USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

The following paper was originally published in the

5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99)

San Diego, California, USA, May 3–7, 1999

Supporting Automatic Configuration of Component-Based Distributed Systems

Fabio Kon and Roy H. Campbell
University of Illinois at Urbana-Champaign

For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738
Email: office@usenix.org WWW: http://www.usenix.org

Supporting Automatic Configuration of Component-Based Distributed Systems*

Fabio Kon† Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 West Springfield Avenue, Urbana, IL 61801-2987 USA
{f-kon,roy}@cs.uiuc.edu
http://choices.cs.uiuc.edu

Abstract

Recent developments in Component technology enable the construction of complex software systems by assembling together 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.

Component-based software systems must maintain explicit representations of inter-component dependence and component requirements. This provides a common ground for supporting fault-tolerance and automating dynamic configuration.

In this paper, we present a generic model for reifying dependencies in distributed component systems and discuss how it can be used to support automatic configuration. We describe our experience deploying the framework in a CORBA-compliant reflective ORB and discuss the use of this model in a new distributed operating system.

1 Introduction

Research on object-oriented technology and its intensive use by the industry has led to the development of *component-oriented programming*. Rather than being an alternative to object-orientation, component technology extends the initial concepts of objects. It stresses the desire for independent pieces of software that can be reused and combined in different ways to implement complex software systems.

Recently developed component architectures [Ham97, Den97, OMG97] 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. However, there is still very little support for managing the interactions between 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 dynamic dependencies between components are not well understood. It is very common to find cases, in both legacy and component-based systems, in which a module fails to accomplish its goal because an unspecified dependency is not properly resolved. Sometimes, the graceful failure of one module is not properly detected by other modules leading to system failure.

A similar problem can be detected in a different context. Current systems are continuously being updated and modified. For example, system administrators working on UNIX or Windows NT environments must be aware of security announcements on a daily basis and be prepared to update the operating system kernel with security patches. In addition, users demand new versions of applications such as web browsers, text editors, software development tools, and the like. Often, building and installing a

*This research is supported by a grant from the National Science Foundation, NSF 98-70736.

†Fabio Kon is supported in part by a grant from CAPES, the Brazilian Research Agency, proc.#1405/95-2.

new package requires that a series of other tools be updated.

Users of workstations and personal computers are also not free from the burden of system or account maintenance. In environments like MS-Windows, the installation of some applications is partially automated by "wizard" interfaces which directs the user through the installation process. However, it is common 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 problem behind all these difficulties is the lack of a model for representing the dependencies among system and application components and mechanisms for managing these dependencies.

We argue that operating system and middleware environments must provide support for representing the dependencies among software components in an explicit way. 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 better support 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 and with minimal impact in performance.

Our research builds on previous and ongoing work on software architecture [SG96], dynamic reconfiguration of distributed systems [HWP93, Hof94, SW98], and quality of service specification [FK98, LBS+98]. Our long-term goal is to develop a generic model for automatic configuration that can be applied to modern component architectures.

1.1 Paper Contents

The initial objective of our research is the support for representing dependencies among software components in an explicit way. With that support, we develop mechanisms that utilize this representation to perform automatic (re)configuration of software components in dynamic environments.

This paper describes our model for representing component prerequisites (section 2.1) and runtime inter-component dependence (section 2.2). Although we describe the implementation of a framework for reifying inter-component dependence, the details about the implementation of prerequisites are out of the scope of this paper and will be addressed in a future document.

Section 3 presents two application scenarios: section 3.1 describes our experience using the framework to support on-the-fly reconfiguration of *dynamicTAO*, a reflective CORBA-compliant ORB and section 3.2 discusses the use of our model in the *2K* distributed operating system.

After discussing related work in section 4, we describe our plans for the future in section 5 and present our conclusions in section 6.

2 Inter-Component Dependence

To address the problems described in the previous section, a configuration system must explore two distinct kinds of dependencies:

1. Requirements for loading an inert component into the runtime system (called *prerequisites*).
2. *Dynamic dependencies* among loaded components in a running system.

As long as the system knows exactly what the requirements are for installing and running a software component, the installation and configuration of new components can be automated. As a byproduct of this knowledge, component performance can be improved by analyzing the dynamic state of system resources, analyzing the characteristics of each component, and by configuring them in the most efficient way.

Also, if the system knows what the dynamic dependencies among running components are, it can (1) better handle exceptional behavior that could potentially trouble component operation, and (2) support dynamic reconfigurations of large systems by replacing individual components on-the-fly.

Prerequisites and runtime dependencies are two distinct forms of the same entity. Prerequisites usually are expressed as dependencies on "persistent" hardware and software components while runtime dependencies refer to dynamic, possibly volatile, components. In particular, if one freezes a component's state (including its runtime dependencies) and stops it, one could later resume its execution by using the frozen runtime dependencies as the prerequisites for reloading the component. However, in order to make the model as clear as possible, we are going to treat prerequisites and runtime dependencies as separate entities. Prerequisites usually refer to hardware resources, QoS requirements, and software services. Runtime dependencies refer to loaded software components. Thus, we believe that the separation of concepts is justifiable. In the future, after the basic problems are solved, we may consider to unify these concepts in order to build a simpler and more generic model.

2.1 Prerequisites

The prerequisites for a particular inert component 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 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., components) it requires.

The first two items may be used by a distributed Resource Management Service 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 is the one which determines which auxiliary components must be loaded and in which kind of software environment they will execute.

The first two items can be expressed by QoS specification languages [FK98, LBS+98]. The third item is equivalent to the *requires* clause in module interconnection languages like, for instance, the one used in Polylith [Pur94]. We are in the process of analyzing existing specification languages to study which ones would best fit our needs. The language must allow processing specifications at execution time with little overhead. We will deploy initial prototypes in *2K*, a new CORBA-based distributed operating system [KSC+98, CNM98] currently under development. The main purpose of this paper, however, is to describe the design and implementation of the infrastructure for representing runtime dependencies presented next.

2.2 Dynamic Dependencies

In our model, each component is managed by a *component configurator* which is responsible for storing the dependencies between a specific component and other system and application components.

Depending on the way it is implemented, a component configurator may be able to refer to components running on a single address space, on different address spaces and processes, or even running on different machines in a distributed system. Figure 1 depicts the dependencies that a component configurator reifies.

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, each 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, and a local file

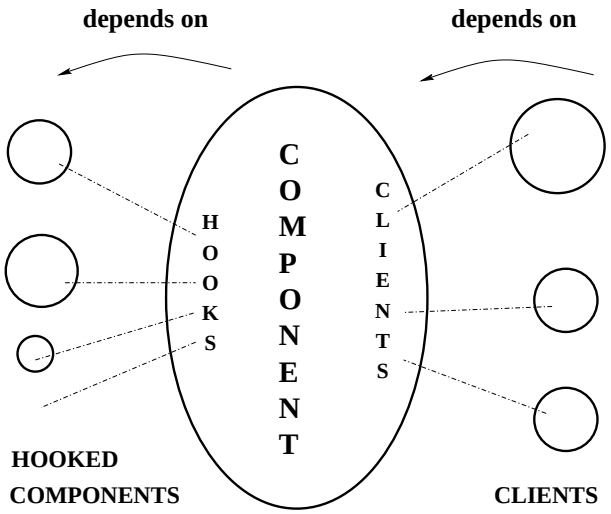

Figure 1: Reification of component dependence.

service. Its component configurator should maintain a hook for each of these services. When the browser is loaded, the 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.

2.2.1 The ComponentConfigurator class

The reification of runtime dependencies is accomplished by assigning one *ComponentConfigurator* object to each component. A simplified declaration of the *ComponentConfigurator* class in pseudo-C++ follows. Figure 2 shows a schematic representation of some of its method calls.

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

 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 eventOnHookedComponent
      (ComponentConfigurator *hookedComponent,
       Event e);
 int eventOnClient
      (ComponentConfigurator *client,
       Event e);

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

 Object *implementation ();
}
```

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

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.

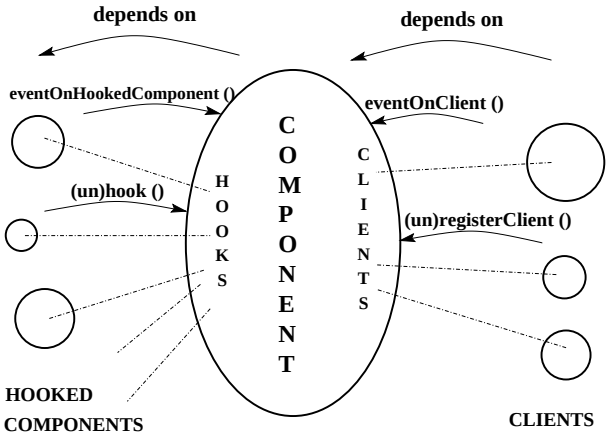

Figure 2: Methods for specifying dependencies and sending events.

eventOnHookedComponent() announces that a component which is attached to this component has generated 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 in-

ternal reconfiguration of a hooked component, or the replacement of the implementation of a hooked component.

eventOnClient() is similar to the previous method but it announces that a client has generated 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 *DELETED*, *FAILED*, *RECONFIGURED*, *REPLACED*, and *MIGRATED*. 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.

2.2.2 Towards Automatic Configuration

As discussed above, reified inter-component dependence can help the automation of configuration processes. By scanning the list of prerequisites, the operating system or middleware can be certain that all hardware and software requirements for the execution of a particular component are met before it is initiated. This can avoid a large number of problems that are common in existing systems where the lack of a particular component or resource is only detected after the application is running.

The dynamic dependence information, in its turn, enables the reconfiguration of components that are already running. In addition, it provides important information for implementing fault-tolerance and smooth exception handling in an environment of centralized or distributed components.

As an example, consider the deletion of a component containing our *ComponentConfigurator* class. Different policies for dealing with component deletion can be adopted. In general, when a component C is destroyed, an announcement must be made to components that depend on C and to components on which C depends. The following piece of pseudo-C++ code illustrates this process with a conservative implementation of the *ComponentConfigurator* destructor.

```
ComponentConfigurator::~ComponentConfigurator()
{
 for (c in hookedComponents) {
  c.configurator->unregisterClient (this);
 }

 for (c in clients) {
  c.configurator->
     eventOnHookedComponent (this,
                             DELETED);
 }

 // delete list of hooks and hookedComponents
 // delete list of clients
 // release resources
 // delete component implementation
}// ~ComponentConfigurator ()
```

Implementations of this destructor can be specialized to adjust its behavior to different component types and to meet special requirements. Also, different component types must implement methods such as *eventOnHookedComponent()* in proper ways to take care of the different kinds of dependencies. In an extreme case, deleting a component will cause all components that depend on it to be deleted. In the other extreme case, these other components will only be notified and nothing else will change. In most of the cases, we expect that these components will try to reconfigure themselves in order to deal with the loss of one of its dependencies.

The problem with this implementation is that the complete destruction of the component only takes

place if all the method calls to hooked components and clients return. If any of these calls block, the component is not deleted. This problem is particularly important if some of the clients decide to initiate their own destruction as a result of the call to *eventOnHookedComponent()* and a long chain of calls is established.

A naïve solution to this problem could be to execute the method calls asynchronously, for example, by creating new threads to perform the calls. This solution would incur in the additional cost of creating new threads and could lead to dangerous situations as a C++ component could try to call a method on another component after the latter is destroyed.

Thus, it seems that we are trapped between a safe, conservative solution that might block indefinitely and a liberal but unsafe solution that may crash the whole system by executing invalid code. We have been studying this problem and, in [KC98], we discuss solutions that lie somewhere between these two extremes. They are as safe as the conservative one but are less subject to blocking.

2.2.3 Managing Dependencies

The use of our model in a language like C++ requires strict collaboration from the component developer to conform to proposed guidelines. It is also important that all the communication between components be done through controlled interfaces. In order to avoid a proliferation of programming errors related to dependence reification, it would be necessary to develop special languages, compilers, and runtime systems to guarantee the safety of component execution and reconfiguration.

A cleaner solution would be to use existing reflective languages and environments. Iguana [GC96] and OpenC++ [Chi95], for example, are extensions to C++ that reify several features of this language, allowing dynamic modification of their implementations. In these languages, it would be possible to instrument method invocation to take care of dependence maintenance.

However, a major goal of our research is not to limit the implementation to a particular programming language and only use widely accepted standards. We could also tie together the mechanisms for communication and dependence representation using, for example, abstract connectors [SDZ96]. But this could limit the expressiveness of the model. Our objective is to develop a generic methodology that could be utilized in a large number of heterogeneous environments. These requirements can only be met by using a standard architecture like CORBA.

2.3 CORBA ComponentConfigurator

CORBA permits the integration of components written in different programming languages on heterogeneous environments. In addition, CORBA's (remote) method invocation mechanism can be decoupled from the base language method call. Thus, it is possible to guarantee that bad CORBA references are not translated into bad base language references (like dangling C++ pointers for example). Instead, exceptions are neatly handled by the runtime and the application is informed of its occurrence.

In the CORBA implementation of our model, a *DependencySpecification* stores a CORBA Interoperable Object Reference (IOR) so that the *ComponentConfigurator* is able to reify dependencies among distributed components. Prerequisites for software components can be specified either in terms of persistent IORs [Hen98] or in terms of service type and attributes. In the former case, an implementation repository can be used to dynamically create a new CORBA object if one is not available. In the latter case, the CORBA Trading Object Service [OMG98] can be used to locate an instance of the server component that meets the requirements specified by the given attributes.

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. In this case, it is the responsibility of the client component to locate a new server component and update its *ComponentConfigurator*.

As future work, we intend to perform experiments with the different ways of using the CORBA *ComponentConfigurator* to manage distributed applications. In particular, 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 centralized node on the network while the component implementations are distributed. We will investigate the benefits of the different approaches.

2.4 Implementation Status

We have implemented prototypes of the *ComponentConfigurator* for centralized applications in C++ and Java. The C++ implementation was deployed in the *dynamicTAO* ORB as described in section 3.1. We have recently completed an implementation of distributed *ComponentConfigurator*s based on CORBA.

We plan to extend the Java implementation to support Java Bean components and distributed object communication with Java RMI. We will, then, work on the interoperability among different implementations of the model in different component architectures.

3 Application Scenarios

This section describes the deployment of the *ComponentConfigurator* framework in *dynamicTAO*, a reflective Object Request Broker. It illustrates how our model can be used to represent and manipulate the internal structure of a legacy system, enabling dynamic reconfiguration. We, then, discuss how this framework will be used to support architectural awareness in the *2K* distributed operating system.

3.1 *dynamicTAO*

One of the major constituent elements of *2K*, a distributed operating system our group is developing [KSC+98, CNM98], is a reflective middleware layer based on CORBA. After carefully studying existing Object Request Brokers, we came to the conclusion that the TAO ORB [SC99] would be the best starting point for developing our infrastructure. TAO is a portable, flexible, extensible, and configurable ORB based on object-oriented design patterns. It uses the *Strategy* design pattern [GHJV95] to separate different aspects of the ORB internal engine. A configuration file is used to specify the strategies the ORB uses to implement aspects like concurrency, request demultiplexing, scheduling, and connection management. At ORB startup time, the configuration file is parsed and the selected strategies are loaded.

TAO is primarily targeted for static hard real-time applications such as Avionics systems [HLS97]. Thus, it assumes that, once the ORB is initially configured, its strategies will remain in place until it completes its execution. There is very little support for on-the-fly reconfiguration.

The *2K* project seeks to build a flexible infrastructure to support adaptive applications running on dynamic environments. On-the-fly adaptation is extremely important for a wide range of applications including the ones dealing with multimedia, mobile computers, and dynamically changing environments.

The design of *2K* depends on *dynamicTAO*, 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 servants running on top of the ORB and even for reconfiguring non-CORBA applications.

3.1.1 Problems Encountered

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

Consider the case in which *dynamicTAO* receives a request for replacing one of its strategies (S_{old}) by a new strategy (S_{new}). The first problem is that, since TAO strategies are implemented as C++ objects that communicate through method invocations, before unloading S_{old}, the system must be sure that no one is running S_{old} code and that no one is ex-

pecting 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}.

These problems can be addressed with the help of the *ComponentConfigurator* which is used to reify the dependencies among strategies, instances of *dynamicTAO*, and servants.

3.1.2 DomainConfigurator and TAOConfigurator

Each process running the *dynamicTAO* ORB contains a *ComponentConfigurator* 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 subclass of *ComponentConfigurator* called *TAOConfigurator*.

TAOConfigurator contains hooks to which *dynamicTAO* strategies are attached. A *NetworkBroker* implements a simple TCP-based protocol that allows remote entities to connect to the process to inspect and change the configuration of *dynamicTAO* by loading new strategies and attaching them to specific hooks. Local servants and remote CORBA clients can also access the *Configurator* objects through a programmatic CORBA interface. Figure 3 illustrates this mechanism when a single instance of the ORB is present.

If necessary, individual strategies may have their own customized subclass of *ComponentConfigurator* to manage their dependencies upon ORB instances and other strategies. These subclasses may also store references to client connections that depend on them. With this information, it is possible to decide when a strategy can be safely unloaded.

Consider, for example, the three concurrency strategies supported by *dynamicTAO*: Single-Threaded Reactive [Sch94], 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.

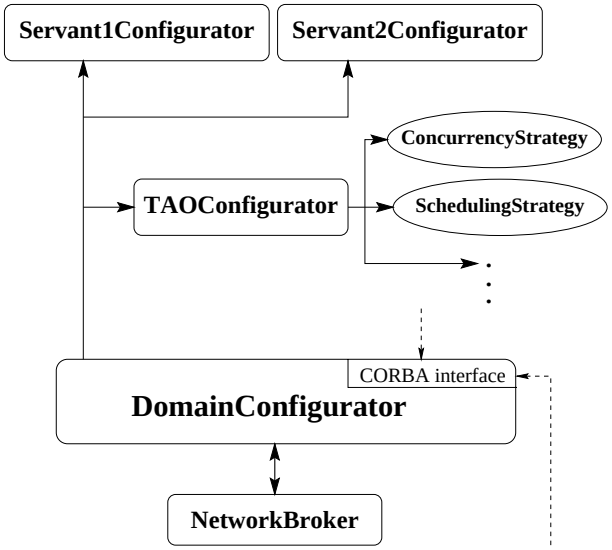

Figure 3: Remote Configuration of *dynamicTAO* strategies.

However, if one switches from the Thread-Pool strategy to another one, special care must be taken. 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 signalizes 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 enqueued in the strategy waiting for a thread to execute them. The so-

lution 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 enqueued connections.

Our group is currently expanding the set of *dynamicTAO* strategies that can be replaced on-the-fly. The *TAOConfigurator* will have hooks for holding strategies for connection management, concurrency, (de)marshalling, request demultiplexing, method dispatching, scheduling, and security. An explicit knowledge of the dependencies among the ORB components is essential for implementing dynamic reconfiguration safely.

3.2 Architectural Awareness in *2K*

In contrast to existing systems where a large number of non-utilized modules are carried along with the basic system installation, the *2K* operating system is based upon a "what you need is what you get" (WYNIWYG) model. The system configures itself automatically and loads the minimum set of components required for executing user applications in the most efficient way. Components are downloaded from the network and only a small subset of system services are needed to bootstrap a node.

This is achieved by reifying the hardware and software prerequisites for each loadable component. As mentioned in section 2.1, the operating system can use this information to make sure that all the basic services that a component requires are available before the component is loaded. In addition, a distributed resource manager uses the specifications of the component hardware requirements to decide in which machine the component should be loaded and perform admission control and resource reservation. That way, one will not face a situation in which a component fails to execute its task with the desired quality of service because an unspecified dependency was not resolved.

As a component is loaded into the system, its prerequisites are scanned and all the specified services are made available. During this process, the system can incrementally build a dynamic graph of dependencies using the *ComponentConfigurator* framework.

The design of *2K* supports fault-tolerant, self-adapting systems by monitoring the environment and maintaining a representation of the dynamic structure of its services and applications. The CORBA implementation of the *ComponentConfigurator* framework reifies the distributed system dynamic structure.

When a *2K* component fails, the system inspects its dependencies and informs the proper components about the failure. The system may alternatively recover from a failure by replacing the faulty component with a new one. The same mechanism can be used for adapting the system and its components to changing parameters such as network bandwidth, CPU load, resource availability, user access patterns, etc.

4 Related Work

The idea of using prerequisites to represent the dependencies among operating system objects was introduced in the SOS operating system [SGH+89] 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. Not much experimentation was carried out [SGM89, Sha98]. SOS does not include a model for dynamic management of inter-component dependence.

Previous research in microkernels and customizable operating systems – such as Mach [Lop91], SPIN [BSP+95], Exokernel [KEG+97], and μChoices [LTC96] – developed low-level techniques for dynamic loading new modules to the operating system both in kernel and user space. Nevertheless, a high-level model for operating system reconfiguration is still inexistent. These previous works have not addressed a number of problems related to fault-tolerance and dynamic reconfiguration. Using the *ComponentConfigurator* framework, our research investigate answers to the following questions.

- What are the consequences of reconfiguring the operating system?
- When a system module is replaced, which other

modules are affected?

- How must those other modules react?
- When (re)configuring the system, which components must be loaded to meet the service demand and the required quality of service?
- If a system component fails, how can the system detect it and recover gracefully?

We are currently investigating languages for prerequisite specification. They must be able to represent hardware and quality of service requirements as well as dependencies on other software components. Thus, we believe that an ideal language for prerequisite specification will build on previous work on Architecture Description Languages [Cle96] and QoS Specification Languages [FK98, LBS+98].

Connector-based systems like UniCon [SDZ96] and 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. Our framework is independent of the paradigm for inter-component communication; it can be used in conjunction with connectors, buses, local method invocation, CORBA, Java RMI, etc.

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 (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 informed of substantial changes in the state and configuration of the services that may affect its performance.

Differently from previous work in this area, our model does not dictate a particular communication paradigm like connectors or buses. As shown in section 3.1, the model was applied to a legacy system without requiring any modification to its functional implementation or to its inter-component communication mechanisms.

We are particularly interested in investigating the possibilities of applying results from previous and ongoing work in dynamic reconfiguration [HWP93, SW98, BBB+98] to standard architectures such as CORBA and Java Beans.

5 Ongoing and Future Work

The current implementation of the framework in C++ is being used in *dynamicTAO* as its dynamic reconfigurability is enhanced. In addition, the Java implementation is being used by researchers at the University of São Paulo to prototype a domain decomposition manager. This manager has two demonstration applications: a Distributed Information System for Mobile Agents [SGE98] and the parallelization of an Atmospheric Modeling System [Bar98].

Work on implementations of the framework in Java RMI is underway. As discussed in 3.2, the CORBA implementation of the *ComponentConfigurator* will be used in the *2K* operating system to support runtime architectural awareness as the basis for implementing fault-tolerant reconfigurable systems. The prerequisites model will be used for QoS-aware resource management. This will provide components with all the hardware and software resources they need to execute with the desired quality of service.

6 Conclusions

We have presented a model for runtime architectural awareness in centralized and distributed component-based systems. We believe that the reification of inter-component dependence and component prerequisites is fundamental for systems supporting fault-tolerant, reconfigurable components.

The model has been prototyped in Java, C++, and CORBA. The C++ framework was successfully deployed in *dynamicTAO*, a legacy system, which was

made aware of its own internal structure.

Future work in the *2K* operating system will demonstrate how the model behaves in a complex, distributed CORBA-based system.

Acknowledgments

We gratefully acknowledge the help provided by Manuel Román on the implementation of dynamicTAO. We thank Dilma Menezes, Francisco Ballesteros, and the members of the *2K* team for their feedback on the ideas presented in this paper. Finally, we thank the anonymous reviewers who contributed with valuable comments.

Availability

The framework source code in C++ and Java is available at `http://choices.cs.uiuc.edu/2k/ComponentConfigurator` .

The source code and detailed documentation for *dynamicTAO* can be found at `http://choices.cs.uiuc.edu/2k/dynamicTAO` .

References

[Bar98] Saulo Barros. The Regional Atmospheric Modeling System (RAMS) Project. `http://www.ime.usp.br/~rams/`, 1998.

[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)*, The Lake District, UK, September 1998.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chabers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th Symposium on Operating System Principles*, December 1995.

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

[Cle96] Paul C. Clements. A survey of architecture description languages. In *Proceedings of The Eighth International Workshop on Software Specification and Design*, Paderborn, Germany, March 1996.

[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.

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

[FK98] Svend Frølund and Jari Koistinen. Quality of Service Specification in Distributed Object Systems Design. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technology and Systems (COOTS)*, Santa Fe, New Mexico, April 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.

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

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

[Hen98] Michi Henning. Binding, Migration, and Scalability in CORBA. *Communications of the ACM*, 41(10), October 1998.

[HLS97] Tim 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.

[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.

[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.

[KC98] Fabio Kon and Roy H. Campbell. On the Role of Inter-Component Dependence in Supporting Automatic Reconfiguration. Technical Report UIUCDCS-R-98-2080, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1998.

[KEG+97] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor Briceño, Russell Hunt, David Mazières, Tom Pinckney, Robert Grimm, John Janotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, Saint Malo, France, October 1997. ACM.

[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 *Proceedings of the 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. To appear in Lecture Notes in Computer Science, Springer-Verlag.

[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 *Proceedings Fifth International Workshop on Object-Orientation in Operating Systems*, pages 62–66, Seattle, Washington USA, October 1996.

[OMG97] OMG. *CORBA Component Model RFP*. Object Management Group, Framingham, MA, 1997. OMG Document 97-05-22.

[OMG98] OMG. *CORBAservices: Common Object Services Specification*. Object Management Group, Framingham, MA, 1998. OMG Document 98-07-05.

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

[SC99] Douglas C. Schmidt and Chris Cleeland. Applying Patterns to Develop Extensible ORB Middleware. *IEEE Communications Magazine*, 1999. (to appear), available at `http://www.cs.wustl.edu/~schmidt/ACE-papers.html`.

[Sch94] Douglas C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In *Proceedings of the 1st Pattern Languages of Programs Conference*, August 1994.

[SDZ96] Mary Shaw, R. DeLine, and G. Zelesnik. Abstractions and implementations for architectural connections. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, 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. Available at `http://www.ime.usp.br/~dilma/papers/semish98.ps`.

[SGH+89] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin,

and Cline 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.

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