


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

 The authors present a generic model for reifying dependencies in distributed component systems. They discuss how a representation model makes it possible to develop efficient, reliable, and dynamically configurable component-based systems.

Dependence Management in Component-Based Distributed Systems

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

Recent component-architecture developments—such as Enterprise JavaBeans, ActiveX Controls, and the CORBA Component Model—support the construction of sophisticated systems by assembling a collection of off-the-shelf software components with the help of visual tools or programmatic interfaces. However, very little support exists for managing the dependencies between components. Different programmers create components, often working in different groups with different methodologies. It is hard to create robust and efficient systems if they do not understand the dynamic dependencies between components. Thus, it is very common to find cases, in both legacy and component-based systems, in which a module fails to accomplish its goal because the system does not properly resolve an unspecified dependency. Sometimes, other modules do not properly detect the graceful failure of one module, which leads to a total system failure.

Dependence problems

To further illustrate the importance of understanding and managing component dependencies, consider a similar problem in a different context. Because administrators must continuously update and modify current systems, dependency conflicts may arise. For example, UNIX and Windows NT system administrators must monitor security announcements daily and be prepared to update their operating system kernels with the appropriate security patches. In addition, users demand new versions of applications such as Web browsers, text editors, software development tools, and so forth. Often, building and installing a new software package requires updates to a series of other tools as well.

Like system administrators, workstation and personal computer users are also burdened with system or account maintenance. In environments such as Microsoft Windows, wizard interfaces partially automate some application installations by directing users through the installation

Related work

Researchers at INRIA in France introduced the idea of using prerequisites to represent the dependencies between operating system objects in the SOS operating system.¹ In the SOS model, objects contain a list of prerequisites that must be satisfied before activation. Even though the idea was promising, it was not fully explored. The researchers used prerequisites only to express that an object depends on the code implementing it. SOS does not include a model for dynamic management of intercomponent dependence.

Previous research in microkernels and customizable operating systems—such as SPIN, Exokernel, and μ Choices²—developed low-level techniques for dynamically loading new modules to the operating system, both in kernel and user space. Nevertheless, a high-level model for operating system reconfiguration is still nonexistent. These previous works have not addressed a number of problems related to fault-tolerance and dynamic reconfiguration. Using the `ComponentConfigurator` framework, our research addresses 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 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 that was done both on architecture description languages^{3,4} and QoS specifications.⁵

Systems based on architectural connectors, such as UniCon³ and ArchStudio,⁶ and systems based on software buses, such as Polyolith,⁷ separate issues concerning component functional behavior from component interaction. Our model goes one step further by separating intercomponent communication from intercomponent dependence. Connectors and software buses require that applications be programmed to a particular communication paradigm. Our framework is independent of the paradigm for intercomponent communication; it can be used in conjunction with connectors, buses, local method invocations, CORBA, Java RMI, and so forth.

Communication and dependence are often intimately related. But, in many cases, the distinction between intercomponent dependence and intercomponent communication is beneficial. For example, the QoS provided by a multimedia application is greatly influenced by the mechanisms used by underlying services such as virtual memory, scheduling, and memory allocation. The interaction between the application and these services is often implicit, in other words, no direct communication (such as library or system calls) takes place. Yet, if the system infrastructure lets developers 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 might affect its performance.

process. However, it is common for users to encounter situations in which the installation cannot complete or it completes but the software package does not run properly because some unspecified requirements are not met. Or, after a new installation or update, other applications stop working. Even after users execute special uninstall procedures to remove the problem application, junk libraries and files might remain in the system because Windows applications typically cannot uninstall cleanly.

The problem behind these difficulties is the lack of a representation model for the dependencies between system and application components and mechanisms for managing these dependencies.

We argue that operating system and middleware environments must explicitly represent the dependencies between software components. Then, we can manipulate this representation to implement software components that can configure themselves and adapt to ever-changing dynamic environments.

Reification of the interactions between system and application components lets system software recognize the need for reconfiguration to better support fault tolerance, security, quality of service (QoS), and optimization. In addition, reification lets system software reconfigure without compromising system stability and reliability and with minimal impact on system performance.

Our research builds on previous and ongoing work in software architecture, dynamic configuration of distributed systems, and QoS specification. (See the sidebar “Related work” for more information.) Rather than simply look at the architectural connections between a single application’s components, we look at 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 we can apply to modern component architectures.

Intercomponent dependence

To address the problems of component dependencies, 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* between loaded components in a running system.

As long as the system knows the requirements for installing and running each software component, it can automate the installation and configuration of new components. It can improve component performance by analyzing the dynamic state of system resources, analyzing the characteristics of each component, and configuring each component in the most efficient way. Also, if a system knows the dynamic dependencies between running components, it can

Different from previous work in this area, our model does not dictate a particular communication paradigm like connectors or buses. As we show in our discussion of dynamicTAO, we applied the model to a legacy system without requiring any modification to its functional implementation or to its intercomponent communication mechanisms.

Researchers at the Imperial College in London used the Darwin architectural description language in environments such as Regis⁸ and CORBA⁹ 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 toward 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 these 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 it by reasoning about all the dependencies that might affect reliability, performance, and QoS.

The final solution to the problem of supporting reliable automatic configuration might reside on the combination of our model with recent work in ADLs and dynamic reconfiguration.^{4,10}

References

1. M. Shapiro et al., "SOS: An Object-Oriented Operating System—Assessment and Perspectives," *Computing Systems*, Vol. 2, No. 4, Dec. 1989, pp. 287–338.
2. W.S. Liao, S. Tan, and R.H. Campbell, "Fine-grained, Dynamic User Customization of Operating Systems," *Proc. Fifth Int'l Workshop Object-Oriented in Operating Systems*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1996, pp. 62–66.
3. M. Shaw, R. DeLine, and G. Zelesnik, "Abstractions and Implementations for Architectural Connections," *Proc. Third Int'l Conf. Configurable Distributed Systems (CDS'96)*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1996.
4. R. Balter et al., "Architecting and Configuring Distributed Applications with Olan," *Proc. IFIP Int'l Conf. Distributed Systems, Platforms, and Open Distributed Processing (Middleware '98)*, Springer-Verlag, 1998, pp. 241–256.
5. S. Frolund and J. Koistinen, "Quality of Service Aware Distributed Object Systems," *Proc. Fifth USENIX Conf. Object-Oriented Technology and Systems (COOTS'99)*, USENIX Assoc., 1999, pp. 69–83.
6. P. Oreizy and R.N. Taylor, "On the Role of Software Architectures in Runtime System Reconfiguration," *Proc. Fourth Int'l Conf. Configurable Distributed Systems (CDS'98)*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1998.
7. J. Purtilo, "The Polyolith Software Bus," *ACM Trans. Programming Languages and Systems*, Vol. 16, No. 1, Jan. 1994, pp. 151–174.
8. J. Magee, N. Dulay, and J. Kramer, "Regis: A Constructive Development Environment for Distributed Programs," *IEEE/IO/BCS Distributed Systems Eng. J.*, Vol. 1, No. 1, 1994, pp. 37–47.
9. J. Magee, A. Tseng, and J. Kramer, "Composing Distributed Objects in CORBA," *Proc. Third Int'l Symp. Autonomous Decentralized Systems (ISADS'97)*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1997.
10. S.K. Shrivastava and S.M. Wheeler, "Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Application," *Proc. Fourth Int'l Conf. Configurable Distributed Systems (CDS'98)*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1998.

better handle exceptional behavior that could potentially trouble component operation and 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, and runtime dependencies refer to dynamic, possibly volatile, components. In particular, if we freeze a component's state (including its runtime dependencies), we can later resume the component's execution by using the frozen runtime dependencies as the prerequisites for reloading the component. However, 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 we solve the basic problems, we might consider unifying these concepts to build a simpler and more generic model.

PREREQUISITES

An inert component's prerequisites must specify any special requirement needed to load, configure, and execute it. A prerequisite list can contain three different kinds of information:

- The nature of the hardware resources the component needs.
- The capacity of the hardware resources it needs.
- The software services (such as components) it requires.

A distributed resource management service might use the first two items to determine where, how, and when to ex-

ecute the component. QoS-aware systems can use this data to enable proper admission control, resource negotiation, and resource reservation. The last item determines which auxiliary components must load and in which kind of software environment they will execute.

Recent QoS specification languages can express the first two items. The third item is equivalent to the require clause in architecture description languages, such as Darwin, and module interconnection languages, like the one used in Polyolith (see the "Related work" sidebar).

We recently completed a prototype implementation of prerequisite-based automatic configuration¹ in the 2K distributed operating system. (Visit choices.cs.uiuc.edu/2K for more information.). We based the prototype on a skeleton that different kinds of prerequisite parsers and prerequisite resolvers can plug into, which allows for different specification languages and different prerequisite resolution poli-

```

: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)

```

Figure 1. A simple prerequisite description.

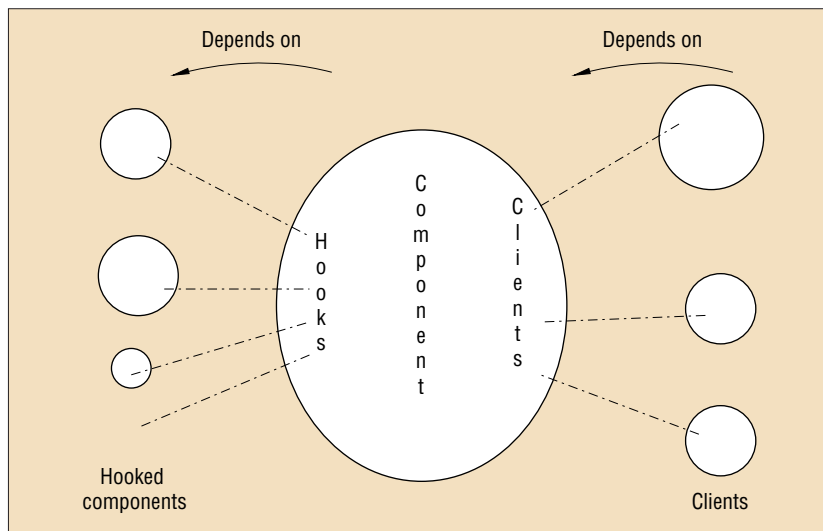


Figure 2. Reification of component dependence.

cies. The prototype uses a simple prerequisite description format (SPDF) that supports the three kinds of prerequisites mentioned earlier. 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 prerequisite specification and handling is a field that deserves close attention from the software community because it is fundamental for achieving a good level of reliability and quality of service in component-based systems. However, we focus on describing the infrastructure's design and implementation to represent runtime dependencies.

DYNAMIC DEPENDENCIES

In our model, a *component configurator* manages each component. The component configurator is responsible for storing the runtime dependencies between a specific component and other system and application components. Depending on the implementation, a component configurator might be able to refer to components running on a single address space, on different address spaces and processes, or even 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 attach. These *hooked components* are the components on which C depends. There might be other components, called *clients*, that depend on C . In general, each time 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 .
2. Add C_1 to the list of clients of C_2 .

Consider a Web browser that specifies, in its prerequisite list, that it requires a TCP/IP service, a window manager, a local file service, and a Java Virtual Machine implementation. Its component configurator should maintain a hook for each of these services. When the browser loads, the system must verify whether these services are available in the local environment. If they are not available, the system must create new instances of them. In any case, the system stores references to the services in the browser configurator hooks and can later retrieve and update them.

The ComponentConfigurator class

We accomplish the reification of runtime dependencies 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. Users can later obtain this pointer through the *implementation()* method.

The *hook()* method specifies that this component depends on 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 on this component.

The *eventOnHookedComponent()* method announces that a component that is attached to this component has generated an event. Subclasses of *ComponentConfigurator()* implement different behaviors to treat events in different ways. Examples of common events are the destruction of a hooked component, the internal reconfiguration of a hooked component, and the replacement of the implementation of a hooked component.

The `eventOnClient()` method is similar to the previous method but it announces that a client has generated an event. This method 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.

The `name()` method 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 such as a list of configuration options accepted by the component, or a URL for its documentation and source code.

The `listHooks()` method returns a pointer to a list of `DependencySpecifications`. A `DependencySpecification` is a structure defined as

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

The `listClients()` method 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 `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.

Toward automatic reconfiguration

Reified intercomponent dependencies can help automate configuration processes. The operating system or middleware can scan the prerequisite list to ensure that all hardware and software requirements for the execution of a particular component are met before the

```
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 ();
};
```

Figure 3. The `ComponentConfigurator` abstract class.

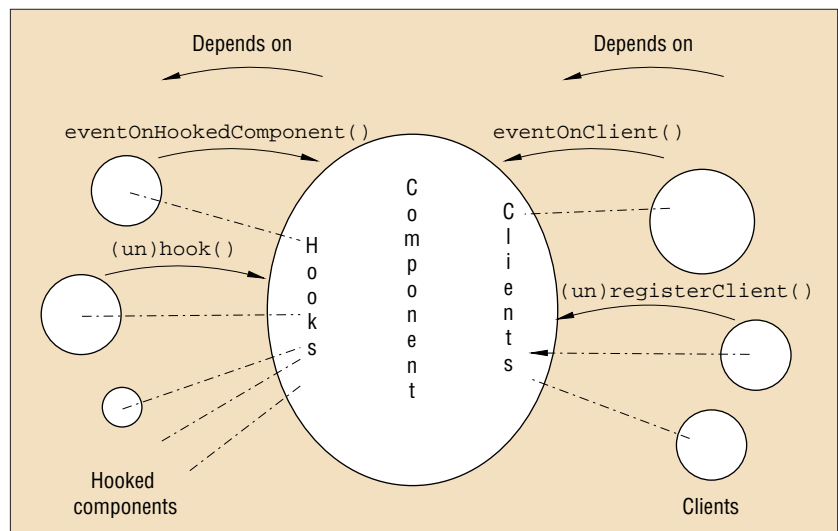


Figure 4. Methods for specifying dependencies and sending events.

component initiates. This can prevent many problems that are common in existing systems where detection of the lack of a particular component or resource happens only after the application is running.

In its turn, the dynamic dependence

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

```

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

```

Figure 5. Customization of the eventOnHookedComponent method.

```

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 ()

```

Figure 6. A ComponentConfigurator destructor.

Continuing with our Web browser example, the application developer could implement a `WebBrowserConfigurator` by using inheritance from the `ComponentConfigurator` and customizing it to handle the dynamic replacement of the system's JVM. Figure 5 shows that the `eventOnHookedComponent` method can be overridden to catch `REPLACED` events coming from the JVM `ComponentConfigurator`.

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

Generally, when replacing an old component with a new one, it might be neces-

sary to transfer the state from the former to the latter. The underlying reconfiguration engine can automate this process 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 that type of component's internal state. 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, we must make sure that no other component will try to contact the component being removed. We can achieve this by using a combination of four mechanisms:

- using the `ComponentConfigurator` to notify all the components that have a reference to the old one,

- using the `ComponentConfigurator` as an indirection on calls to replaceable components,
- leaving a forwarding pointer in place of the old component, and
- 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, we can adopt a fifth option: 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. For example, consider the deletion of a component containing our `ComponentConfigurator` class. We can adopt different policies for dealing with component deletion. 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.

We can customize implementations of this destructor to adjust its behavior to different component types and to meet special application requirements. Different component types must properly implement methods such as `eventOnHookedComponent()` 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 another extreme case, these other components will only be notified and nothing else will change. In most cases, we expect that these components will try to reconfigure themselves 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 naive solution we could use would be to execute the method calls asynchronously, for example, by creating new threads to perform the calls. This solution would incur the additional cost of creating new threads and could lead to dangerous situations because 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 might crash the whole system by executing invalid code. We believe that there are alternatives² that lie somewhere between these two extremes, alternatives that are as safe as the conservative solution but are less subject to blocking.

Managing dependencies

Using 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 is done through controlled interfaces. However, to prevent a proliferation of programming errors related to dependence reification, we need 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. For example, Iguana,³ an extension of C++, reifies several features of the language, allowing dynamic modification of their implementations. We can use reflective languages to instrument a 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 but to use widely accepted standards.

Another possible solution is to tie together the mechanisms for communication and dependence representation using, for example, architectural connectors (see the “Related work” sidebar). However, our objective is not to limit the expressiveness of the model but to develop a generic methodology that we can use in numerous heterogeneous environments. These requirements can only be met when we use a standard architecture such as CORBA.

A major goal of our research is not to limit the implementation to a particular programming language but to use widely accepted standards.

CORBA COMPONENT CONFIGURATOR

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 do not get translated into bad base language references (such as dangling C++ pointers). Instead, the runtime neatly handles exceptions and informs the application of its occurrence.

In our model's CORBA implementation, a `DependencySpecification` stores a CORBA interoperable object reference (IOR) so that the `ComponentConfigurator` is able to reify dependencies between distributed components. We can specify software component prerequisites either in terms of persistent IORs⁴ or in terms of a pair `<ServiceTypeName, Constraints>`. In the former case, we can use an

implementation repository to dynamically create a new CORBA object if one is not available. In the latter case, we can use the CORBA Trading Object Service⁵ 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 crashes, 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`.

In future work, we intend to experiment with different ways of using the `CORBA ComponentConfigurator` to manage distributed applications. In particular, component configurators can be co-located with their respective component implementations, located in a separate process in the same machine, or located in a central node on the network while the component implementations are distributed. We might adopt a combination of two of these schemes. For example, each component can have a colocated instance of the `ComponentConfigurator` as well as another instance 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 colocated instance would provide fast interaction between each component and its configurator. Finally, the redundant information would aid fault tolerance because the information lost with the central node's failure can be reconstructed by contacting the distributed instances.

CONCURRENCY

In multithreaded and multiprocess environments, we must take additional care with regard to reliability and consistency because two threads accessing the same object concurrently might leave the system in an inconsistent state or cause its failure. One of our framework's `ComponentConfigurator` subtypes uses locks to protect the configurator from simultaneous updates by multiple clients. Clients can also use these locks 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, we can coordinate the access to distributed configurators by using the standard concurrency control service.⁵ Ideally, a configuration system should provide support for grouping operations into atomic transactions that satisfy the ACID properties: atomicity, consistency, isolation, and durability. We can achieve this by using the CORBA object transaction service.⁵

SECURITY

In networked environments, we must secure the configuration system from unauthorized access. A hostile agent obtaining access to the `ComponentConfigurators` might totally disrupt system activities. Even read-only access might be dangerous because sensitive information about the internal structure of an institution's system can be stolen. Therefore, we must provide access control to the configuration system. In some cases, it is also desirable to prevent eavesdropping by encrypting the messages exchanged by components and `ComponentConfigurators`, such as the ones containing reconfiguration events.

To support security in environments such as Java, we must extend the configuration model to make it security-aware. On the other hand, in environments supporting reflection and in CORBA, we can define security policies and deploy

security mechanisms without modifying our model.

We can use the CORBA security service⁵ to add message-level interceptors to the ORB so that every data exchange between CORBA objects is properly encrypted. In addition, we can use request-level interceptors to control the access to each individual operation on the `ComponentConfigurators` based on who issues the call, capabilities, or any other customized mechanism the programmer defines.

In environments supporting reflection and in CORBA, it is possible to define security policies and deploy security mechanisms without modifying our model.

DYNAMIC ADAPTABILITY

Although we use the prerequisites primarily to load new components into the system and make sure that their QoS expectations are met, we can also use the prerequisites later for dynamic adaptation to resource availability changes. Typically, the prerequisites' resource requirements should specify ranges of acceptable service. A video-on-demand application, for example, can specify that it requires a network bandwidth of 500 kilobits per second on average but that it might use peak rates of up to 1 megabit per second. In addition, the application can specify that although 500 Kbps is the desirable average bandwidth, it can function using as little as 53 Kbps by changing the video stream characteristics. In that case, the application would be able to support mobile computers that move from ATM to wireless to modem connections by dynamically adapting to these changes. Thus, prerequisites should be available to the system at run-time so that it can reorganize its resource allocation to better fulfill the require-

ments of all the applications sharing the system.

Application scenarios

We investigated the deployment of the `ComponentConfigurator` framework in both centralized and distributed applications. On the one hand, *dynamicTAO*, a reflective ORB, illustrates how we can use our model to represent and manipulate the internal structure of a centralized legacy system, enabling dynamic reconfiguration. On the other hand, the 2K distributed operating system shows how we can use our model in the early system design phases to achieve maximum levels of reliability and dynamic flexibility.

DYNAMICTAO

One of the major constituent elements of 2K is a reflective middleware layer⁶ based on CORBA. After carefully studying existing ORBs, we concluded that the TAO ORB⁷ 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 pattern⁸ to separate different aspects of the ORB internal engine. Administrators use a configuration file to specify the strategies the ORB uses to implement aspects such as concurrency, request demultiplexing, scheduling, and connection management. At ORB startup time, the configuration file is parsed and selected strategies are loaded.

TAO is primarily targeted for static, hard real-time applications such as avionics systems. After the initial ORB configuration, TAO assumes that its strategies will remain in place until it completes its execution. There is little support for on-the-fly reconfiguration.

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

The 2K design 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. We can use the interface for dynamic reconfiguration of servants running on top of the ORB and even for reconfiguring nonCORBA applications.

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 to replace one of its strategies (S_{old}) with a new strategy (S_{new}). The first problem is that because TAO strategies are implemented as C++ objects that communicate through method invocations, the system must be sure that no one is running S_{old} code and that no one expects to run S_{old} code in the future before the system unloads S_{old} . Otherwise, the system might crash. Thus, it is important to assure that S_{old} is unloaded only 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 with S_{new} , part of the internal state of S_{old} might need to be transferred to S_{new} .

We can address these problems with the help of the `ComponentConfigurator`, which is used to reify the dependencies among strategies, instances of dynamicTAO, and servants.

DomainConfigurator and TAOConfigurator

Each process running the dynamicTAO ORB contains a `ComponentConfigurator` instance called `DomainConfigurator`. It is responsible for maintaining references to ORB instances and servants running in that process. In addition, each ORB instance contains a customized subclass of the `ComponentConfigurator` called `TAOConfigurator`.

`TAOConfigurator` contains hooks

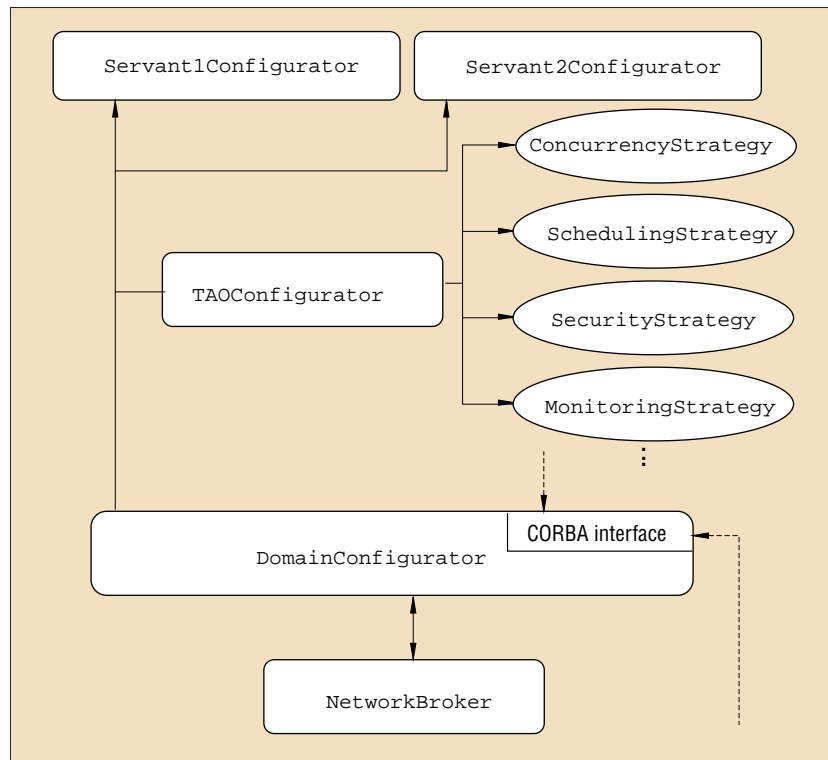


Figure 7. Remote configuration of dynamicTAO strategies.

to which dynamicTAO strategies are attached. A `NetworkBroker` implements a simple TCP-based protocol that lets remote entities connect to the process to inspect and change the dynamicTAO configuration 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 might have their own customized subclass of `ComponentConfigurator` to manage their dependencies on ORB instances and other strategies. These subclasses may also store references to client connections that depend on them. With this information, the system can decide when to unload a strategy safely.

Consider, for example, the three concurrency strategies dynamicTAO supports—single-threaded, thread-per-connection, and thread-pool. If we switch from the thread-per-connection or reactive strategies to another concurrency strategy, we do not need to do anything special. dynamicTAO might simply load the new strategy, update the proper `TAOConfigurator` hook, unload the old strategy, and continue. Old client

connections will complete using the concurrency policy dictated by the old strategy. New connections will use the new policy.

However, if we switch from the thread-pool strategy to another strategy, we must take special care. Our thread-pool strategy maintains a pool of threads created when the strategy initializes. All incoming connections share the threads in order to achieve a good level of concurrency without having the runtime overhead of creating new threads. A problem arises when we switch from this strategy to another strategy: the code of the strategy being replaced cannot be immediately unloaded. This happens because reused threads return to the thread-pool strategy code each time a connection finishes. We can solve this problem by using a `ThreadPoolConfigurator` to keep 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 we replace the thread-pool strategy with a new one. There might be several incoming connections that are queued in the strategy and are waiting for a thread to

execute them. The solution is to use the *Memento* pattern⁸ to encapsulate the old strategy state in an object that is passed to the new strategy. The object is used to encapsulate the queue of waiting connections. The system simply passes this object to the new strategy, which then takes care of the queued connections.

Our group is currently expanding the set of dynamicTAO strategies that can be replaced on-the-fly. At the present, `TAOConfigurator` hooks hold strategies for concurrency, security, and performance monitoring. We plan to add hooks for connection management, (de)marshalling, request demultiplexing, method dispatching, transport protocols, and scheduling. With dynamicTAO, we learned that an explicit knowledge of the dependencies between the ORB components is essential to implement dynamic reconfiguration safely.

ARCHITECTURAL AWARENESS IN 2K

In contrast to existing systems where a large number of unused modules are carried along with the basic system installation, we base the 2K operating system on a “what you need is what you get” (WYNIWYG) model.¹ The system configures itself automatically and loads the minimum set of components required to execute user applications efficiently. The system downloads components from the network and only a small subset of system services is needed to bootstrap a node.

We achieve this by reifying the hardware and software prerequisites for each loadable component. As mentioned earlier, the operating system uses 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 to load the component and also to perform admission control and resource reservation. That way, we are less likely to encounter a situation in which a component fails to execute its task with the desired QoS because an unspecified dependency was not resolved.

As a component is loaded into the system, the component’s 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 2K design supports fault-tolerant, self-adapting systems by monitoring the environment and maintaining a dynamic structural representation 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 might alternatively recover from a failure by replacing the faulty component with a new one.

A *local resource manager*, that resides in each machine and monitors resource utilization, supports QoS. Changing parameters such as network bandwidth, CPU load, memory availability, and user access patterns might trigger adaptations and resource reallocation based on the component prerequisites, which are accessible at runtime.

Implementation status and future work

We implemented prototypes of the `ComponentConfigurator` for single-process applications in C++ and Java. We deployed the C++ implementation in the dynamicTAO ORB. Researchers at the University of São Paulo are using the Java implementation 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` that was based on CORBA. We are using the implementation to construct 2K distributed services such as the persistent object service and the automatic configuration service.

We provide complete documentation and source code for the framework in C++, Java, and CORBA/C++ at our Web

site: choices.cs.uiuc.edu/2K/Dynamic-Configuration.

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 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 we must resolve 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, process 1405/95-2. The National Science Foundation grants NSF 98-70736 and 99-70139 support this research. We gratefully acknowledge the help provided by Manuel Román on the implementation of dynamicTAO. We thank Dilma Menezes, Francisco Ballesteros, and the 2K team members for their feedback on the ideas presented here. Finally, we thank the anonymous reviewers and Murthy Devarakonda who contributed valuable comments for improving this article.

References

1. F. Kon, D. Carvalho, and R. Campbell, “Automatic Configuration in the 2K Operating System,” *Proc. ECOOP ’99 Workshop Object Orientation and Operating Systems*, Chemnitz Informatik-Berichte, Germany, June 1999, pp. 10–14.
2. F. Kon and R.H. Campbell, *On the Role of Inter-Component Dependence in Supporting Automatic Reconfiguration*, Tech. Report UIUCDCS-R-98-2080, Dept. Computer Science, Univ. of Illinois, Urbana-Champaign, Dec. 1998.
3. B. Gowing and V. Cahill, “Meta-Object Protocols for C++: The Iguana Approach,” *Proc. Reflection ’96* Apr. 1996, pp. 137–152.
4. M. Henning, “Binding, Migration, and

- Scalability in CORBA," *Comm. ACM*, Vol. 41, No. 10, Oct. 1998, pp. 62–71.
5. *CORBA Services: Common Object Services Specification*, OMG Document 98-12-09, Object Management Group, Framingham, Mass., 1998.
 6. A. Singhai, A. Sane, and R. Campbell, "Quarterware for Middleware," *Proc. 18th Int'l Conf. Distributed Computing Systems (ICDCS)*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1998, pp. 192–201.
 7. D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Comm.*, Vol. 37, No. 4, May 1999, pp. 54–63.
 8. E. Gamma et. al., *Design Patterns: Elements of Object-Oriented Software*, Addison-Wesley, Reading, Mass., 1995.
 9. M. Roman, F. Kon, and R.H. Campbell, "Design and Implementation of Runtime Reflection in Communication Middleware: the dynamicTAO Case," *Proc. ICDCS '99 Workshop Middleware*, IEEE Computer Soc. Press, Los Alamitos, Calif., June 1999, pp. 122–127.

Fabio Kon is a PhD candidate in computer science at the University of Illinois at Urbana-Champaign. He is working on the overall design and implementation of the 2K distributed operating system. His research interests include distributed operating systems, multimedia, and computer music. He developed SODA, a consistent distributed file system based on leases; MAXAnnealing, a tool for algorithmic musical composition; a scalable multimedia distribution system; and dynamicTAO, a dynamically configurable reflective ORB. He received his BS and MS in computer science from the University of São Paulo and his BA in music from the São Paulo State University. Contact him at Digital Computer Lab, 1304 W. Springfield Ave., Urbana, IL, 61801; 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. 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 received his BSc in mathematics from the University of Sussex, and his MSc and 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.