep for exactly 33.3 milliseconds. Hence, our adaptive program ends up sending one packet every 30 or 40 milliseconds which explains most of the variations throughout the experiment in Figure 8.7. The small and the localized variations (such as the one at $8.4s$) are due to changes in the network and machine loads caused by other applications.

Figure 8.8 shows the impact of reconfigurations on our first implementation of the fault-tolerance mechanism described in Section 7.2.4.2. The continuous arrows point to the instants in which the Reflector B is killed and the Reflector network topology switches from the configuration in Figure 8.5 to the one in Figure 8.6. The dashed arrows point to the instants in which the Reflector B recovers and the topology switches back to the one in Figure 8.5.

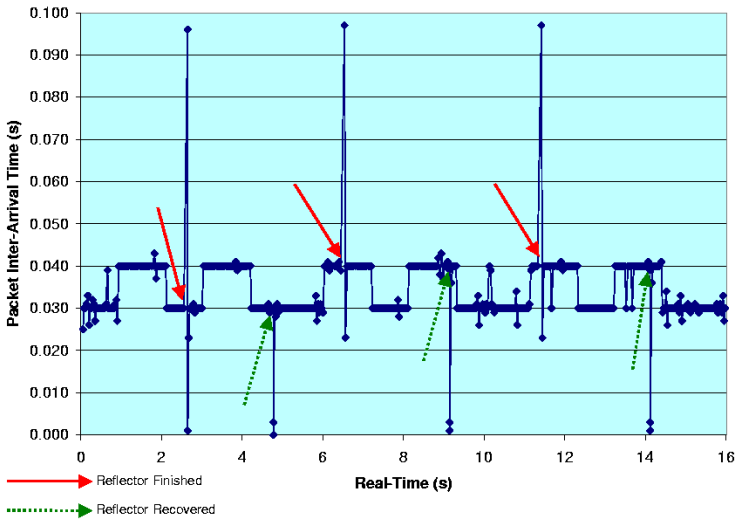


Figure 8.8: Reflector Reconfigurations. Bandwidth = 1.2Mbps, Packet Rate = 30pps

One can see that when we killed the Reflector, there was a peak in the inter-arrival time at the client. This happened because, in that implementation, as soon as the reflector received the termination signal, it stopped forwarding packets and sent events to its neighbors announcing that

[4]The measurement of the output bandwidth is based on the values returned by the *send* system call.

it was going to shutdown. This caused a delay until the other reflectors were able to reconfigure their inputs and outputs. In a wide-area network, the delays would be even larger and the quality of service as perceived by the end-user would be degraded significantly.

We solved this problem by creating a new thread to manage the reconfiguration. While the new thread contacts the Reflector's sources and destinations to announce the end of its service, the old thread continues to perform the Reflector's normal packet-forwarding operations. The Reflector only shuts itself down after it receives a confirmation from its neighbors that its service is no longer needed[5].

One can observe another problem in the graph of Figure 8.8; as the reflector recovers, there is a sudden change in the packet inter-arrival time, which approaches zero for a couple of packets in a row. This happens because the end-user receives some repeated packets that are sent both by the "backup" connection being deactivated and the new, "recovered" connection. This problem of duplicated packets is solved very easily by implementing a new subclass of `Connection` that uses the RTP sequence number to drop repeated packets.

Figure 8.9 shows the result of running the same experiment after implementing these two improvements. One can see that the reconfigurations take place without affecting the quality of service as perceived by the end-user (one can say that because there is no correlation between the arrows in Figure 8.9 and the variations in packet inter-arrival times). Furthermore, since the old Reflector keeps a thread performing the Reflector's normal operations until the reconfiguration is completed, the quality of service is not degraded even in wide-area networks where the latency to contact the neighboring Reflectors may be large.

The lack of any correlation between the Reflector reconfigurations and the quality of service is even more clear in Figure 8.10 that shows a similar experiment carried out in a period of high network and machine loads. In this case, we set K to 10 so that the TestServer would keep its transmission rate constant for longer periods.

Finally, Figure 8.11 shows one more experiment in which we used the Reflector network to distribute an RTP stream generated by the vat audioconference tool for the MBone [SRL96]. The acoustic perception of five users listening to the audio stream in our laboratory confirms what one

[5]If this confirmation does not arrive, the Reflector administrator or a controller program still has the chance to force the termination of the Reflector program.

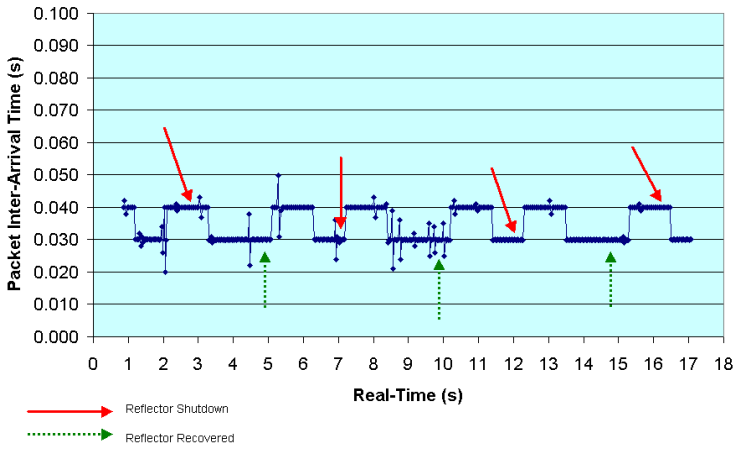


Figure 8.9: Reflector Reconfigurations. Bandwidth = 1.2Mbps, Packet Rate = 30pps

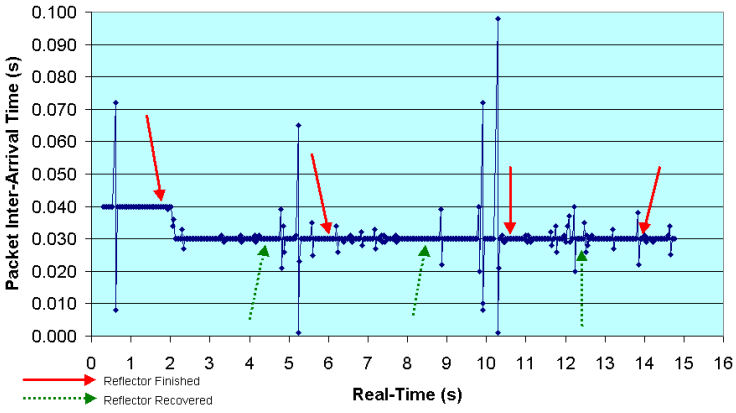


Figure 8.10: High Network and CPU Load. Bandwidth = 1.2Mbps, Packet Rate = 30pps

sees in the graph: the reconfigurations do not affect the quality of service of the audio application.

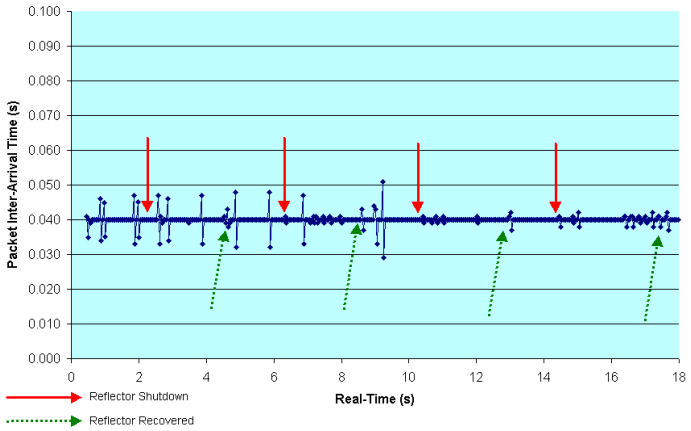


Figure 8.11: Reflector Reconfigurations. Vat Bandwidth = 45kbps, Packet Rate = 25pps

Discussion

The experiments described in this section addressed only one of the many ways in which component configurators can be helpful. Nevertheless, they show how the reification of dependencies can facilitate the implementation of reliable systems with little or no performance penalty. In fact, we believe that, in some cases, a proper dependence management can actually enable performance optimizations [SSC98a].

8.3 Mobile Agents for Reconfiguration, Inspection, and Code Distribution

In order to evaluate the response time and relative performance gains made possible by our mobile agent based reconfiguration infrastructure, we established an intercontinental testbed with the collaboration of the Departments of Computer Science at the Rey Juan Carlos University in Madrid, Spain and at the Campinas State University in Campinas, Brazil. The testbed consisted of the following three groups of machines.

1. 2 Sun Ultra-60 and 1 Sun Ultra-5 machines running Solaris 7 in the `cs.uiuc.edu` domain.

2. 3 Pentium-based 333MHz PCs running Linux RedHat 6.1 in the `escet.urjc.es` domain.

3. 3 Pentium-based 300MHz PCs running Linux RedHat 6.1 in the `ic.unicamp.br` domain.

The machines inside each group were connected by 100Mbps Fast Ethernet networks, while the groups were connected among themselves through the public Internet. We executed several instances of a test application running on top of *dynamicTAO* and injected different kinds of agents in this network.

To avoid drastic oscillations in the available Internet bandwidth and latency, and to minimize undesired interference, we carried out the experiments during the night[6]. We ran all the experiments between 11:30PM on Tuesday November 23, 1999, and 1:00 am in the following day (USA CST). Thus, the starting time in Brazil and Spain were 3:30 am and 6:30 am, respectively, and the Internet traffic was relatively low. Table 8.2 shows the average bandwidth (measured by transferring a 100Kbyte file via FTP five times) and latency (measured with the *ping* command) between our lab in Illinois and the remote ones.

	`escet.urjc.es`	`ic.unicamp.br`
available bandwidth	76 Kbytes/sec	32 Kbytes/sec
round-trip latency	170 ms	270 ms

Table 8.2: Average Bandwidth and Latency from `cs.uiuc.edu`

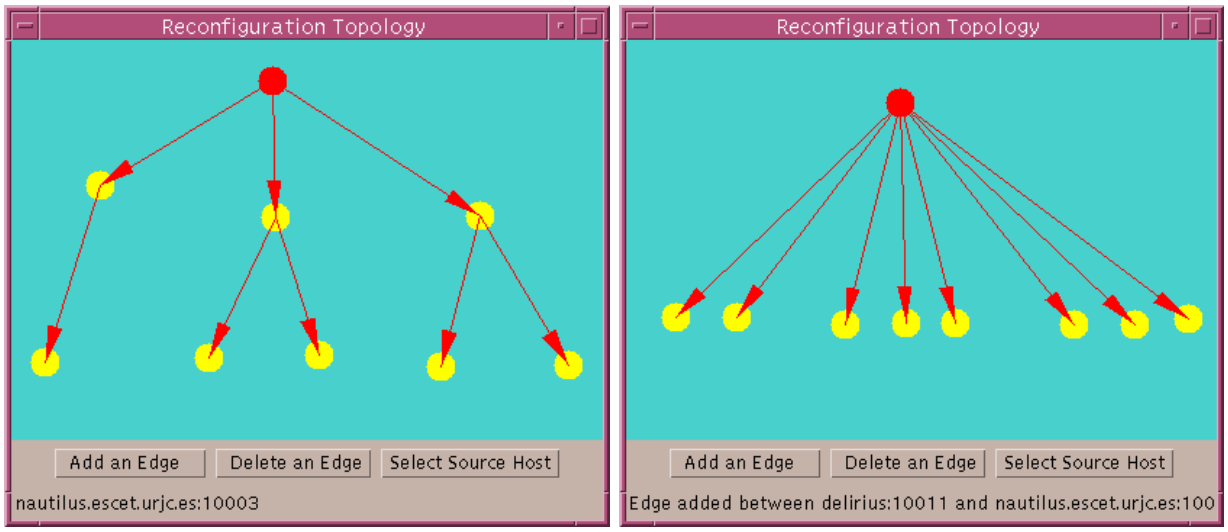
Table 8.3 shows the total round-trip time to execute five DCP inspection commands (*list_categories*, *list_implementations*, *list_loaded_implementations*, *list_domain_components*, and *get_comp_info*) using different techniques. In all the subsequent tables and graphs, each number is the arithmetic mean of ten runs of each experiment and the numbers in parentheses (vertical bars in the graphs) are the standard deviations. The first column shows the elapsed times for the execution of the agents in

	9 ORBs	1 ORB
Agents using distribution tree	847(33)	
Agents point-to-point	2127(1044)	362(10)
Client/Server		1128(141)

Table 8.3: Elapsed Time (in ms) for the Execution of Five Different Inspection Commands

[6] When we ran the experiments during times of high network traffic and congestion, the performance numbers were even more in favor of our mobile agents approach.

9 instances of the *dynamicTAO* ORB, 3 in each domain. The second column shows the times for execution in a single ORB in a remote node. The first line shows the time for agents following the distribution tree shown in Figure 8.12(a). The second line shows the times for agents sent directly from a single node in `cs.uiuc.edu` for parallel execution on the other eight nodes as shown in Figure 8.12(b). Finally, the third line shows the time for executing the same five inspection commands by using five consecutive requests in a traditional client/server model rather than using the agent model.



(a) Using a tree

(b) Using point-to-point connections

Figure 8.12: Agent Distribution Topology

Looking at Table 8.3, we can see that by using a well-thought distribution tree, we obtained a 60% performance improvement in relation to a point-to-point distribution scheme (847 *ms* against 2127 *ms*). Using an agent containing the five commands, instead of a separate request for each command, led to a 68% performance improvement (362 *ms* against 1128 *ms*).

To measure how the benefits of using a distribution tree as the one in Figure 8.12(a) varies with the size of the agent, we sent a series of agents carrying the code for components to be installed in the remote nodes. As shown in Figure 8.13, as the size of the component being uploaded increases, the relative gain of using a distribution tree instead of point-to-point connections increases significantly. This shows clearly the benefits of using the mobile agents infrastructure for code distribution.

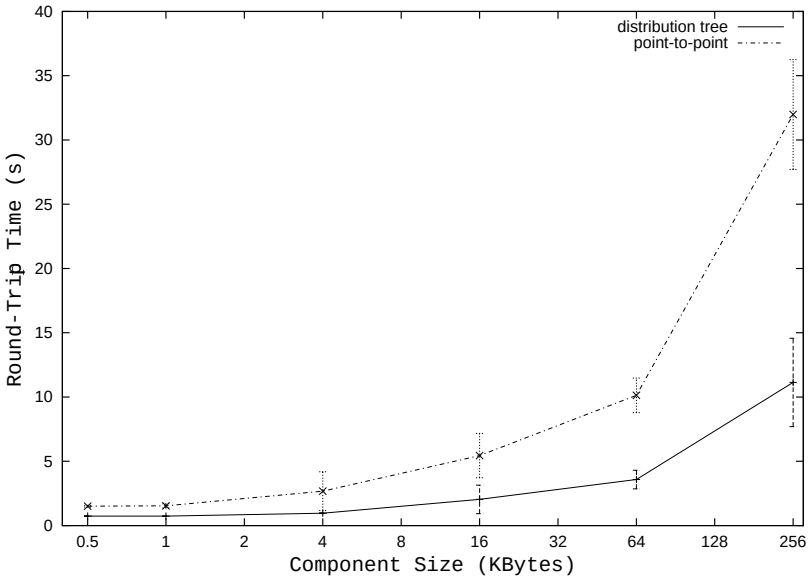


Figure 8.13: Agent Uploading a New Component to Nine Nodes

Figure 8.14 shows the elapsed time for executing agents carrying from one to eight reconfiguration commands.

Finally, Figure 8.15 shows the comparative performance between the agent approach and conventional client/server requests as the number of requests increases from one to eight. The client/server times were measured by building an application that sends the commands contained in the agent as separate requests transmitted via TCP/IP connections. Since the overhead of processing this kind of agent (around $1ms$) is negligible compared to the latency of long distance Internet lines, the performance of both approaches when sending a single command is roughly the same. As we increase the number of commands in each experiment, the total completion time for the agent barely varies while the completion time for the client/server application increases rapidly as expected.

Discussion

Although our implementation could still be improved significantly with more tuning and optimizations, these results are very encouraging. They demonstrate clearly that mobile agents can provide extreme performance improvements for the management and reconfiguration of wide-area distributed systems.

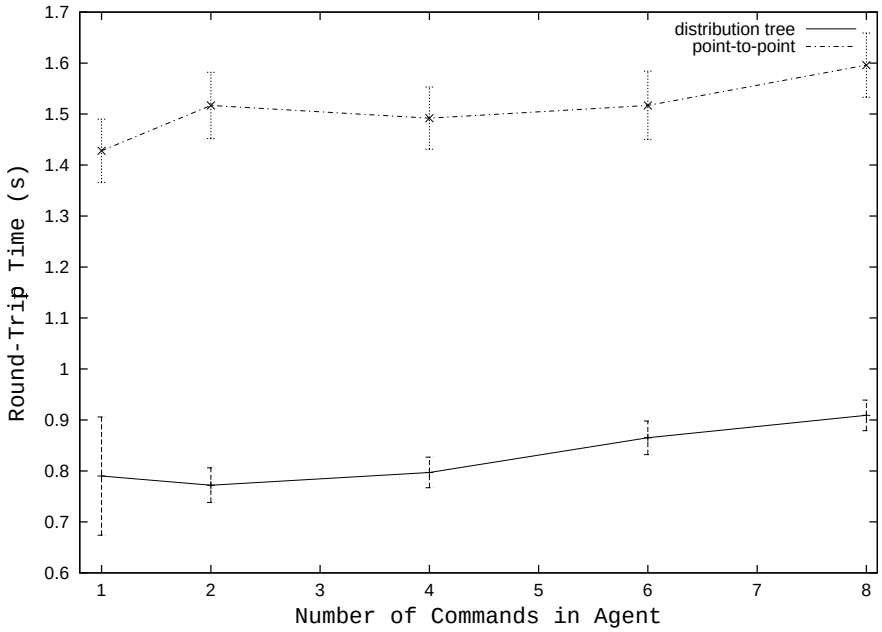


Figure 8.14: Reconfiguration Agent Visiting Nine Nodes

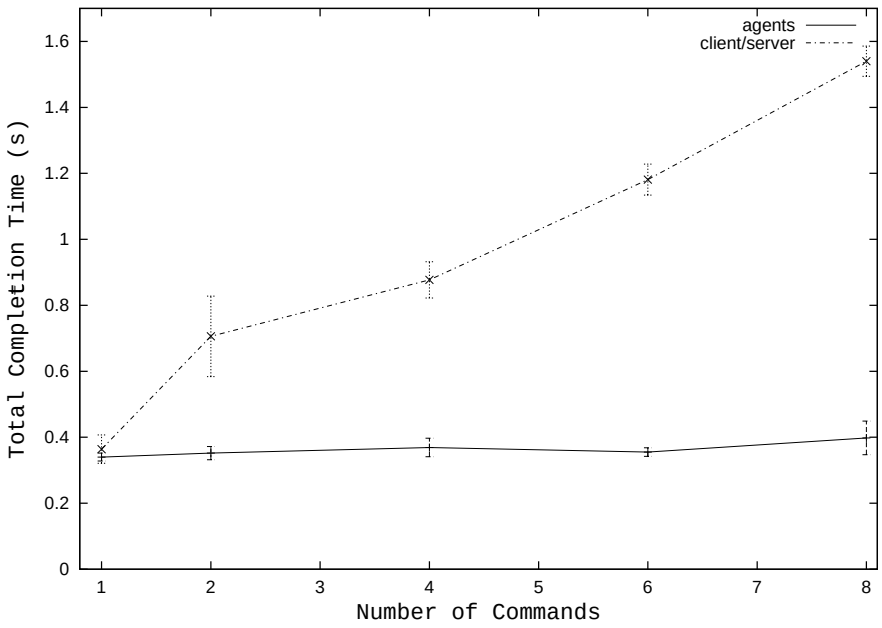


Figure 8.15: Agents Versus Conventional Client/Server

Chapter 9

Related Work

No other system architecture offers a more complete and integrated solution for automatic configuration in heterogeneous, distributed systems than the one proposed in this thesis. To the best of our knowledge, our work is the first to identify the lack of an explicit representation of inter-component dependence as one of the most fundamental problems in existing computer systems.

Our architecture leverages modern technologies such as CORBA, component-based programming, and mobile agents with the results of the research in operating systems and dynamic configuration of the previous decade.

In this section, we describe previous work in related areas, explain how they influenced our research, and discuss the differences with our approach.

9.1 Programming Models

Our work was strongly influenced by the advances in programming languages and methodologies for software development of the last two decades. These advances include not only the concepts of reflection and aspect-oriented programming, described next, but also, object-orientation and design patterns [GHJV95].

9.1.1 Reflective Programming

The idea of having a *ComponentConfigurator* object associated with each component is similar to the concept of *meta-objects* present in reflective languages such as CLOS [KdRB91], OpenC++ [Chi95], Iguana [GC96], and Guaraná [OB99] and reflective systems such as Apertos [ILY95]. Meta-objects

are used to modify the object model and customize all aspects of object behavior, at compile time, link time, or run time. In contrast, the intent of the component configurator is not to modify the object model, but to maintain extra information about inter-component dependence. This extra information helps the system to provide automatic configuration and fault-tolerance and offers components the possibility of reasoning about their own dependencies.

Reflective languages and environments could be used to implement the component configurator model. The management of dependencies could be carried out by meta-objects associated with the relevant components. The reflective mechanisms could then be used to instrument component creation and destruction to take care of the dependencies properly. The architecture described here does not rely on a particular reflective language or environment. We use only standard languages and architectures such as C++, Java, and CORBA.

In spite of the differences between our model and previous implementations of reflective systems, we believe that component configurators are a valuable tool for implementing high-performance reflective systems. Component configurators help realizing the following basic concepts of reflective systems.

- *Introspection of system structure* is achieved by examining inter-component dependence relationships through the component configurator interface.
- *Reconfiguration of system structure* is achieved by manipulating the dependencies reified by the component configurator and by adding and removing components.
- *The construction of systems that can reason about themselves* is achieved by customizing the implementation of the component configurators, adding specialized code to implement specific policies.

Unlike most existing reflective systems, in our model, the component configurator is not present in the normal invocation path to the component or to the objects inside it. The configurator is not activated each time the component is invoked. Normally, the component configurator code is only executed when some meta-level operation, such as reconfiguration, is required.

9.1.2 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [KLM+97] is a novel programming paradigm in which different aspects of a program are coded in separate subprograms. A tool called *aspect weaver* is used to combine multiple aspects into a single executable program image. As noted in [Lun98], AOP can be seen as a reflection model in which each object has multiple meta-objects (its aspects).

AOP could also be used to implement the model described in this thesis. Dependence management could be coded in a separate aspect that would be inserted in the executable program by the aspect weaver. Another option would be to use AOP to implement complex component configurators. Programmers would write the code to deal with the different aspects of dependence management (such as fault-tolerance, dynamic reconfiguration, and adaptability) using AOP aspects. The weaver would then be used to generate the component configurator.

AOP is still a very recent, emerging technology. The modeling of dependencies using aspects is an open problem. It is not clear yet how to support dynamic (re)configuration of aspects. In existing AOP platforms, executables are created off-line by the aspect weaver that mixes the code of all the aspects, making dynamic replacement of a single aspect extremely difficult. A solution to this problem may reside in some of the recent research in this area [BA00].

9.2 Component Architectures

Three major component architectures are likely to be present in the software development arena in the next few years: Enterprise JavaBeans [Ham97], ActiveX Controls [Can97, Den97], and the CORBA Component Model (CCM) [OMG99]. They all intend to provide an infrastructure to support the construction of complex applications from off-the-shelf components.

9.2.1 Enterprise Java Beans and Jini

Sun Microsystem's Enterprise Java Beans specify a standard for packaging components, called JAR, and for inspecting the interfaces exported by each component (Introspection). Java beans must be completely composed of Java byte code, which limits the applicability of the model.

Jini is a set of mechanisms for managing dynamic environments based on Java. It provides

protocols to allow services to *join* a network and *discover* what services are available in this network. It also defines standard service interfaces for leasing, transactions, and events [Wal98].

When a Jini server registers itself with the Jini lookup service, it stores a piece of Java byte code, called proxy, in its entry in the lookup service. When a Jini-enabled client uses the lookup service to locate the server, it receives, as a reply, a *ServiceItem*, which is composed by a service ID, the code for the proxy, and a set of service attributes. The proxy is then linked into the client address-space and is responsible for communication with the server. In this way, the communication between the client and the server can be customized, and optimized protocols can be adopted.

This Jini mechanism for proxy distribution can be achieved in a CORBA environment by using the Automatic Configuration Service (see Chapter 4) in conjunction with a reflective ORB such as *dynamicTAO* (see section 7.1). The Automatic Configuration Service would fetch the proxy code and dynamically link it, while *dynamicTAO* would use the TAO pluggable protocols framework [SC00] to plug the proxy code into the TAO framework.

Jini is typically limited to small-scale networks and it does not address the management of component-based applications and inter-component dependence. Due to the large memory requirements imposed by Java/Jini, this is not yet a viable alternative for most PDAs and embedded devices.

9.2.2 CORBA Component Model (CCM)

The CORBA Component Model (CCM) adopted by OMG on November, 1999, specifies a standard framework for building, packaging, and deploying CORBA components [OMG99]. Unlike our model, which focuses on prerequisites and dynamic dependencies, the CORBA Component Model concentrates on defining an XML vocabulary and an extension to the OMG IDL to support the specification of component packaging, customization, and configuration. We believe that our model and CCM complement each other and could be integrated. CCM provides a static description of component needs and interactions while our model manages the runtime dynamics.

Although CCM was already approved by OMG, publicly available ORBs do not support it yet. Once this happens, we intend to work towards the integration of the two models.

9.2.3 ActiveX Controls, COM, and DCOM

Microsoft's ActiveX Controls architecture [Can97, Den97] also provides mechanisms for packaging, dynamic instantiation, and inspection of the interfaces exported by COM components. Distributed components can communicate via DCOM, which defines a protocol based on the Distributed Computing Environment (DCE) RPC specification [Loc94]. The architecture is tailored to Microsoft operating systems and very little was done to support interoperability with other platforms. As shown in [CHY+98], implementing DCOM clients and servers is clearly more complicated than writing CORBA clients and servers. The architecture provides no support for dependence management.

9.2.4 OpenDoc and OpenStep

OpenDoc [OHE96], promoted in the past by Apple and CI Labs, defines a compound document infrastructure based on IBM's SOM and CORBA. Its design influenced modern component architectures but it was abandoned in favor of Java Beans. Apple is now supporting CORBA, Java Beans, and ActiveX in their OpenStep environment [Don97]. OpenStep is an object-oriented software development environment that runs on top of Mach, Windows NT, Solaris, and HP-UX and provides an interface independent of the underlying operating system.

9.3 Prerequisites

A few systems dealt with concepts similar to the prerequisites presented in section 3.2.1, thus addressing parts of the problem of dependence management.

9.3.1 Job Control Languages

The concept of prerequisites has its origins in the Job Control Languages of the mid-1960s. The OS/360 operating system, deployed in a wide range of IBM computers, used JCL [Flo71] to control the multiple jobs in its innovative "multiprogramming" environment. The `JOB` statement, in JCL, may have a series of keyword parameters of the form `<NAME=value>` to specify certain characteristics of the job. The statement

```
//Little2KJob     JOB     TIME=(3,45) REGION=20K PRTY=5
```

specifies that the job `Little2KJob` takes approximately 3 minutes and 45 seconds to run, requires approximately 20 Kbytes of memory and must be executed with priority 5. Thus, the parameters of the `JOB` statement can be seen as a primitive version of the "hardware prerequisites" portion of a SPDF specification (see section 4.2.1).

9.3.2 SOS

The use of the term "prerequisites" to refer to the dependencies of operating system objects originated in the SOS operating system [SGH+89, MGLNS94] developed at INRIA, France. In the SOS model, objects contain a list of prerequisites that must be satisfied before they are activated. Even though the idea was promising, it was not fully explored in that project. Prerequisites were only used to express that an object depends on the code implementing it and not much experimentation was carried out [SGM89, Sha98]. Besides, SOS does not include a model for dynamic management of inter-component dependence.

9.3.3 Dynamically Loadable Libraries

Modern operating systems, like Solaris, use an explicit representation of the dependencies among shared libraries to implement dynamic linking [Sun97]. A shared library contains a list of the pathnames of other libraries upon which it depends. The system tries to locate all the required libraries before executing any application code.

Our approach can be seen as an extension to that, since it uses a representation of dependencies to support dynamic loading. The difference is that our approach is able to represent dependencies among generic components (not only shared libraries) in heterogeneous, distributed environments and it uses this representation to support different kinds of (re)configurations (not only dynamic linking).

9.3.4 Globus and RSL

The Globus project [FK98] provides a "computational grid" [FK99a] integrating heterogeneous distributed resources in a single wide-area system. It supports scalable resource management based

on a hierarchy of resource managers similar to those of *2K* [Yam00]. But, unlike *2K*, Globus is tailored to computationally-intensive applications such as large-scale simulations and teleimmersive applications.

Globus defines an extensible Resource Specification Language (RSL) that is similar to our SPDF. RSL [Glo00] allows Globus users to specify the executables they want to run as well as their resource requirements and environment characteristics.

RSL could be integrated in our system by plugging an *RSLParser* into our Automatic Configuration framework. A fundamental difference between Globus and our work is that we focus on *component-based* applications that are dynamically configured by assembling components fetched from a network repository. In Globus, on the other hand, the user specifies the application to be executed by giving the name of a single executable on the target host file system or by giving a URL from which the executable can be fetched.

Although the Globus design includes a service for managing application code (the Globus Executable Management service or GEM), the current implementation does not include it as a separate service. The name of this service, however, implies that Globus views an application as a single executable, rather than a collection of components that can be dynamically instantiated. Since Globus is one of the most important works in this area, we hope that their system evolves to incorporate support for dynamic component-based applications.

9.4 WYNIWYG in Operating Systems

The idea of building a small system core that could be extended in different ways depending on user and application requirements was explored extensively by previous work on customizable operating systems, microkernels, and exokernels. As discussed below, this research created the basic mechanisms that enable extensions to operating systems, but it did not reach far enough to provide a real "what you need is what you get" model as the one described in section 3.2.

9.4.1 Customizable Operating Systems

Systems like Choices [CIMR93] let administrators build a new instance of the kernel by specifying at compile or link time which mechanisms to use in the various subsystems. System developers

implement a fixed set of modules from which the administrator can choose to construct a certain instance of the kernel. Different modules contain different implementations for virtual memory, file system, networking, and so on. In addition, users can choose to implement a new mechanism by writing a new system module. Unfortunately, implementing a new system module requires intimate knowledge of system internals and conventions. As a result, extending the system is not as easy as expected.

Choices contributed very significantly to the field of operating system design and construction as it showed how principles of object-orientation and design patterns can help to build efficient systems more easily. But it did not address the problems of dynamic configuration or how to express the requirements for constructing the system. The choice of which modules would compose each instance of the kernel were left to a human administrator that edited Makefiles manually.

9.4.2 Microkernels

Mach [Lop91], SPIN [B+95], *μChoices* [LTC96], and L4 [HHL+97], are operating systems with a small core, called microkernel, that runs in privileged mode. A collection of user-level processes, running on top of this small core, support services that are traditionally provided by the kernel in monolithic systems. Thus, administrators can configure the system by selecting different sets of user-level services to be executed in each machine. So, even if all the machines in a network run the same microkernel, their services can be adjusted to the requirements of their users.

Some microkernels also provide mechanisms for modifying their configuration dynamically by adding new modules at runtime. *μChoices* can, for example, dynamically load a new module written in Java into its kernel. The code is then run by a JAVA interpreter that resides in the kernel. SPIN can load modules written in the type safe language Modula-3 and then compiled by a trusted compiler. Synthetix [CAK+96] dynamically generates new code that is specialized to provide improved performance for a particular situation.

9.4.3 Exokernels

The concept of exokernels, first introduced by Engler, Kaashoek and O'Toole [EKJ95, K+97] and further extended by Ballesteros [BHK+99, BHKC99, BKC97], suggests that high-performance,

configurable operating systems can be built by implementing a minimal kernel exporting only low-level hardware abstractions. In these systems, traditional operating system abstractions, such as processes and files, are implemented by user-level libraries. Each application can be linked to different libraries at runtime and, therefore, use customized system abstractions. Exokernels have one major objective: to export the hardware resources safely.

9.4.4 Comparison with our Approach

As described above, customizable operating systems, microkernels, and exokernels supported system configuration using low-level techniques for linking new modules into the operating system in kernel and user space, either statically or dynamically. However, all of them lack a high-level model for operating system configuration and reconfiguration. Although they permit the establishment of different, customized system configurations, they do not have any model for expressing the requirements of user and applications with respect to system modules. All the work is left to the system administrator, who must know about all users, all applications, and have total control over the configuration.

The model presented in this thesis is an integrated solution to the problem of configuring operating systems. Each system module, application, and user defines its requirements explicitly. The system processes all this information and configures the user environment automatically to provide exactly what is needed, no less, no more.

Previous work in operating systems have not addressed a number of problems related to fault-tolerance and dynamic reconfiguration. Using prerequisites and the *ComponentConfigurator* framework, this thesis shows how a system can answer the following questions.

- What system modules are needed to support the user's environment?
- What system modules are needed to run a particular application?
- When a system module is replaced, which other modules are affected?
- How must those other modules react?
- If a system component fails, how can the system detect it and recover gracefully?

By answering these questions, this thesis presents a framework that helps future research efforts to address the following questions.

- What are the consequences of reconfiguring the system?
- What is the cost of reconfiguring the system?
- When (re)configuring the system, which components must be loaded to meet the required quality of service in light of dynamic system conditions?
- When is it time to reconfigure the system?

9.5 Software Architecture

As the complexity of software increases, the design of the overall system architecture (also known as *programming in the large*) becomes even more important than the choices of the individual algorithms and data structures that are used in each component (*programming in the small*). Software architecture is an emerging discipline in software engineering that aims to teach developers to program in the large effectively.

Many of our ideas regarding how to deal with dynamic configuration extend previous work in the field of software architecture. We now discuss some of the most important work in this area and, in Section 9.5.4, explain how our approach differs from them.

9.5.1 Software Buses

James Purtilo's group developed the Polylith system [Pur94] in which components interact with each other through *software buses*. The major idea behind Polylith is that it would allow programmers to implement functional aspects of components separately from their interfacing aspects. Components interface with the bus only, while Polylith provides different implementations of the bus using different communication protocols.

Polylith can be seen as one of the precursors of OMG's CORBA, helping programmers to interconnect software components written in different languages and executing them in heterogeneous

environments. CORBA, in turn, can be seen as a particular implementation of the software bus concept.

Surgeon [HWP93] is a tool for building dynamically reconfigurable applications based on Polylith. Based on *module specifications* and *application specifications*, Polygen [CP91] integrates application modules and builds executable files. Surgeon pre-processes modules that are subject to dynamic reconfiguration instructing Polygen about how the modules must be packaged. At execution time, Surgeon agents, called *catalysts*, manage the reconfiguration process. The catalysts automate the process of updating the bindings between a module being replaced and the modules with which it communicates.

Surgeon works with a limited set of applications in which the modules do not need to participate in the reconfiguration process. As identified in their research, this implies that the modules that are safely reconfigurable using a catalyst must satisfy the following conditions.

1. The module's state can be safely discharged during reconfiguration.

2. Modules must be able to start without receiving state from old modules.

3. There must be no synchronization between the reconfigurable module and its neighbors. For example, modules that use synchronous calls with return values do not satisfy this condition. Thus, modules must use asynchronous messages for communication only.

In her PhD thesis [Hof94], Christine Hofmeister extends this work presenting a framework for dynamic reconfiguration of distributed programs with module participation [HP93]. Although this framework does not support the degree of automation that Surgeon does, it does not require that the component state be discharged. Instead, they apply a technique suggested by Herlihy and Liskov, for transmission of abstract data types [HL82]. Each module is responsible for exporting its internal state, by implementing an `encode` operation, and for importing state through a `decode` operation. In addition, programmers must carefully design application modules so that they can reach a stable state for reconfiguration. When a user requests a module reconfiguration, the module continues its execution until a stable state is achieved. The entire application need not be suspended for reconfiguration; only the affected modules are.

Aster [IB96] is an implementation of a software bus on top of CORBA. Instead of allowing

components to implement and use arbitrary interfaces defined with IDL, Aster components interface with a bus written in CORBA. This bus supports only one interface which contains operations like `sendBus(out msg m)` and `receiveBus(in msg m)`. In section 9.6, we describe ongoing work on supporting dynamic configuration in the Aster environment.

9.5.2 Architectural-Awareness

Sefika addressed the issue of architectural-awareness in the Choices operating system, emphasizing quantitative aspects of the interaction between objects in the Choices kernel [SC95, SSC96a]. The goal was to monitor compliance of the system with its design models [SSC96b].

The motivation of our research, however, is to extend that concept to a broader group of software systems including, not only operating system kernels, but also system libraries, distributed services, and user-level applications. Also, rather than concentrating on the quantitative aspects of the interactions, our work focuses on representing dependencies among software components to support automatic configuration, reliability, and QoS.

9.5.3 Architectural Description Languages

Architectural Description Languages (ADLs) provide a way for practitioners to define the architecture of their systems using a formal notation. Thus, they can present their architectures to other people, reason about them, and even submit the descriptions to automated tools to analyze their consistency or generate code automatically.

ADLs evolved as an extension of Module Interconnection Languages (MILs) first defined in 1975 by DeRemer and Kron [DK76]. MIL compilers were able to build static systems by combining different modules while ensuring system integrity by performing type-checking at compile-time. Early MILs posed a series of restrictions to software developers, requiring that all the modules be written in the same language, be available during system construction, and that each module describe the other modules with which it interacts [SDZ96].

The research carried out by Kramer and Magee at the Imperial College in London provided some of the most significant results in the area of ADLs and dynamic configuration. After earlier work on the Conic [MKS89] and Rex [KMSD92] projects, their most recent research is based on

the Darwin ADL.

Darwin has been used in environments like Regis [MDK94] and CORBA [MTK97] to specify the overall architecture of component-based applications. A Darwin specification defines all the components of an application and the communication interactions between them. At application start time, the middleware loads all the application components and establishes the links between them.

UniCon [SG96, SDZ96] introduced the concept of *Architectural Connectors*, first class objects that mediate the interactions between the components of a system. Components are identified by their *interfaces* and connectors are identified by *protocols*. Applications are built as composite components specified in the UniCon ADL. A composite component specification contains three parts: (1) a list of components to be instantiated, (2) a description of the connections between the components inside the composite, and (3) a binding of the external interface of the composite with the internal configuration. Given this description, UniCon is able to build executable files and scripts that are used to instantiate the whole application.

UniCon connectors implement various communication mechanisms such as UNIX pipes, file I/O, and local and remote procedure calls. Developers may associate an *expert* to each connector. The expert is a piece of executable code that is capable of building the executable code for the connections and checking if a particular use of a connector is consistent. The experts are called by the UniCon compiler as the application is built.

Olan [BABR96, BBB+98] is an execution environment and an architectural description language focusing primarily on distributed applications. The ADL, called Olan Configuration Language (OCL), is an extension of Darwin that includes two new abstractions: the *connectors*, as in UniCon, and *collections*, that represent a dynamic set of objects that can be controlled as a group by the configuration language.

In Olan, applications are organized as a hierarchy of components, where each component may have a set of sub-components. A sub-component can be instantiated dynamically in three ways: (1) at the same time as the enclosing component, (2) when the first communication request reaches the sub-component, or (3) when an explicit instantiation request arrives. These explicit instantiation requests are used to build dynamic groups of components that are stored in *collections*. Thus,

in contrast to traditional ADLs, where the architecture of the system is statically determined by the ADL description, OCL provides a little more dynamism by allowing *collections* of components whose contents are determined according to decisions made at runtime.

The Olan developers implemented a prototype in the Python programming language and made experiments with simple applications, like a configurable electronic mail tool. They targeted Olan towards rapid prototyping and, as a consequence, system performance was low. Other major drawbacks were the lack of a graphical interface and the lack of support for dynamic reconfiguration and dynamic replacement of components. Even with the limitations of its prototype implementation, Olan may be, today, the most complete language for representing software architectures.

9.5.4 Comparison with our Approach

Systems based on architectural connectors like UniCon [SDZ96] and ArchStudio [OT98] and systems based on software buses like Polylith [Pur94] separate issues concerning component functional behavior from component interaction. Our model goes one step further by separating inter-component communication from inter-component dependence. Connectors and software buses require that applications be programmed to a particular communication paradigm. Unlike previous work in this area, our model does not dictate a particular communication paradigm like connectors or buses. It can be used in conjunction with connectors, buses, local method invocation, CORBA, Java RMI, and other methods. As shown in our discussion about *dynamicTAO* (see section 7.1), the model was applied to a legacy system without requiring any modification to its functional implementation or to its inter-component communication mechanisms.

Communication and dependence are often intimately related. But, in many cases, the distinction between inter-component dependence and inter-component communication is beneficial. For example, the quality of service provided by a multimedia application is greatly influenced by the mechanisms utilized by underlying services such as virtual memory, scheduling, and memory allocation (e.g., through the `new` operator). The interaction between the application and these services is often implicit, i.e., no direct communication (e.g. library or system calls) takes place. Yet, if the system infrastructure allows developers to establish and manipulate dependence relationships between the application and these services, the application can be notified of substantial changes

in the state and configuration of the services that may affect its performance.

Research in software architecture and dynamic configuration generally assumes that the operating system is an omnipresent, monolithic black box that can be left out of the discussion; it concentrates on the architecture of individual applications. Previous work in this field does not represent dependencies of application components towards system components, other applications, or services available in the distributed environment. Our approach differs from them in the sense that, for each component, we specify its dependencies on all the different kinds of environment components and we maintain and use these dynamic dependencies at runtime.

Approaches based on software architecture typically rely on global, centralized knowledge of application architecture. In contrast, our method is more decentralized and focuses on more direct component dependencies. We believe that, rather than conflicting with the software architecture approach, our vision complements them by reasoning about *all* the dependencies that may affect reliability, performance, and quality of service.

The final solution to the problem of supporting reliable automatic (re)configuration may reside on the combination of our model with recent work in ADLs and dynamic (re)configuration. This is certainly an important open research problem to be investigated in the future (see chapter 10).

9.6 Dynamic Configuration

Following the early work on dynamic configuration performed by Bloom and Day [BD93] and Hofmeister [HP93], a significant international community organized around the International Conference on Configurable Distributed Systems [PCS98] has been working on the difficult problem of dynamic configuration.

Oreizy, Taylor, and Rosenblum [ORT98, OT98] have identified the need for an explicit representation of the system architecture in order to support consistent, dynamic reconfiguration. In their model, components interact with each other through *Connectors*, which are used both for communication and for representing the dependencies among groups of components. However, because connectors are also used for group communication (for broadcasting asynchronous messages), the model does not make it clear whether two components linked by a connector really depend on

each other. It is a good idea to restrict the freedom of the application developer but, in this case, they also restricted the expressiveness of the model.

Valérie Issarny at IRISA, France developed a Dynamic Reconfiguration Service for CORBA [BISZ98]. In this work, a centralized Dynamic Reconfiguration Manager (DRM) maintains a representation of the distributed system architecture, receives requests for reconfiguration, and invokes the proper distributed components to carry out the reconfiguration. Maintaining application consistency is the responsibility of the application. The infrastructure guarantees that the semantics of the RPC is kept consistent even if components are replaced on-the-fly. Ongoing work includes the integration of this Dynamic Reconfiguration Service in the Aster programming environment (see 9.5.1).

Shrivastava and Wheater [SW98] are currently working on a workflow-based model for distributed applications. In their model, a scripting language [RSW98] is used to provide high-level descriptions of *workflow schemas*. Schemas represent the structure of *tasks* in a distributed application with respect to task composition and inter-task *dependencies*.

The meaning of the term "dependency" in their work is directly related to the workflow model and differs slightly from the one we adopt. A dependency can be either

- a notification dependency (meaning that a task can only start its execution after its dependency has completed its execution) or a
- dataflow dependency (meaning that a task needs to receive input data from another task before starting its execution).

A *workflow execution service* coordinates the execution of the workflow and helps to guarantee that reconfigurations will be performed in a consistent way. Each task is managed by a *task controller* that handles workflow dependencies and helps support reconfiguration.

A task is modeled as having a set of *input sets* (analogous to our *hooks*) and a set of *output sets* (analogous to the *clients* in the component configurator model). Their *workflow schemas* resemble our *prerequisites* slightly. Their *task controller* resembles our component configurator.

The main difference between their model and ours is that their model mixes the representation of data flow, coordination, and inter-component dependence. Their model is suited to expressing

temporal workflow dependencies among components in a certain application. It does not address the interactions between application components and system services.

However, it is important to note that, in Shrivastava and Wheater's model, users are able to specify what the requirements for application tasks must be, so that reconfigurations can be performed in a consistent way. This problem is out of the scope of our work and it could be a PhD thesis in itself. Our framework, however, gives developers the chance to write code to take care of consistency inside the component configurators.

9.7 Application Scenarios

Using the architecture for dependence management and automatic configuration presented in this thesis, we were able to implement two novel applications, the multimedia distribution system and *dynamicTAO*. These applications provided original contributions to the research in their respective Computer Science fields. We now discuss previous work in these fields and emphasize our contributions.

9.7.1 Multimedia Distribution System

Our research on the multimedia distribution system benefits from and builds on previous work on IP-Multicast [Dee89] and the Internet MBone [Eri94, SRL96]. The MBone is a virtual network that is layered on top of the Internet to support routing of IP-Multicast packets. It is composed of islands that can directly support IP-Multicast (e.g., multicast LANs connected by multicast-capable routers), linked by virtual point-to-point links called *tunnels*. The system described in this article can be seen either as extending the MBone capabilities or, (since we also support IP-Multicast) as a set of tools for managing MBone broadcasts. The MBone relies on a multicast distribution engine that is hard-wired into the networking infrastructure, making it difficult to deploy new technologies for distribution. Our approach not only uses user-level entities that can be replaced easily, but it also supports dynamic reconfiguration of running systems, allowing for easy incremental evolution of the communication mechanisms with respect to QoS, security, and reliability.

Amir, McCanne and Katz developed an *active service framework* [AMK98] that provides support for the dynamic instantiation of server agents (or *servents*), providing the desired service, in a

collection of distributed nodes running their host manager (HM) daemon. They used this framework to implement a Media Gateway (MeGa) service [AMZ95]. Like our Reflector, the MeGa service can be used to transcode multimedia streams. In fact, their research has focused on algorithms for efficient transcoding and down-sizing and, more recently, on adaptive bandwidth allocation algorithms [AMK97] and on the dynamic instantiation of media gateways between MBone sessions to deal with heterogeneous networks. Our research on the Reflector architecture shares some concerns with their active service framework (which has some similarities with our Automatic Configuration Service) and with their MeGa service (which could be implemented in our architecture with customized *Connection* classes loaded into the Reflector). Nevertheless, we also focus on providing scalable multimedia distribution using multiple communication protocols, reconfiguration of the network topology to deal with failures, and scalable code distribution and dynamic reconfiguration.

Baldi, Picco, and Risso designed a videoconference system based on active networks [BPR98]. Their proposed architecture allows clients to customize their videoconference server (or Reflector) by uploading mobile Java code. The main difference is that, in their approach, the Reflector is designed to run in active network routers while ours is designed as a user-level application to be executed on networked workstations. Our infrastructure for mobile reconfiguration agents plays the same role as the active network in their system. We chose to implement our system in C++ because our experience shows that Java is not yet ready to provide the high-performance and predictability that most QoS-sensitive applications require.

9.7.2 *dynamicTAO*

Recent research in middleware have identified a number of limitations on first-generation CORBA implementations. This has led to ORB extensions for dealing with specific aspects such as real-time [HLS97b], group communication [MS97], and fault-tolerance [Maf95]. The goal of *dynamicTAO*, on the other hand, is to provide a generic infrastructure in which different kinds of customizations can be performed using reflection.

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 [HBG+98], 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, the difference being 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 either of the two approaches.

Blair and others at Lancaster University has proposed a reflective architecture for next generation middleware [BCRP98, CB99, BCCD00]. 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 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 group is now investigating (1) an efficient implementation of their reflective architecture using lightweight components in a COM/C++ environment and (2) the use of *component frameworks* [PCB00] to manage the development of meta-interfaces in a robust and meaningful way.

COMERA [WL98] (COM Extensible Remote Architecture) provides a framework based on 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 a componentized architecture design that allows the use of user-specified components. Like in *dynamicTAO*, by using COMERA, system developers can customize the middleware according to application requirements.

Our work in *dynamicTAO* is an extension of previous work in reflective middleware carried out by Ashish Singhai in our research group. Singhai developed a prototype of a reflective ORB that

could be customized at startup time by selecting policies for real-time invocation, fault-tolerance, and load balancing [SSC97]. His PhD thesis presented *Quarterware*, a middleware construction technique and a toolkit to help developers build customized middleware systems. Using Quarterware components, one can build communication middleware systems as diverse as MPI and Java RMI [SSC98b, Sin99].

The middleware community is starting to realize that static middleware infrastructures are not appropriate for the highly dynamic environments of the future. A trend towards more flexible middleware architectures could be observed in the recent IFIP/ACM Middleware conference [SC00] and its Workshop on Reflective Middleware [KS00].

Chapter 10

Future Work

As noted by a forgotten scientist, "We live on an island surrounded by a sea of ignorance. As our island of knowledge grows, so does the shore of our ignorance". The work on this thesis contributed immensely for the enlargement of the shore of the author's ignorance. In the following sections, we discuss six topics for future research that we consider particularly important.

10.1 Libraries of Customized Component Configurators

The software infrastructure resulting from this thesis includes a few basic types of component configurators providing some basic functions. According to our model, component developers are responsible for implementing application-specific (or component-specific) policies by implementing customized subclasses of component configurators.

In future research, we intend to investigate the possibility of developing various libraries containing customized versions of component configurators that can be reused in different applications and systems. In the same way that component-based programming lets developers build complex applications by selecting appropriate off-the-shelf components, we envision a future in which application developers will be able to add a wide range of non-functional features, simply by selecting appropriate versions of component configurators.

In the near future, we will study the design of libraries of component configurators supporting reconfiguration, fault-tolerance, adaptability, scalability, and migration.

10.2 Automatic Prerequisite Generation and Verification

In the current implementation of our architecture, we rely on developers to write consistent prerequisite specifications for the components and applications they create. It would be desirable to automate the creation of prerequisite specifications and to develop tools to verify their correctness. This is a difficult problem that will require many years to be solved. Tools like QualProbes [LN00] address part of the problem by automating the creation of prerequisites specifying application resource requirements.

The prerequisite resolver should also be extended to detect inconsistencies on the prerequisite specifications. These inconsistencies could lead to deadlocks or infinite loops, compromising system liveness.

10.3 Dynamic Adaptability

Although the prerequisites are primarily used for loading new components into the system and making sure that their quality of service expectations are met, they are also useful for dynamic adaptation in case of changes in resource availability. Resource requirements expressed in the prerequisites should specify ranges of acceptable service. A video-on-demand application, for example, could specify that it requires a network bandwidth of 1.2Mbps on average, but that it may utilize peak rates of up to 2Mbps. Finally, it could add that, even though 1.2Mbps is the desirable average bandwidth, it would still be able to function by using as little as 53Kbps by changing the characteristics of the video stream. In that case, the application would be able to support mobile computers moving from ATM to wireless, to modem connections, adapting to these changes dynamically. Thus, prerequisites should be available to the system at runtime so that it can reorganize its allocation of resources to fulfill better the requirements of all applications sharing the system.

An interesting topic for future research is how to use, modify, and generate prerequisite specifications at runtime with minimal interference in performance.

10.4 Integration with ADLs

As noted in section 9.5.4, we believe that our component configurators and Architectural Description Languages address complementary issues. Our model is dynamic and focuses on localized inter-component dependence, while ADLs are static and focus on global architectures. Combining both approaches in a single model is a fascinating challenge. Since the two approaches are not completely orthogonal to each other, their integration is non-trivial.

10.5 Component Repository

The current implementation of the *2K* component repository is very simple and works as a CORBA repository of files that can be organized in a hierarchical directory structure. As future work, it would be important to develop a more sophisticated component repository with support for security, version control, garbage collection, and fault-tolerance.

To provide scalability it will be required to replicate the repository and offer mechanisms to update collections of repositories with new components. This could be achieved with the help of our mobile agents infrastructure. In this case, it would be important to investigate policies for maintaining the consistency among the repository replicas.

The security requirements include (1) mutual authentication between the repository and its users, (2) controlling the access to the repository contents, and (3) digitally signing the components stored in the repository.

10.6 Security

In networked environments, it becomes necessary to secure the configuration system from unauthorized access. A hostile agent that obtains access to component configurators may be able to totally disrupt system activities. Even read-only access may be dangerous as sensitive information about the internal structure of an institution's system may be stolen. Therefore, it is important to provide support for controlling the access to the configuration system. In some cases, it is also desirable to avoid eavesdropping by encrypting the messages exchanged by components and component configurators, for example, those containing reconfiguration events.

To support security in environments such as Java with RMI, it is necessary to extend the configuration model, making it security-aware. On the other hand, in environments supporting reflection and in CORBA, it is possible to define security policies and deploy security mechanisms without modifications to our model.

Using the CORBA Security Service [OMG98], it is possible to add *message-level interceptors* into the ORB so that data exchanged between CORBA objects are properly encrypted. In addition, *request-level interceptors* can control the access to each individual operation on component configurators based on who is issuing the call, based on capabilities, or based on any other customized mechanism defined by the programmer.

10.7 Concurrency

In multi-threaded and multi-process environments, we must take additional care with regard to reliability and consistency since two threads accessing the same object concurrently may leave the system in an inconsistent state or cause its failure. One of the *ComponentConfigurator* subtypes offered by our framework uses locks to protect the configurator from simultaneous updates by multiple clients. These locks can also be used by clients to perform a sequence of operations on a single configurator without interference from other clients.

At the present moment, our framework does not provide any guarantee that a group of reconfiguration actions performed in a collection of configurators will be processed as a single unit. In a CORBA environment, one could coordinate the access to distributed configurators by using the standard Concurrency Control Service [OMG98]. Ideally, a reconfiguration system should provide support for grouping operations into atomic transactions satisfying the ACID properties, i.e., atomicity, consistency, isolation, and durability. This requires further research but it could be achieved with the help of the CORBA Object Transaction Service [OMG98].

Chapter 11

Conclusions

Dependence management is probably the most crucial problem to be resolved before operating systems and middleware are able to provide automatic configuration of component-based applications and services. Only then will we be able to remove the burden of system configuration from users and administrators.

In this thesis, we presented an integrated architecture for managing dependencies in component-based distributed systems. Our experience with the architecture has been very fruitful, encouraging further enhancements. Future work in our and other research groups will show whether the architecture is suitable for complex, distributed, component-based systems developed by other programmers.

11.1 Original Contributions

The novel architecture presented in this thesis provided the first three original contributions listed below. The deployment of the architecture in two example applications provided the two additional contributions.

1. An Automatic Configuration Service for component-based distributed systems(described in Chapter 4 and [KCC99]).

2. A framework for representation and management of dependencies in component-based distributed systems (described in Chapter 5 and [KC00]).

3. A mobile agent infrastructure for dynamic reconfiguration and code distribution (described

in Chapter 6 and [KGCM99]).

4. *dynamicTAO*, a CORBA-compliant reflective ORB [KRL+00].

5. A dynamically configurable, reliable multimedia distribution system [KC99, KCN00].

11.2 Perspectives

Within the next few years, we will witness great changes in our everyday computing environments, including higher degrees of dynamism, composability, mobility, heterogeneity, and interactions among heterogeneous computing devices connected to global networks. Probably, the major change will be that computers will no longer be heavy boxes on our desks, but instead, will be everywhere, at home, at the supermarket, at the hospital, on the street, from our glasses to soda cans. One of the few things that are clear at this moment is that conventional middleware and operating system architectures are inadequate to support these highly dynamic, heterogeneous environments of the future. We believe that the contributions presented on this thesis constitute a few small bricks that will help the construction of the big building of next century's software technology.

Above all, we scientists must not forget that, while the amazing technology of the 21^{st} century will certainly improve the quality of life of the top 20% wealthiest people, we live in a planet where the large majority of the population live in deep poverty. This is a political problem that cannot be solved by science and technology alone; but scientists must not forget that and always have in mind that our role in society is much more than improving systems performance.

References

[AMK97] Elan Amir, Steven McCanne, and Randy Katz. Receiver-driven Bandwidth Adaptation for Light-weight Sessions. In *Proceedings of the ACM Multimedia Conference*, Seattle, WA, November 1997.

[AMK98] Elan Amir, Steven McCanne, and Randy Katz. An Active Service Framework and its Application to Real-time Multimedia Transcoding. In *Proceedings of the ACM SIGCOMM Conference*, Vancouver, BC, September 1998.

[AMZ95] E. Amir, S. McCanne, and H. Zhang. An Application-level Video Gateway. In *Proceedings of ACM Multimedia*, San Francisco, CA, November 1995.

[B+95] B. N. Bershad et al. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. of the 15th SOSP*. ACM, December 1995.

[BA00] Lodewijk M. Bergmans and Mehmet Aksit. Aspects and Crosscutting in Layered Middleware Systems. In *IFIP/ACM Middleware'2000 Workshop on Reflective Middleware*, pages 23–25, Palisades, NY, April 2000.

[BABR96] L. Bellissard, S. Ben Atallah, F. Boyer, and M. Riveill. Distributed Application Configuration. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 579–585, Hong-Kong, May 1996. IEEE Computer Society.

[BBB+98] R. Balter, L. Bellissard, F. Boyer, M. Riveill, and J.Y. Vion-Dury. Architecturing and Configuring Distributed Applications with Olan. In *Proc. IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, pages 241–256, The Lake District, UK, September 1998. Springer-Verlag.

[BCCD00] Gordon Blair, Geoff Coulson, Fabio Costa, and Hector A. Duran. On the Design of Reflective Middleware Platforms. In *IFIP/ACM Middleware'2000 Workshop on Reflective Middleware*, pages 3–5, Palisades, NY, April 2000.

[BCRP98] 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.

[BD93] T. Bloom and M. Day. Reconfiguration and Module Replacemente in Argus: Theory and Practice. *IEE Software Engineering Journal*, 8(2):102–108, March 1993.

[BHK+99] Francisco J. Ballesteros, Christopher Hess, Fabio Kon, Sergio Arévalo, and Roy H. Campbell. Object Orientation in Off++ - A Distributed Adaptable μKernel. In *ECOOP'99 Workshop on Object Orientation and Operating Systems*, pages 49–53, Lisbon, June 1999.

[BHKC99] Francisco J. Ballesteros, Christopher Hess, Fabio Kon, and Roy H. Campbell. The Design and Implementation of the Off++ and vOff++ μkernels. Technical Report UIUCDCS-R-98-2086, Department of Computer Science, University of Illinois at Urbana-Champaign, March 1999.

[BISZ98] Christophe Bidan, Valérie Issarny, Titos Saridakis, and Apostolos Zarras. A Dynamic Reconfiguration Service for CORBA. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, Annapolis, Maryland, USA, May 1998.

[BKC97] Francisco J. Ballesteros, Fabio Kon, and Roy H. Campbell. A Detailed Description of Off++, a Distributed Adaptable Microkernel. Technical Report UIUCDCS-R-97-2035, University of Illinois at Urbana-Champaign, August 1997. Also available at `http://choices.cs.uiuc.edu/2k/off++`.

[BKR98] Jonathan Bredin, David Kotz, and Daniela Rus. Market-based Resource Control for Mobile Agents. In *Proceedings of the Second International Conference on Autonomous Agents*, pages 197–204, May 1998.

[Bou99a] Sahra Bouchenak. Capture et Restauration du Contexte d'Exécution d'un Thread dans l'Environnement Java. In *1ère Conférence Française sur les Systèmes d'Exploitation (CFSE'1)*, Rennes, June 1999. ACM-SIGOPS.

[Bou99b] Sahra Bouchenak. Pickling Threads State in the Java System. In *Proceedings of the Third European Research Seminar on Advances in Distributed Systems (ERSADS'99)*, Madeira Island, April 1999.

[BPR98] Mario Baldi, Gian Pietro Picco, and Fulvio Risso. Designing a Videoconference System for Active Networks. In *Proceedings of the Second International Workshop on Mobile Agents (MA'98)*, pages 273–284, September 1998.

[BPSK96] Hari Balakrishnan, Venkata Padmanabhan, Srinivasan Seshan, and Randy Katz. A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. In *Proceedings of the ACM SIGCOMM Conference*, Stanford, CA, August 1996.

[CAK+96] Crispin Cowan, Tito Autrey, Charles Krasic, Calton Pu, and Jonathan Walpole. Fast Concurrent Dynamic Linking for an Adaptive Operating System. In *International Conference on Configurable Distributed Systems (ICCDS'96)*, Annapolis MD, USA, May 1996.

[Can97] Saul Candib. *Using ActiveX Technology to Create Programmable Applications.* Microsoft Press, Redmond, 1997.

[CB99] 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.

[Chi95] Shigeru Chiba. A Metaobject Protocol for C++. In *Proceedings of the OOPSLA'95*, pages 285–299, October 1995.

[CHY+98] P. E. Chung, Y. Huang, S. Yajnik, D. Liang, J. C. Shih, C. Y. Wang, and Y. M. Wang. DCOM and CORBA Side by Side, Step By Step, and Layer by Layer. *C++ Report*, January 1998.

[CIMR93] Roy Campbell, Nayeem Islam, Peter Madany, and David Raila. Designing and Implementing Choices: an Object-Oriented System in C++. *Communications of the ACM*, 36(9):117–136, September 1993.

[CKB+00] Dulcineia Carvalho, Fabio Kon, Francisco Ballesteros, Manuel Román, Roy Campbell, and Dennis Mickunas. Management of execution environments in 2k. In *Proceedings of the Seventh International Conference on Parallel and Distributed Systems (ICPADS'2000)*. IEEE Computer Society, July 2000.

[CL99] M. C. Chan and A. A. Lazar. Designing a CORBA-based High Performance Open Programmable Signalling System for ATM Switching Platforms. *IEEE Journal on Selected Areas in Communications*, 17(9), September 1999.

[CNM98] Roy H. Campbell, Klara Nahrstedt, and M. Dennis Mickunas. 2K: A Component-Based Network-Centric Operating System. Project home page: `http://choices.cs.uiuc.edu/2K`, 1998.

[CP91] J. Callahan and James Purtilo. A Packaging System for Heterogeneous Execution Environments. *IEEE Transactions*, SE-17:626–635, 1991.

[CTCL95] Zhigang Chen, See-Mong Tan, Roy H. Campbell, and Yongcheng Li. Real Time Video and Audio in the World Wide Web. In *Fourth International World Wide Web Conference*, Boston, December 1995. Also published in World Wide Web Journal, Volume 1 No 1, January 1996.

[DC90] S. Deering and D. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANS. *ACM Transactions on Computer Systems*, pages 85–110, May 1990.

[Deb00] The Debian Project. Debian linux operating system home page, 2000. `http://www.debian.org`.

[Dee89] S. Deering. Host Extensions for IP Multicasting. RFC 1112, August 1989.

[Den97] Adam Denning. *ActiveX Controls Inside Out*. Microsoft Press, Redmond, second edition, 1997.

[DK76] Frank DeRemer and Hans H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86, June 1976.

[Don97] Terry Donoghue. *Discovering OpenStep: A Developer Tutorial*. Apple Computer, Inc, Cupertino, California, 1997.

[EKJ95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995.

[EMS91] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *CAMELOT and AVALON: A Distributed Transaction Facility*. Morgan Kaufmann Publishers, Inc., 1991.

[End94] Markus Endler. A Language for Generic Dynamic Configuration of Distributed Programs. In *Proceedings of the 12th Brazilian Symposium on Computer Networks*, pages 175–187, Curitiba, May 1994.

[Eri94] Hans Eriksson. MBone: The Multicast Backbone. *Communications of the ACM*, 37(8):54–60, August 1994.

[ESS+00] Markus Endler, Dilma M. Silva, Francisco J. S. Silva, Ricardo C. A. Rocha, and Marcos A. Moura. SIDAM Project: Overview and Preliminary Results. In *Proceedings of the Workshop on Wireless Comunication, Brazilian Symposium on Computer Networks*, 2000.

[FK98] I. Foster and C. Kesselman. The Globus Project: A Status Report. In *Proceedings of the IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, 1998.

[FK99a] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, San Francisco, 1999.

[FK99b] Svend Frølund and Jari Koistinen. Quality of Service Aware Distributed Object Systems. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technology and Systems (COOTS'99)*, pages 69–83, San Diego, May 1999.

[Flo71] Ivan Flores. *Job Control Language and File Definition.* Prentice-Hall, Englewood Cliffs, NJ, 1971.

[Fun98] Stefan Funfroken. Transparent Migration of Java-Based Mobile Agents – Capturing and Reestablishing the State of Java Programs. In *Proceedings of the Second International Workshop on Mobile Agents (MA'98)*, pages 26–37, September 1998.

[GC96] Brendan Gowing and Vinny Cahill. Meta-Object Protocols for C++: The Iguana Approach. In *Proceedings of Reflection '96*, pages 137–152, San Francisco, USA, April 1996.

[GCE+97] M. P. Golombek, R. A. Cook, T. Economou, W. M. Folkner, A. F. Haldemann, P. H. Kallemeyn, J. M. Knudsen, R. M. Manning, H. J. Moore, T. J. Parker, R. Rieder, J. T. Schofield, P. H. Smith, and R. M. Vaughan. Overview of the Mars Pathfinder Mission and Assessment of Landing Site Predictions. *Science*, 278(5344):1734–42, December 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Object-Oriented Software.* Addison-Wesley, 1995.

[Glo00] The Globus Project. *Globus Resource Specification Language RSL v1.0*, 2000. Available at `http://www.globus.org/gram/rsl_spec1.html`.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Mateo, California, 1993.

[GWTB96] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wiley Hacker. In *Proceedings of the USENIX Security Symposium*, July 1996.

[Ham97] Graham Hamilton. *JavaBeans specification.* Sun Microsystems, 1997. Available at `http://java.sun.com/beans/docs`.

[HBC00] Christopher K. Hess, Francisco J. Ballesteros, and Roy H. Campbell. An Adaptable Distributed File Service. In *Proceedings of the ECOOP PhD Workshop on Object Oriented Systems (PHDOOS'00)*, Cannes, France, June 2000.

[HBG+98] 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.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The Performance of μ-Kernel-Based Systems. In *Proceedings of the 16th Symposium on Operating Systems Principles*, Saint Malo, France, October 1997. ACM.

[HL82] M. Herlihy and B. Liskov. A Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 2:527–551, 1982.

[HLS97a] Timothy Harrison, David Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Object Event Service. In *Proceedings of OOPSLA '97*, Atlanta, Georgia, October 1997.

[HLS97b] 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.

[Hof94] Christine R. Hofmeister. *Dynamic Reconfiguration of Distributed Applications.* PhD thesis, University of Maryland, Department of Computer Science, January 1994. Technical Report CS-TR-3210.

[Hoh98] Fritz Hohl. Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts. In G. Vigna, editor, *Mobile Agents and Security*, volume LNCS 1419, pages 92–113. Springer-Verlag, 1998.

[HP93] Christine Hofmeister and James M. Purtilo. Dynamic Reconfiguration in Distributed Systems: Adapting Software Modules for Replacement. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 101–110, Pittsburgh, Pennsylvania, USA, 1993.

[HWP93] Christine Hofmeister, E. White, and James M. Purtilo. SURGEON: A Packager for Dynamically Reconfigurable Distributed Applications. *IEE Software Engineering Journal*, 8(2):95–101, March 1993.

[IB96] Valérie Issarny and Christophe Bidan. Aster: A Corba-Based Software Interconnection System Supporting Distributed System Customization. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems (CDS'96)*, Annapolis, Maryland, USA, 1996.

[ILY95] Jun-ichiro Itoh, Rodger Lea, and Yasuhiko Yokote. Using Meta-objects to Support Optimization in the Apertos Operating System. In *USENIX Conference on Object-Oriented Technologies (COOTS)*, June 1995.

[Iye99] Arjun Chandrasekar Iyer. Persistent Object Service Framework Using Component Configuration Model. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1999.

[JS95] Eric Jul and Bjarne Steensgaard. Object and Native Code Thread Mobility among Heterogeneous Computers. In *Proceedings of the 15th ACM SOSP)*, Copper Mountain, Colorado, December 1995.

[JS97] Prashant Jain and Douglas C. Schmidt. Dynamically Configuring Communication Services with the Service Configuration Pattern. *C++ Report*, 9(6), June 1997.

[K+97] M. Frans Kaashoek et al. Application Performance and Flexibility on Exokernel Systems. In *Proc. 16th SOSP*, Saint Malo, France, October 1997. ACM.

[KC99] Fabio Kon and Roy Campbell. A Framework for Dynamically Configurable Multimedia Distribution. In *Proceedings of the ECOOP'99 Workshop for PhD Students in Object Oriented Systems*, pages 118–127, Lisbon, June 1999.

[KC00] Fabio Kon and Roy H. Campbell. Dependence Management in Component-Based Distributed Systems. *IEEE Concurrency*, 8(1):26–36, January-March 2000.

[KCC99] Fabio Kon, Dulcineia Carvalho, and Roy Campbell. Automatic Configuration in the 2K Operating System. In *Proceedings of the ECOOP'99 Workshop on Object Orientation and Operating Systems*, pages 10–14, Lisbon, June 1999.

[KCM+99] Fabio Kon, Roy H. Campbell, M. Dennis Mickunas, Klara Nahrstedt, and Francisco J. Ballesteros. 2K: A Distributed Operating System for Heterogeneous Environments. Technical Report UIUCDCS-R-99-2132, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1999.

[KCM+00] Fabio Kon, Roy H. Campbell, M. Dennis Mickunas, Klara Nahrstedt, and Francisco J. Ballesteros. 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC'9)*, Pittsburgh, August 2000.

[KCN00] Fabio Kon, Roy H. Campbell, and Klara Nahrstedt. Using Dynamic Configuration to Manage A Scalable Multimedia Distribution System. *Computer Communication Journal (Special Issue on QoS-Sensitive Distributed Systems and Applications)*, Fall 2000.

[KCS+99] Fabio Kon, Roy H. Campbell, Balaji Srinivasan, Ramesh Chandra, and Arun Viswanathan. Dynamic Reconfiguration of Scalable Internet Systems with Mobile Agents. Technical Report UIUCDCS-R-99-2105, Department of Computer Science, University of Illinois at Urbana-Champaign, March 1999.

[KCT+98] 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.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[KG99] David Kotz and Robert S. Gray. Mobile Agents and the Future of the Internet. *ACM Operating Systems Review*, 33(3):7–13, July 1999.

[KGA+00] Fabio Kon, Binny Gill, Manish Anand, Roy H. Campbell, and M. Dennis Mickunas. Secure Dynamic Reconfiguration of Scalable CORBA Systems with Mobile Agents. In *Proceedings of the IEEE Joint Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA'2000)*, Zurich, September 2000.

[KGCM99] 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.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Also in Springer-Verlag LNCS 1241.

[KMSD92] J. Kramer, J. Magee, M. Sloman, and N. Dulay. Configuring Object-based Distributed Programs in REX. *IEE Software Engineering Journal*, 7(2), 1992.

[Kon98] Fabio Kon. Distributed Configuration Protocol. Project home page: `http://choices.cs.uiuc.edu/2k/DCP`, June 1998.

[KP88] Glen E. Krasner and Stephen T. Pope. A Cookbook for Using the Model-View-Controller Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, pages 26–49, 1988.

[Kra90] Jeff Kramer. Configuration Programming – a Framework for the Development of Distributed Systems. In *Proceedings of the IEEE International Conference on Computer Systems and Software Engineering*, pages 374–384, Tel Aviv, May 1990.

[KRL+00] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121–143, New York, April 2000. Springer-Verlag.

[KS00] Fabio Kon and Katia Barbosa Saikoski, editors. *IFIP/ACM Middleware'2000 Workshop on Reflective Middleware*, New York, April 2000. Gordon Blair and Roy Campbell (co-chairs).

[KSC+98] 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.

[LBS+98] J. P. Loyall, D. E. Bakken, R. E. Schantz, J. A. Zinky, D. A. Karr, R. Vanegas, and K. R. Anderson. QoS Aspect Languages and Their Runtime Integration. In *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, Pittsburgh, Pennsylvania, May 1998.

[LN00] Baochun Li and Klara Nahrstedt. QualProbes: Middleware QoS Profiling Services for Configuring Adaptive Applications. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 256–272, New York, April 2000. Springer-Verlag.

[Loc94] H. W. Lockhart Jr. *OSF DCE Guide to Developing Distributed Applications*. McGraw-Hill, Inc., New York, 1994.

[Lop91] Keith Lopere. Mach 3 kernel principles. *Open Software Foundation*, 1991.

[LTC96] W. S. Liao, S. Tan, and R. H. Campbell. Fine-grained, Dynamic User Customization of Operating Systems. In *Proc. 5th Int. Workshop on Object-Orientation in Operating Systems*, pages 62–66, Seattle, October 1996.

[Lun98] Charlotte Pii Lunao. Is Composition of Metaobjects = Aspect-Oriented Programming. In *ECOOP Aspect-Oriented Programming Workshop*, Brussels, July 1998.

[Maf95] 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.

[MB+99] Dejan Milojicic, Bill Bolosky, et al. Operating Systems – Now and in the Future. *IEEE Concurrency*, 7(1):12–21, January-March 1999.

[MDK94] Jeff Magee, Naranker Dulay, and Jeff Kramer. Regis: A Constructive Development Environment for Distributed Programs. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 1(1):37–47, 1994.

[ME98] David MacKenzie and Ben Elliston. *Autoconf - Creating Automatic Configuration Scripts*. Free Software Foundation, December 1998. edition 2.13.

[MG99] K. Moazami-Goudarzi. *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*. PhD thesis, Imperial College London, March 1999.

[MGLNS94] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Fragmented Objects for Distributed Abstractions. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, July 1994.

[MKS89] J. Magee, J. Kramer, and M. Solman. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, 15(6):663–675, 1989.

[MS97] Silvano Maffeis and Douglas C. Schmidt. Constructing Reliable Distributed Communication Systems with CORBA. *IEEE Communications Magazine*, 14(2), February 1997.

[MTK97] Jeff Magee, Andrew Tseng, and Jeff Kramer. Composing Distributed Objects in CORBA. In *Proceeding of the 3rd International Symposium on Autonomous Decentralized Systems (ISADS'97)*, Berlin, April 1997.

[NhCN98] Klara Nahrstedt, Hao hua Chu, and Srinivas Narayan. QoS-aware Resource Management for Distributed Multimedia Applications. *Journal of High-Speed Networking, Special Issue on Multimedia Networking*, 7:227–255, 1998.

[NL97] J. Nieh and M. S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, Saint Malo, France, October 1997. ACM.

[NWX00] Klara Nahrstedt, Duangdao Wichadakul, and Dongyan Xu. Distributed QoS Compilation and Runtime Instantiation. In *Proceedings of the IEEE/IFIP International Workshop on QoS (IWQoS'2000)*, Pittsburgh, June 2000.

[OB99] Alexandre Oliva and Luiz Eduardo Buzato. The Design and Implementation of Guaraná. In *Proc. 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, pages 203–216, San Diego, CA, May 1999.

[Obr98] Katia Obraczka. Multicast Transport Mechanisms: A Survey and Taxonomy. *IEEE Communications Magazine*, January 1998.

[OHE96] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, inc, 1996.

[OMG98] OMG. *CORBAservices: Common Object Services Specification*. Object Management Group, Framingham, MA, 1998. OMG Document 98-12-09.

[OMG99] OMG. *CORBA Components*. Object Management Group, Framingham, MA, 1999. OMG Document orbos/99-07-01.

[ORT98] Peyman Oreizy, David S. Rosenblum, and Richard N. Taylor. On the Role of Connectors in Modeling and Implementing Software Architectures. Technical Report UCI-ICS-98-04, Department of Information and Computer Science, University of California, Irvine, February 1998.

[OT98] Peyman Oreizy and Richard N. Taylor. On the Role of Software Architectures in Runtime System Reconfiguration. In *Proceedings of the 4th International Conference on Configurable Distributed Systems (CDS'98)*, Annapolis, Maryland, USA, May 1998.

[PCB00] Nikos Parlavantzas, Geoff Coulson, and Gordon Blair. Applying Component Frameworks to Develop Flexible Middleware. In *IFIP/ACM Middleware'2000 Workshop on Reflective Middleware*, pages 6–7, Palisades, NY, April 2000.

[PCS98] Jim Purtilo, Robert Cole, and Rick Schlichting, editors. *Fourth International Conference on Configurable Distributed Systems*. IEEE, May 1998.

[Pur94] James Purtilo. The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.

[RC00] Manuel Román and Roy H. Campbell. Gaia: An Operating System to Enable Active Spaces, 2000. Submitted to the 9th ACM SIGOPS European Workshop.

[RKC99] Manuel Román, Fabio Kon, and Roy H. Campbell. Design and Implementation of Runtime Reflection in Communication Middleware: the *dynamicTAO* Case. In *Proceedings of the ICDCS'99 Workshop on Middleware*, pages 122–127, Austin, TX, June 1999.

[RMKC00] Manuel Román, Dennis Mickunas, Fabio Kon, and Roy H. Campbell. LegORB and Ubiquitous CORBA. In *Proceedings of the IFIP/ACM Middleware'2000 Workshop on Reflective Middleware*, pages 1–2, Palisades, NY, April 2000.

[RS98] James Riordan and Bruce Schneier. Environmental Key Generation Towards Clueless Agents. In G. Vigna, editor, *Mobile Agents and Security*, volume LNCS 1419, pages 15–24. Springer-Verlag, 1998.

[RSW98] F. Ranno, S. K. Shrivastava, and S. M. Wheater. A Language for Specifying the Composition of Reliable Distributed Applications. In *Proceeding of the 18th International Conference on Distributed Computing Systems (ICDCS '98)*, Amsterdam, May 1998.

[SC95] Mohlalefi Sefika and Roy H. Campbell. An Open Visual Model for Object-Oriented Operating Systems. In *Fourth International Workshop on Object-Orientation in Operating Systems*, Lund, Sweden, August 1995. IEEE Computer Society Press.

[SC99] Douglas C. Schmidt and Chris Cleeland. Applying Patterns to Develop Extensible ORB Middleware. *IEEE Communications Magazine Special Issue on Design Patterns*, 37(4):54–63, May 1999.

[SC00] Joseph Sventek and Geoffrey Coulson, editors. *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, New York, April 2000. Springer-Verlag.

[SCFJ00] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. Internet Draft, revision of RFC 1889, January 2000.

[Sch93] Douglas C. Schmidt. The ADAPTIVE Communication Environment. In *Proceedings of the Sun User Group Conference*, San Jose, California, December 1993.

[SDZ96] Mary Shaw, R. DeLine, and G. Zelesnik. Abstractions and Implementations for Architectural Connections. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems (CDS'96)*, Annapolis, Maryland, USA, May 1996.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[SGE98] Dilma Menezes da Silva, Marco Dimas Gubitoso, and Markus Endler. Sistemas de Informação Distribuídos para Agentes Móveis. In *Proceedings of the XXV Integrated Seminars in Software and Hardware (SEMISH'98)*, pages 125–140, Belo Horizonte, Brazil, August 1998. SBC.

[SGH+89] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An Object-Oriented Operating System — Assessment and Perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[SGM89] Marc Shapiro, Philippe Gautron, and Laurence Mosseri. Persistence and Migration for C++ Objects. In Stephen Cook, editor, *ECOOP'89, Proc. of the Third European Conf. on Object-Oriented Programming*, British Computer Society Workshop Series, pages 191–204, Nottingham (GB), July 1989. The British Computer Society, Cambridge University Society.

[Sha98] Marc Shapiro. Personal communication, July 1998.

[Sin99] Ashish Singhai. *Quarterware: A Middleware Toolkit of Software RISC Components*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1999.

[Smi84] Brian C. Smith. Reflection and Semantics in LISP. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, January 1984.

[SPZ98] Flávio Assis Silva and Radu Popescu-Zeletin. An Approach for Providing Mobile Agent Fault Tolerance. In *Proceedings of the Second International Workshop on Mobile Agents (MA'98)*, pages 14–25, September 1998.

[SRL96] Kevin Savetz, Neil Randall, and Yves Lepage. *MBone: multicasting tomorrow's Internet*. IDG Books, Foster City, CA, 1996.

[SSC96a] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Architecture-Oriented Visualization. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications(OOPSLA'96)*, 1996.

[SSC96b] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring Compliance of a Software System With Its High-Level Design Models. In *18th International Conference on Software Engineering*, Berlin, Germany, March 25-26 1996.

[SSC97] 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.

[SSC98a] Aamod Sane, Ashish Singhai, and Roy Campbell. End-to-End Considerations in Framework Design. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP)*, July 1998.

[SSC98b] 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.

[Sun97] Sun Microsystems. *Linker and Libraries*, 1997. On-line document available at `http://docs.sun.com`.

[SVSB99] P. Sinha, N. Venkitaraman, R. Sivakumar, and V. Bharghavan. WTCP: A Reliable Transport Protocol for Wireless Wide-Area Networks. In *Proceedings of ACM Mobicom*, Seattle, WA, August 1999.

[SW91] Frank Schmuck and Jim Wyllie. Experience with Transactions in QuickSilver. *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 239–53, 1991.

[SW98] S. K. Shrivastava and S. M. Wheater. Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications. In *Proceeding of the 4th International Conference on Configurable Distributed Systems (CDS'98)*, Annapolis, Maryland, May 1998.

[VB99] Jan Vitek and Ciaran Bryce. The JavaSeal Mobile Agent Kernel. In *Proceedings of the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents (ASA/MA'99)*, October 1999.

[Vig98] Giovanni Vigna, editor. *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[Wal98] Jim Waldo. Jini Architecture Overview. Available at `http://java.sun.com/products/jini/whitepapers`, 1998.

[Wei92] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):94–104, September 1992.

[WL98] Y. M. Wang and Woei-Jyh Lee. COMERA: COM Extensible Remoting Architecture. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'98)*, April 1998.

[WSA+95] Roy Want, Bill N. Schilit, Norman I. Adams, Rich Gold, Karin Petersen, David Goldberg, John R. Ellis, and Mark Weiser. An Overview of the ParcTab Ubiquitous Computing Experiment. *IEEE Personal Communications*, pages 28–43, December 1995.

[XWN00] 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.

[Yam00] Tomonori Yamane. The Design and Implementation of the 2K Resource Management Service. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, February 2000.

[Yel95] Frank Yellin. Low-level Security in Java. In *Fourth International World Wide Web Conference*, Boston, December 1995.

Vita

Fabio Kon was born in São Paulo, Brazil on October 15, 1969. He received BS and MS degrees in Computer Science from the University of São Paulo and a BA degree in Music (Percussion) from the São Paulo State University. His research interests include Distributed Operating Systems, Reflective Middleware, Mobile Agents, Multimedia, and Computer Music. He has published over 30 papers on these topics.

Fabio has developed SODA, a consistent distributed file system based on leases; MAXAnnealing, a tool for algorithmic musical composition; a scalable multimedia distribution system; *dynamicTAO*, a dynamically configurable reflective ORB, and a framework for automatic configuration of component-based distributed systems.

During the course of his PhD, Fabio was seen many times playing the vibraphone with the Parkland Jazz Big Band and holding a pandeiro in the Corcovado choro ensemble.

After a period of post-doctoral research at the University of Illinois, Fabio is moving back to São Paulo, where he will continue his research in the Department of Computer Science at the University of São Paulo. His home pages are `http://choices.cs.uiuc.edu/f-kon` and `http://www.ime.usp.br/~kon`.

Index