Dependence Management in Component-Based Distributed Systems*

Fabio Kon†  Roy H. Campbell‡

Department of Computer Science
University of Illinois at Urbana-Champaign

{f-kon, roy}@cs.uiuc.edu
http://choices.cs.uiuc.edu/2K

This paper will appear in a future issue of IEEE Concurrency.

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 article, 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.

Keywords: components, dynamic configuration, distributed systems, operating systems, middleware, CORBA.

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 such as Enterprise JavaBeans, ActiveX Controls, and the CORBA Component Model 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 dependencies 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 total system failure.

A similar problem can be detected in a different context. Current systems are continuously being updated and modified. 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 ker-
nel 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 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 relying 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, dynamic configuration of distributed systems, and quality of service specification. 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 tie each component to other application, middleware, and system components. Our long-term goal is to develop an integrated model for automatic configuration that can be applied to modern component architectures.

2 Inter-Component Dependence

To address the problems described above, 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 reconfiguration 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 unifying 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 recent QoS specification languages. The third item is equivalent to the require clause in architecture description languages like Darwin and module interconnection languages like the one used in POLYLITH (see “Related Work” sidebar).

We have recently completed a prototype implementation of prerequisite-based automatic configuration [1] in the 2K distributed operating system (visit http://choices.cs.uiuc.edu/2K). The prototype is based on a skeleton to which different kinds of prerequisite parsers and prerequisite resolvers can be plugged, allowing for different specification languages and different prerequisite resolution policies. The prototype uses a Simple Prerequisite Description Format (SPDF) that supports the three above mentioned kinds of prerequisites. The prerequisite resolver fetches component implementations from remote CORBA implementation repositories and caches them locally. We are currently extending the prototype to specify dependencies in terms of the standard CORBA trading format and to locate a close to optimal machine for executing each component. Figure 1 shows a typical SPDF description.

Proper specification and handling of prerequisites is a field that deserves close attention from the software community as it is fundamental for achieving a good level of reliability and quality of service in component-based systems. The main focus of this article, however, is to describe the design and implementation of the infrastructure for representing runtime dependencies, which is presented next.

2.2 Dynamic Dependencies

In our model, each component is managed by a component configurator which is responsible for storing the runtime 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 2 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, 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

Figure 1: A Simple Prerequisite Description

:hardware requirements
machine_type  SPARC	native_os  Solaris
min_ram  5MB
optimal_ram  40MB
cpu_speed  >300MHz
cpu_share  10%

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

COMPONENT

HOOKED
COMPONENTS

depends on

CLIENTS

Figure 2: Reification of component dependence.

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. Figure 3 contains a simplified declaration of the ComponentConfigurator abstract class in pseudo-C++. Figure 4 shows a schematic representation of some of its method calls.

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.

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

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 ();
}
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 DependencySpecifications. A DependencySpecification is a structure defined as

```c
struct DependencySpecification {
    const char *hookName;
    ComponentConfigurer *component;
};
```

listClients() returns a pointer to a list of DependencySpecifications corresponding to the components that depend on this component (its clients) and the name of the hooks (in the client’s ComponentConfigurer) 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 Reconfiguration

As discussed above, reified inter-component dependencies 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. 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, the application developer could implement a WebBrowserConfigurer by using inheritance from the ComponentConfigurer and customizing it to deal with the dynamic replacement of the system’s JVM. As shown in Figure 5, the eventOnHookedComponent method can be overridden to catch REPLACED events coming from the JVM ComponentConfigurer.

```c
int WebBrowserConfigurer::eventOnHookedComponent (ComponentConfigurer *cc, Event e)
{
    if (cc == JVMConfigurer)
    {
        if (e == REPLACED)
        {
            try {
                FrozenObjs fo = currentJVM->freezeAllObjs();
                currentJVM = JVMConfigurer->implementation ();
                currentJVM->meltObjects (fo);
            }
            catch (Exception exp)
            {
                throw ReconfigurationFailed (exp);
            }
        }
    }
    else ...
}
```

Figure 5: Customization of the eventOnHookedComponent method

When the implementation of the JVM is updated, the JVMConfigurer sends a REPLACED event to its clients. When the WebBrowserConfigurer receives this event, it freezes all the objects in the current JVM, updates the current JVM with the new JVM implementation, and melts the objects in the new JVM.

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 underlying reconfiguration engine by requiring every component to implement a pair of operations export_state() and import_state(). All the components of a certain type should then agree a priori on a common external representation of the internal state of the components of that type. The underlying engine would simply transfer 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 four mechanisms: (1) using the ComponentCon-
figurator to notify all the components that have a reference to the old one; (2) using the ComponentConfigurator as an indirect on calls to replaceable components; (3) leaving a forwarding pointer in place of the old component; and (4) making every access to the old component throw an exception that is captured by the client which then gets a reference to the new component by contacting a third party such as a Naming Service. In certain cases, a fifth option can be adopted: 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. Different combinations of these mechanisms can be used in different parts of a single system.

Dynamic dependencies also provide 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. Figure 6 illustrates this process with a conservative implementation of the ComponentConfigurator destructor.

```cpp
ComponentConfigurator::ComponentConfigurator()
{
  for (c in hookedComponents)
    c.configurator->unregisterClient (this);
  for (c in clients)
    c.configurator->eventOnHookedComponent (this,
        ComponentConfigurator::DELETED);

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

Figure 6: A ComponentConfigurator destructor

Implementations of this destructor can be specialized to adjust its behavior to different component types and to meet special application requirements. 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 even-

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 [3], for example, is an extension of C++ that refines 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 stan-
2.3 CORBA ComponentConfigurer

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 ComponentConfigurer is able to reify dependencies among distributed components. Prerequisites for software components can be specified either in terms of persistent IORs [4] or in terms of a pair <ServiceTypeName, Constraints>. 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 [5] can be used to locate an instance of the server component that meets the requirements specified by the given constraints.

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 ComponentConfigurer.

As future work, we intend to perform experiments with the different ways of using the CORBA ComponentConfigurer 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 central node on the network while the component implementations are distributed. In addition, one may adopt a combination of two of these schemes. For example, each component could have an co-located instance of the ComponentConfigurer as well as another one in a central node on the network. In that case, the centralized dependence graph would allow the execution of algorithms dealing with the dependencies of the distributed system efficiently within a single process. The co-located instance would provide fast interaction between each component and its configurator. Finally, the redundant information would aid fault-tolerance since the information lost with the failure of the central node could be reconstructed by contacting the distributed instances.

2.4 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 ComponentConfigurer 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 the standard Concurrency Control Service [5]. Ideally, a configuration system should provide support for grouping operations into atomic transactions satisfying the ACID properties, i.e., atomicity, consistency, isolation, and durability. This can be achieved with the help of the CORBA Object Transaction Service [5].

2.5 Security

In networked environments, it becomes necessary to secure the configuration system from unauthorized access. A hostile agent that obtains access to the
ComponentConfigurators 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 ComponentConfigurators like, for example, the ones containing reconfiguration events.

To support security in environments such as Java, using RMI for communication, 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 [5], it is possible to add message-level interceptors into the ORB so that every data exchanged between CORBA objects is properly encrypted. In addition, request-level interceptors can control the access to each individual operation on the ComponentConfigurators based on who is issuing the call, based on capabilities, or based on any other customized mechanism defined by the programmer.

2.6 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 later for dynamic adaptation in face of changes in resource availability. The resource requirements expressed in the prerequisites should typically specify ranges of acceptable service. A video-on-demand application, for example, could specify that it requires a network bandwidth of 500Kbps on average but that it may utilize peak rates of up to 1Mbps. Finally, it could add that even though 500Kbps 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, dynamically adapting to these changes. Thus, prerequisites should be available to the system at runtime so that it can reorganize its allocation of resources in order to better fulfill the requirements of all the applications sharing the system.

3 Application Scenarios

We have investigated the deployment of the ComponentConfigurator framework in both centralized and distributed applications. On one hand, dynamicTAO, a reflective Object Request Broker, illustrates how our model can be used to represent and manipulate the internal structure of a centralized legacy system, enabling dynamic reconfiguration. On another hand, the 2K distributed operating system shows how our model can be used since the early phases of system design to achieve maximum levels of reliability and dynamic flexibility.

3.1 dynamicTAO

One of the major constituent elements of 2K is a reflective middleware layer [6] based on CORBA. After carefully studying existing Object Request Brokers, we came to the conclusion that the TAO ORB [7] would be the best starting point for developing our infrastructure. TAO is a portable, flexible, extensible, and configurable ORB based on object-oriented design patterns. It is written in C++ and uses the Strategy design pattern [8] to separate different aspects of the ORB internal engine. A configuration file is used to specify the strategies the ORB uses to implement aspects like concurrency, request demultiplexing, scheduling, and connection management. At ORB startup time, the configuration file is parsed and the selected strategies are loaded.

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

The 2K project 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 [9], our 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. It 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 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 transferred 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 the 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 7 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, Thread-Per-Connection, and Thread-Pool. If the user switches from the Reactive or Thread-Per-Connection strategies to any other concurrency strategy, nothing special needs to be done. dynamicTAO may simply load the new strategy, update the proper TAOConfigurator hook, unload the old strategy, and continue. Old client connections will complete with the concurrency policy dictated by the old strategy. New connections will utilize the new policy.

However, if one switches from the Thread-Pool strategy to another one, 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 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 enqueued in the strategy waiting for a thread to execute them. The solution is to use the *Memento* pattern [8] 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 *dynamic TAO* strategies that can be replaced on-the-fly. At the present moment, *TAOConfigurator* hooks hold strategies for concurrency, security, and performance monitoring. We plan to add hooks for connection management, (de)marshalling, request multiplexing, method dispatching, transport protocols, and scheduling. With *dynamic TAO* we learned that 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" (WYNWYG) model [1]. 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 earlier, 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 incrementally builds 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.

Quality of Service is supported by a Local Resource Manager that resides in each machine and monitors resource utilization. Changing parameters such as network bandwidth, CPU load, memory availability, and user access patterns may trigger adaptations and reallocation of resources based on the component prerequisites, which are accessible at runtime.

### 4 Implementation Status and Future Work

We implemented prototypes of the *ComponentConfigurator* for single-process applications in C++ and Java. The C++ implementation was deployed in the *dynamic TAO* ORB as described earlier. The Java implementation is used by researchers at the University of São Paulo to prototype a domain decomposition manager with applications in a Distributed Information System for Mobile Agents and in the parallelization of an Atmospheric Modeling System.

More recently, we completed an implementation of distributed `ComponentConfigurators` based on CORBA and are using them in the construction of 2K distributed services such as the Persistent Object Service and the Automatic Configuration Service.

Complete documentation and source code for the framework in C++, Java, and CORBA/C++ is available at our web site at [http://choices.cs.uiuc.edu/2k/DynamicConfiguration](http://choices.cs.uiuc.edu/2k/DynamicConfiguration).
5 Conclusions

We believe that the reification of inter-component dependence and component prerequisites is fundamental for systems supporting reliable, reconfigurable components. Our initial experience with the framework has proved to be very fruitful. We successfully deployed it in a legacy system, which was made aware of its own internal dependencies, allowing the easy addition of dynamic reconfiguration. Future work in the 2K operating system will demonstrate how the model behaves in a complex, distributed CORBA-based system.

Dependence management is probably the most crucial problem to be resolved before operating systems 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.

Acknowledgments

Fabio Kon is supported in part by CAPES, Brazil, proc. 1405/95-2. This research is supported by grants from the National Science Foundation, NSF 98-70736 and 99-70139. We gratefully acknowledge the help provided by Manuel Román on the implementation of dynamic TAO. We thank Dilma Menezes, Francisco Ballesteros, and the members of the 2K team for their feedback on the ideas presented here. Finally, we thank the anonymous reviewers and Murthy Devarakonda who contributed with valuable comments for improving this article.

References


Fabio Kon is a PhD candidate in Computer Science working on the overall design and implementation of the 2K distributed operating system. He has received BS and MS degrees in Computer Science from the University of São Paulo and a BA degree in Music from the São Paulo State University. His research interests include Distributed Operating Systems, Multimedia, and Computer Music. He has developed SODA, a consistent distributed file system based on leases; MAXAnnealing, a tool for algorithmic musical composition; a scalable multimedia distribution system; and dynamic TAO, a dynamically configurable reflective ORB. Contact him at Digital Computer Lab, 1304 W. Springfield Ave., Urbana, IL, 61801, USA; f-kon@cs.uiuc.edu.

Roy H. Campbell is a professor of Computer Science at the University of Illinois at Urbana-Champaign. His research interests include Operating Systems, Distributed Multimedia, Network Security, and Ubiquitous Computing. He is the principal
designer of the Choices, $\mu$Choices, and $2K$ object-oriented operating systems. His recent research accomplishments include VDP, an adaptive continuous media transport protocol; a robust video compression and packetization scheme for unreliable networks; and dynamic security policy systems for distributed objects, mobile computers, and active networks. He has a BSc in Mathematics from the University of Sussex, and an MSc and a PhD in Computing from the University of Newcastle upon Tyne. Contact him at Digital Computer Lab, 1304 W. Springfield Ave., Urbana, IL, 61801, USA; roy@cs.uiuc.edu.

**SIDEBAR: Related Work**

The idea of using prerequisites to represent the dependencies among operating system objects was introduced in the SOS operating system [1] 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. SOS does not include a model for dynamic management of inter-component dependence.

Previous research in microkernels and customizable operating systems – such as SPIN, Exokernel, and $\mu$Choices [2] – 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 non-existent. 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 formats 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 both on Architecture Description Languages [3, 4] and QoS Specifications [5].

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 (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 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 our discussion about dynamic TAO, the model was applied to a legacy system without requiring any modification to its functional implementation or to its inter-component communication mechanisms.

The Darwin Architectural Description Language was used in environments like Regis [8] and CORBA [9] to specify the overall structure 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. They do not represent dependencies of application components towards system components, other applications, or services available in the distributed environment. Our approach differs from theirs in the sense that, for each individual component, we specify its dependencies on all different kinds of environment components and we maintain and use this dynamic dependencies at runtime.

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. We believe that, rather than conflicting with their approach, our vision complements theirs 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 configuration may reside on the combination of our model with recent work in ADLs and dynamic reconfiguration [4, 10].

References


