Dynamic Resource Management and Automatic Configuration of Distributed Component Systems*

Fabio Kon†
Department of Computer Science
University of São Paulo, Brazil
kon@ime.usp.br http://www.ime.usp.br/~kon

Tomonori Yamane
Energy and Industrial Systems Center
Mitsubishi Electric Corporation
yamane@isl.melco.co.jp

Christopher K. Hess Roy H. Campbell M. Dennis Mickunas
Department of Computer Science
University of Illinois at Urbana-Champaign
{ckhess,roy,mickunas}@cs.uiuc.edu
http://choices.cs.uiuc.edu/2k

Abstract

Component technology promotes code-reuse by enabling the construction of complex applications by assembling off-the-shelf components. However, components depend on certain characteristics of the environment in which they execute. They depend on other software components and on hardware resources.

In existing component architectures, the application developer is left with the task of resolving those dependencies, i.e., making sure that each component has access to all the resources it needs and that all the required components are loaded. Nevertheless, according to encapsulation principles, developers should not be aware of the component internals. Thus, it may be difficult to find out what a component really needs. In complex systems, this manual approach to dependency management can lead to disastrous results.

In this paper, we propose an integrated architecture for managing dependencies in distributed component-based systems in an effective and uniform way. The architecture supports automatic configuration and dynamic resource management in distributed heterogeneous environments. We describe a concrete implementation of this architecture and present experimental results.

*This research is supported by the National Science Foundation, grants 98-70736, 99-70139, and EIA99-72884EQ.

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

1 Introduction

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

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

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

Unfortunately, the existing software infrastructure is not prepared to manage these highly-dynamic environments properly.

Existing component-based systems face significant problems with reliability, administration, architectural organization, and configuration. The problem behind all these difficulties is the lack of a unified model for representing dependencies and mechanisms for dealing with these dependencies. Components depend on hardware resources (such as CPU, memory, and special devices) and software resources (such as other components, services, and the operating system). Not resolving these dependencies properly compromises system efficiency and reliability.

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

Software is in constant evolution and new component versions are released frequently. How can one run the most up-to-date components and make sure that they work together in harmony? This requires mechanisms for (1) code distribution over wide-area networks so we can push or pull new components as they become available and (2) safe dynamic reconfiguration so we can plug new components when desired.

In previous papers, we introduced a model for representing dependencies in distributed component systems [KC99] and described a reflective ORB that supports dynamic component loading in distributed environments [KRL+00, KGA+00]. In this paper, we extend our previous work by describing the design, implementation, and performance of an integrated architecture that provides mechanisms for:

1. Automatic configuration of component-based applications.
2. Intelligent, dynamic placement of applications in the distributed system.
3. Dynamic resource management for distributed heterogeneous environments.
4. Component code distribution using push and pull methods.
5. Safe dynamic reconfiguration of distributed component systems.

1.1 Paper Contents

Section 2 gives a general overview of our architecture for automatic configuration and dynamic resource management. Section 3 details the automatic configuration mechanisms, explaining the concepts of prerequisites (Section 3.1), component configurators (Section 3.2), and the Automatic Configuration Service (Section 3.3). Section 4 describes the Resource Management Service, addressing resource monitoring (Section 4.1) resource reservation (Section 4.2), application execution (Section 4.3), and fault-tolerance and scalability (Section 4.4).

Section 5 gives additional implementation details and present experimental results. We then present related work (Section 6), future work (Section 7), and our conclusions (Section 8).

2 Architectural Framework

To deal with the highly-dynamic environments of the next decades, we propose an architectural framework divided in three parts. First, a mechanism for dependence representation lets developers specify component dependencies and write software that deals with these dependencies in customized ways. Second, an Automatic Configuration Service is responsible for dynamically instantiating component-based applications by analyzing and resolving their component dependencies at runtime.

A Resource Management Service is responsible for managing the hardware resources in the distributed system, exporting interfaces for inspecting, locating, and allocating resources in the distributed, heterogeneous system.

Figure 1 presents a schematic view of the major elements of our architecture. *Prerequisite specifications* reify static dependencies of components towards its environment while *component configurators* reify dynamic, runtime dependencies.

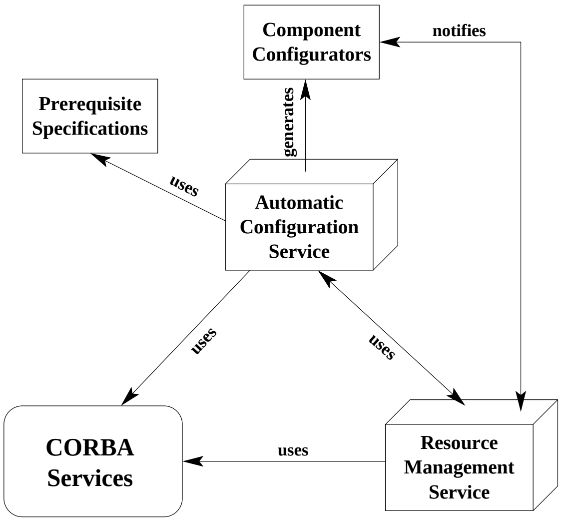

Figure 1: Architectural Framework

As we explain in Section 3, the automatic configuration process is based on the prerequisite specifications and constructs the component configurators. As the Automatic Configuration Service instantiates new components, it uses the Resource Management Service to allocate resources for them. At execution time, changes in resource availability may trigger call-backs from the Resource Management Service to component configurators so that components can adapt to significant changes in the underlying environment.

As described in Section 4.3, when a client requests the execution of an application to the Resource Management Service, the latter finds the best location to execute the application and then uses the Automatic Configuration Service to load the application components.

The elements of the architecture are exported as CORBA services and their implementation relies on standard CORBA services such as Naming and Trading [OMG98].

We have employed the architecture presented here to support a reliable, dynamically configurable Multimedia Distribution System. Readers interested in a detailed description of how our services were used in that particular application scenario should refer to [KCN00]. In the following sections, we provide a more in-depth description of each of the elements of the architecture.

3 Automatic Configuration

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

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

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

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

1. Network-Centrism and
2. a "What You Need Is What You Get" (WYNIWYG) model.

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

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

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

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

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

3.1 Prerequisites

The prerequisites for a particular inert component (stored on a local disk or on a network component repository) must specify any special requirements 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., other components) it requires.

The first two items are used by the 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 determines which auxiliary components must be loaded and in which kind of software environment they will execute.

The first two items – reminiscent of the Job Control Languages of the mid-1960s – can be expressed by modern QoS specification languages such as QML [FK99b] and QoS aspect languages [LBS+98], or by using a simpler format such as SPDF (see Section 3.4.1). The third item is equivalent to the *require* clause in architectural description languages like Darwin [MDK94] and module interconnection languages like the one used in Polylith [Pur94].

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

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

3.2 Component Configurator

The explicit representation of *dynamic* dependencies is achieved through special objects attached to each relevant component at execution time. These objects are called *component configurators*; they are responsible for reifying the runtime dependencies for a certain component and for implementing policies to deal with events coming from other components.

While the Automatic Configuration Service parses the prerequisite specifications, fetches the required components from the Component Repository, and dynamically loads their code into the system runtime, it uses the information in the prerequisite specifications to create component configurators representing the runtime inter-component dependencies. Figure 2 depicts the dependencies that a component configurator reifies.

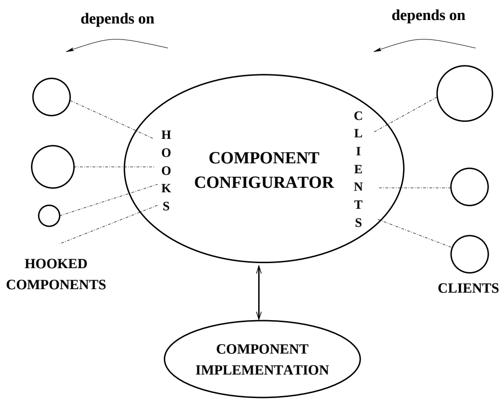

Figure 2: Reification of Component Dependencies

The dependencies of a component C are managed by a component configurator C^c. Each configurator C^c has a set of *hooks* to which other configurators can be attached. These are the configurators for the components on which C depends; they are called *hooked components*. The components that depend on C are called *clients*; C^c also keeps a list of references to the clients' configurators. In general, every time one defines that a component C_1 depends on a component C_2, the system should perform two actions:

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

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

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

As an example of how customized component configurators could help applications, consider a QoS-sensitive video-on-demand client that reserves a portion of the local CPU for decoding a video stream. The application developer can program a special configurator that registers itself with the Resource Management Service. In this way, when the Resource Management Service detects a change in resource availability that would prevent the application from getting the desired level of service, it notifies the configurator (as shown in Figure 1). The configurator, with its customized knowledge about the application, sends a message to the video server requesting that the latter decrease the video frame rate. Then, with a lower frame rate, the client is able to process the video while the limited resource availability persists. When the resources go back to normal, another notification allows the video-on-demand configurator to re-establish the initial level of service.

3.3 Automatic Configuration Service

As described above, automatic configuration enables the construction of network-centric systems following a WYNIWYG model. To experiment with these ideas, we developed an Automatic Configuration Service for the *2K* operating system [KCM+00].

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

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

Figure 3 shows how the architecture uses the two basic classes of the Automatic Configuration framework: *prerequisite parsers* and *prerequisite resolvers*. Administrators and developers can plug different concrete implementations of these classes to implement customized policies.

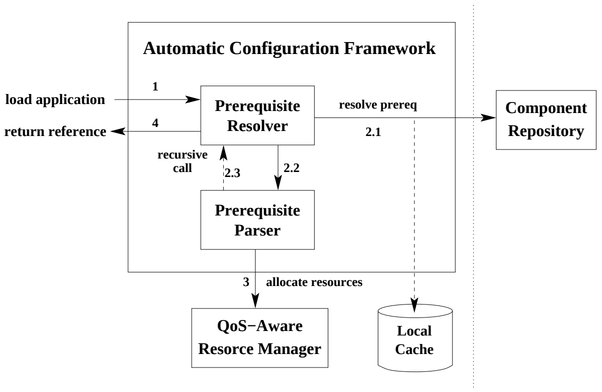

Figure 3: Automatic Configuration Framework

The automatic configuration process works as follows. First, the client sends a request for loading an application by passing, as parameters, the name of the application's "master" component and a reference to a component repository (step 1 in Figure 3). The request is received by the prerequisite resolver, which fetches the component code and prerequisite specification from the given repository, or from a local cache, depending on the policy being used (step 2.1).

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

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

3.4 A Concrete Implementation

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

3.4.1 SPDF

We designed the Simple Prerequisite Description Format (SPDF) to serve as a proof-of-concept for our framework. An SPDF specification is divided in two parts, the first is called hardware requirements and the second, software requirements. Figure 4 shows an example of an SPDF specification for a hypothetical web browser. The first part specifies that this application was compiled for a Sparc machine running Solaris 2.7, that it requires at least 5MB of RAM memory but that it functions optimally with 40 MB of memory, and that it requires 10% of a CPU with speed higher than 300MHz.

The second part, software requirements, specifies

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

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

Figure 4: A Simple Prerequisite Description

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

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

3.4.2 Simple Resolver and Caching Resolver

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

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

3.5 Simplifying Management

The Automatic Configuration Service simplifies management of user environments in distributed systems greatly. Whenever a new application is requested, the service downloads the most up-to-date version of its components from the network Component Repository and installs them locally. This provides several advantages including the following.

- It eliminates the need to upload components to the entire network each time a component is updated.
- It eliminates the need to keep track manually of which machines hold copies of each component because updates are automatic.
- It helps machines with limited resources, which no longer need to store all components locally.

3.6 Pushing Component Updates

The automatic configuration mechanism described here provides a pull-based approach for code updates and configuration. In other words, the service running in a certain network node takes the initiative to *pull* the code and configuration information from a Component Repository.

To support efficient and scalable management in large-scale systems, it may be desirable to allow system administrators to *push* code and configuration information into the network. Our architecture achieves this by using the concept of *mobile reconfiguration agents*, which we describe in detail elsewhere [KGA$^+$00].

4 Resource Management Service

The Resource Management Service [Yam00] is organized as a collection of CORBA servers that are responsible for (1) maintaining information about the dynamic resource utilization in the distributed system, (2) locating the best candidate machine to execute a certain application or component based on its QoS prerequisites, and (3) allocating local resources for particular applications or components.

As shown in Figure 5, the Resource Management Service relies on Local Resource Managers (LRMs) present in each node of the distributed system. The LRM's task is to export the hardware resources of a particular node to the whole network. The distributed system is divided in clusters and each cluster is managed by a Global Resource Manager (GRM).

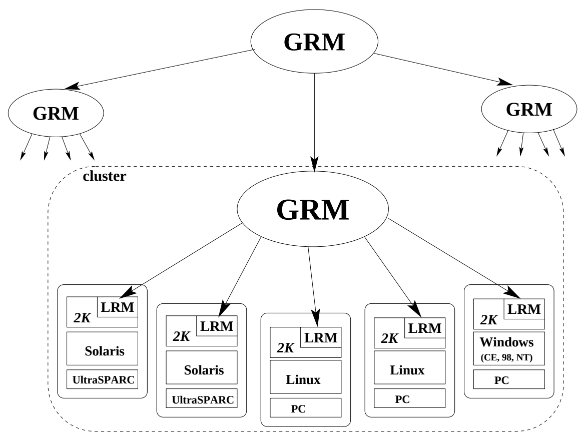

Figure 5: Resource Management Service

4.1 Resource Monitoring

The LRMs running in each network node send updates of the state of their resources (e.g., CPU and memory usage) to the GRM periodically. The GRM implementation encompasses an instance of the standard OMG Object Trading Service [OMG98]. A reference to the LRM of each machine in the cluster is stored in the GRM database as a trader "service offer" and the state of its resources is stored as the offer's "properties".

To reduce network and GRM load, it is important to limit the frequency in which the LRMs send their updates to the GRM. Thus, although LRMs check the state of their local resources frequently (e.g., every ten seconds), they only send this information to the GRM when (1) there were significant changes in resource utilization since the last update (e.g., a variation in more than 20% on the CPU load) or (2) a certain time has passed since the last update was sent (e.g., three minutes). In addition, when a machine leaves the network, in case of a shutdown or a voluntary disconnection of a mobile computer, the LRM unregisters itself from the GRM database. If the GRM does not receive an update from an LRM for a period twice as long as the time in item 2 above, it assumes that the machine with that LRM is unaccessible.

4.2 Resource Reservation

The LRMs are also responsible for performing QoS-aware admission control, resource negotiation, reservation, and scheduling of tasks on a single node. This is achieved with the help of a Dynamic Soft Real-Time Scheduler [NhCN98] that runs as a user-level process in conventional operating systems like Solaris and Windows. The LRM works as a CORBA wrapper for this scheduler, which uses the system's low-level real-time API to provide QoS guarantees to applications with soft real-time requirements.

This CORBArized scheduler can be used at any time by CORBA clients to request QoS guarantees on the availability of CPU and memory. For example, as explained in Section 3.3, a prerequisite parser may issue requests to reserve CPU and memory based on a component's hardware prerequisite specifications.

4.3 Executing Applications

Both the LRM and the GRM export an interface that let clients execute applications (or components) in the distributed system. The GRM maintains an approximate view of the cluster resource utilization state and it uses this information as a hint for performing QoS-aware load distribution within its cluster.

When a client wishes to execute a new application, it sends an `execute_application` request to the local LRM. The LRM checks whether the local machine has enough resources to execute the application comfortably. If not, it forwards the request to the GRM. The latter uses its information about the resource utilization in the distributed system to select a machine that would be the best candidate to execute that application and forwards the request, as a `oneway` message, to the LRM of that machine. The LRM of the latter machine tries to allocate the resources locally, if it is successful, it sends a `oneway ACK` message to the client LRM. If it is not possible to allocate the resources on that machine, it sends a `NACK` back to the GRM, which then looks for another candidate machine. If the GRM exhausts all the possibilities, it returns an empty offer to the client LRM.

When the system finally locates a machine with the proper resources, it creates a new process to host the application. Next, it uses the Automatic Configuration Service to fetch all the necessary components (i.e. the master component's dependencies) from the Component Repository and dynamically load them into that process as described in Section 3.3.

4.3.1 Client Request Format

The format of the client request to the initial LRM is the following.

```
CosTrading::OfferSeq execute_application (
   in string categoryName,
   in string componentName,
   in string args,
   in CosTrading::PropertySeq QoS_spec,
   in CosTrading::Constraint platform_spec,
   in CosTrading::Preference prefs,
   in CosTrading::Lookup::SpecifiedProps
                                  return_props
);
```

`categoryName/componentName` specify which of the components in the *2K* Component Repository is the master component of the application to be executed and `args` contains the arguments that should be passed to it at startup time.

`QoS_spec` defines the quality of service required for this application. It is specified as a list of `<resourceName,resourceValue>` pairs. As an example, if the resource is the CPU, then the resource value should be a structure of the following format (specified by the scheduler's CPU server [NhCN98]).

```
struct CpuReserve {
   long serviceClass;
   long period;
   long peakProcessingTime;
   long sustainableProcessingTime;
   long burstTolerance;
   float peakProcessingUtil;
};
```

`platform_spec` is the criteria to select a cluster node and it is specified using the OMG Trader Constraint Language. For example, `(os_name == 'Linux') and (processor_util < 40)` will select a Linux machine whose CPU utilization is less than 40%.

`prefs` specifies the preferred machine in case multiple machines satisfy the requirements. For example, `max(RAM_free)` will select the machine with the maximum available physical memory.

Finally, `return_props` specifies which properties (resource utilization information) should be included in the service offer that is returned. The returned value also includes a reference to the component configurator (see Section 3.2) of the new application.

4.4 Fault-Tolerance and Scalability

To provide fault-tolerance and scalability, the Resource Management Service architecture depends on a collection of replicated GRMs in each cluster. LRMs send their updates as a multicast message to all the GRMs in the cluster. Since, strong consistency between the GRMs is not required, we can use an unreliable multicast mechanism. Client requests are sent to a single GRM and different clients may use different GRMs for load balancing.

To enhance scalability across multiple clusters connected through the Internet, GRMs can be federated in a hierarchical way. If a request cannot be resolved in a particular cluster, the GRM forwards it to a parent GRM in the hierarchy. The parent GRM maintains an approximate view of the resource utilization in its child clusters and uses this information as a hint to locate a proper cluster to fulfill the client request.

Although we have designed the protocols and algorithms for fault-tolerance and scalability mentioned in this subsection, their implementation is still underway.

5 Implementation and Experimental Results

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

To evaluate the performance of the Automatic Configuration Service, we instrumented a test application [KCN00] to measure the time for fetching, dynamic linking, and configuring its constituent components.

5.1 Loading Multiple Components

Figure 6 shows the total time for the service to load from one to eight components of 19.2Kbytes each. These experiments were carried out on two Sparc Ultra-60 machines running Solaris 7 and connected by a 100Mbps Fast Ethernet network. The Component Repository was executed on one of the machines and the test application with the Automatic Configuration Service on the other. Each value is the arithmetic mean of five runs of the experiment. The vertical bars in the subsequent graphs and the numbers in parentheses in Table 1 represent the standard deviation. As the graph shows, the variation in execution times across different runs of the experiment was very small.

Table 1 shows, in more detail, how the service spends its time when loading a single 19.2Kbyte component. The current version of the Automatic Configuration Service fetches the prerequisites file from the remote Component Repository and saves it to the local disk. The same is done with the file

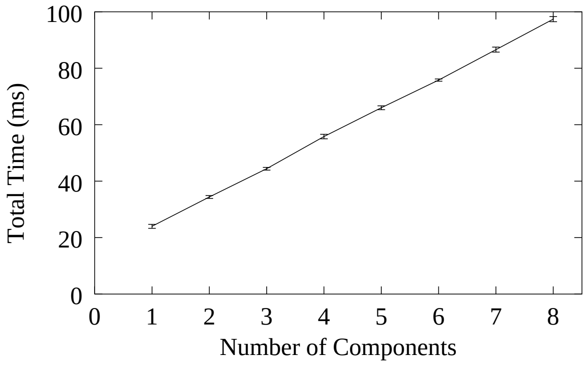

Figure 6: Automatic Configuration Service Performance

containing the component code. Then, it uses the underlying operating system to perform the local dynamic linking of the component into the process runtime.

The table also shows the additional time spent by the service (row labeled as “autoconf protocol additional operations”) to detect if there are more components or prerequisite files to load, to parse the prerequisite file, and to reify dependencies. This overhead accounts for 46% of the total time required to load the component, which suggests that it would be desirable to improve this part of the service by optimizing the implementation of the *SimpleResolver* (see Section 3.4). We believe that an optimized version of the *SimpleResolver* could lead to improvements in the order of 20% for components of this size.

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

5.2 Components of Different Sizes

To evaluate how the time for loading a single component varies with the component size, we created a program that generates components of different sizes. According to its command-line arguments, this program generates C++ source code containing a given number of functions (which include code to perform simple arithmetic operations) and local and

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

Table 1: Discriminated Times for Loading a 19.2Kbyte Component

global variables. Using this program, we created components whose DLL sizes vary from 12 to 115 Kbytes. Figure 7 shows the time for the Automatic Configuration Service to load a single component as the component size increases.

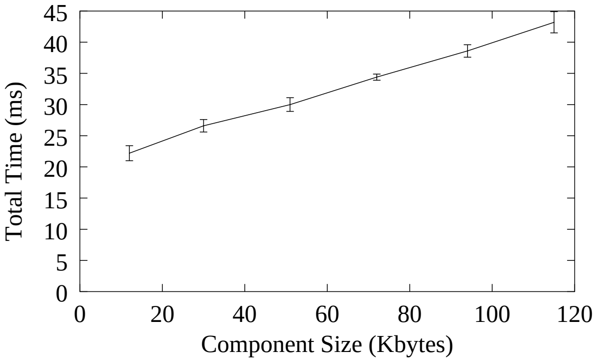

Figure 7: Times for Loading Components of Different Sizes

Figure 8 shows the absolute times spent in each step of the process[3]. We can notice that the time spent in the item labeled "autoconf protocol" is approximately constant[4]. Hence, as the component size increases, its relative contribution to the total time decreases. This can be noticed in Figure 9, which shows the same data in a different form. In this case, the figure shows the percentage of the total time spent in each of the steps of the process.

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

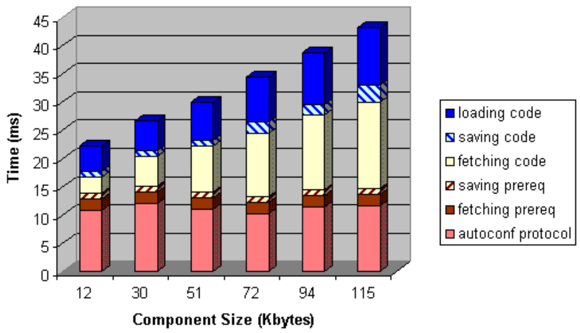

Figure 8: Discriminated Times for Loading Components of Different Sizes

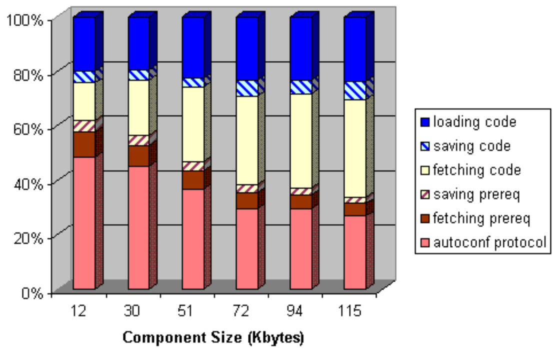

Figure 9: Discriminated Percentual Times for Loading Components of Different Sizes

important with respect to performance. This suggests that deriving intelligent algorithms for component caching, taking component versions and user access patterns into consideration is an important topic for future research.

Although there is still much room for improvements and performance optimizations in the protocols used by the Automatic Configuration Service, the results presented here are very encouraging. They demonstrate that it is possible to carry out automatic con-

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

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

figuration of a distributed component-based application within a tenth of a second, which is what we intended to prove.

5.3 Resource Management

We have not yet carried out an extensive performance evaluation of the Resource Management Service. However, preliminary results [Yam00] show that the overhead imposed by the LRMs in the individual nodes is low and that the time for launching a simple remote application through the GRM and LRM is in the order of a tenth of a second. As future work, we intend to carry out a comprehensive evaluation of this service.

6 Related Work

The OMG CORBA Component Model (CCM) specifies a standard framework for building, packaging, and deploying CORBA components [OMG99]. Unlike our model, which focuses on prerequisites and dynamic dependencies, the CORBA Component Model concentrates on defining an XML vocabulary and an extension to the OMG IDL to support the specification of component packaging, customization, and configuration. The CCM *Software Package Descriptor* is reminiscent of our SPDF as it contains a description of package dependencies, i.e., a list of other packages or implementations that must be installed in the system for a certain package to work. *CORBA Component Descriptors*, on the other hand, describe the interfaces and event ports used and provided by a CORBA component.

We believe that our model and CCM complement each other and could be integrated. CCM provides a static description of component needs and interactions, while our model manages the runtime dynamics. Although CCM was already approved by OMG, publicly available ORBs do not support it yet. Once this happens, we intend to work towards the integration of the two models.

Among the major CORBA implementations, the one that most resembles our work is Orbix 2000 [ION00]. Its *Adaptive Runtime Architecture* lets users add functionality to the ORB by loading plug-ins dynamically. Whenever a request is sent to the ORB, it is processed by a chain of interceptors that can be configured in different ways using the loaded plug-ins. In that way, the ORB can be configured with interceptors that implement security, transactions, different transport protocols, etc. When the ORB loads a plug-in, it checks its version and dependence information. A centralized configuration repository specifies plug-in availability and configuration settings. Using this architecture it could be relatively easy to implement the functionality provided by our Automatic Configuration and Resource Management Services.

Enterprise JavaBeans [Tho98] is a server-side technology for the development of component-based systems. It does not support the functionality for Automatic Configuration and Resource Management provided in our system. Nevertheless, it provides *deployment descriptors* that let one define, at deployment time, the configuration of individual components (Beans). Instead of recording the configuration information in a text format – like our SPDF and CORBA's XML formats – deployment descriptors are serialized Java classes. A deployment descriptor can customize the behavior of a Bean by setting environment properties as well as define runtime attributes of its execution context, such as security, transactions, persistence, etc. [MH00].

Jini is a set of mechanisms for managing dynamic environments based on Java. It provides protocols to allow services to *join* a network and *discover* what services are available in this network. It also defines standard service interfaces for leasing, transactions, and events [AOS+99]. When a Jini server registers itself with the Jini lookup service, it stores a piece of Java byte code, called proxy, in its entry in the lookup service. When a Jini-enabled client uses the lookup service to locate the server, it receives, as a reply, a *ServiceItem*, which is composed of a service ID, the code for the proxy, and a set of service attributes. The proxy is then linked into the client address-space and is responsible for communication with the server. In this way, the communication between the client and the server can be customized, and optimized protocols can be adopted.

This Jini mechanism for proxy distribution can be achieved in a CORBA environment by using the Automatic Configuration Service in conjunction with a reflective ORB such as *dynamicTAO* [KRL+00]. The Automatic Configuration Service would fetch the proxy code and dynamically link it, while *dynamicTAO* would use the TAO pluggable protocols framework [OKS+00] to plug the proxy code into the TAO framework.

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

The Globus project [FK98] provides a "computational grid" [FK99a] integrating heterogeneous distributed resources in a single wide-area system. It supports scalable resource management based on a hierarchy of resource managers similar to the ones we propose. Globus defines an extensible Resource Specification Language (RSL) that is similar to our SPDF (described in Section 3.4.1). RSL [Glo00] allows Globus users to specify the executables they want to run as well as their resource requirements and environment characteristics. RSL could be integrated in our system by plugging an *RSLParser* into our Automatic Configuration framework. A fundamental difference between Globus and our work is that we focus on *component-based* applications that are dynamically configured by assembling components fetched from a network repository. In Globus, on the other hand, the user specifies the application to be executed by giving the name of a single executable on the target host file system or by giving a URL from which the executable can be fetched.

Legion [GW+97] is the system that shares most similarities with *2K* as it also builds on a distributed, reflective object model. However, the Legion researchers focused on developing a new object model from scratch. Legion applications must be built using Legion-specific libraries, compiler, and run-time system (the Legion's ORB). In contrast, we focused on leveraging CORBA technology to build an integrated architecture that could provide the same functionality as Legion, while still preserving complete interoperability with other CORBA systems. In addition, our work emphasizes automatic configuration and dependence management, which are not addressed by Legion.

Systems based on architectural connectors like UniCon [SDZ96] and ArchStudio [OT98] and systems based on software buses like Polylith [Pur94] separate issues concerning component functional behavior from component interaction. Our model goes one step further by separating inter-component communication from inter-component dependence. Connectors and software buses require that applications be programmed to a particular communication paradigm. Unlike previous work in this area, our model does not dictate a particular communication paradigm like connectors or buses. It can be used in conjunction with connectors, buses, local method invocation, CORBA, Java RMI, and other methods. As demonstrated by our experiments with *dynamicTAO* [KRL+00], the model was applied to a legacy system without requiring any modification to its functional implementation or to its inter-component communication mechanisms.

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

Research in software architecture [SG96] and dynamic configuration [PCS98] typically focuses on the architecture of individual applications. It does not deal with dependencies of application components towards system components, other applications, or services available in the distributed environment. Our approach differs from them in the sense that, for each component, we specify its dependencies on all the different kinds of environment components and we maintain and use these dynamic dependencies at runtime. Approaches based on software architecture typically rely on global, centralized knowledge of application architecture. In contrast, our method is more decentralized and focuses on more direct component dependencies. We believe that, rather than conflicting with the software architecture approach, our vision complements them by reasoning about *all* the dependencies that may affect reliability, performance, and quality of service.

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

7 Future Work

Under the *2K* project we have also been working on QoS compilation techniques, addressing the problem of translating application-level QoS specifications to component-level QoS specifications, and then to resource-level QoS specifications [NWX00]. In the near future, our group will work on the implementation of the mechanisms for fault-tolerance and scalability described in Section 4.4. Security will be provided by a CORBA implementation of the standard Generic Security Services (GSS) API [Lin97].

In the previous sections, we alluded to some other important topics for future work, namely, (1) automatic creation and refinement of prerequisite specifications, (2) intelligent algorithms for component caching taking versions into consideration, and (3) the integration of our dependence model with recent research in software architecture.

8 Conclusions

Component technologies will play a fundamental role in the next generation computer systems as the complexity of software and the diversity and pervasiveness of computing devices increase. However, component technologies must offer mechanisms for automatic management of inter-component dependencies and component-to-resource dependencies. Otherwise, the development of component-based systems will continue to be difficult and frequently lead to unreliable and non-robust systems.

Although there are still a number of open problems for future research, we believe that this paper gives an important contribution to the area by presenting an object-oriented architecture for automatic configuration and dynamic resource management in distributed component systems. Performance evaluation demonstrated that our system is able to dynamically instantiate applications by assembling network components in less than a tenth of a second.

Future work in our group will extend the Resource Management Service implementation to improve its fault-tolerance and scalability and enhance the synergy between dynamic resource management and automatic configuration.

Acknowledgments The authors gratefully acknowledge the wise input provided by Klara Nahrstedt, Francisco Ballesteros, Manuel Román, Dulcineia Carvalho, Dilma Silva, and the *2K* team. We thank the anonymous reviewers for their useful comments, which helped to improve the quality of this paper.

Availability Source code and more information about the Automatic Configuration Service and the Resource Management Service can be found at `http://choices.cs.uiuc.edu/2k`.

References

[AOS+99] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification.* Addison-Wesley, June 1999.

[CKB+00] Dulcineia Carvalho, Fabio Kon, Francisco Ballesteros, Manuel Román, Roy Campbell, and Dennis Mickunas. Management of Execution Environments in 2K. In *Proceedings of the Seventh International Conference on Parallel and Distributed Systems (ICPADS'2000)*, pages 479–485. IEEE Computer Society, July 2000.

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

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

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

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

[GW+97] Andrew S. Grimshaw, Wm. A. Wulf, et al. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1), January 1997.

[ION00] IONA Technologies. *Orbix 2000*, March 2000. White paper available at `http://www.iona.com`.

[KC99] Fabio Kon and Roy H. Campbell. Supporting Automatic Configuration of Component-Based Distributed Systems. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, pages 175–187, San Diego, CA, May 1999.

[KCM+00] Fabio Kon, Roy H. Campbell, M. Dennis Mickunas, Klara Nahrstedt, and Francisco J. Ballesteros. 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC'9)*, pages 201–208, Pittsburgh, August 2000.

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

[KGA+00] Fabio Kon, Binny Gill, Manish Anand, Roy H. Campbell, and M. Dennis Mickunas. Secure Dynamic Reconfiguration of Scalable CORBA Systems with Mobile Agents. In *Proceedings of the IEEE Joint Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA'2000)*, pages 86–98, Zurich, September 2000.

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

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

[Lin97] J. Linn. The Generic Security Service Application Program Interface (GSS API). Technical Report Internet RFC 2078, Network Working Group, January 1997.

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

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

[MH00] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 2nd edition, March 2000.

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

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

[OKS+00] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons. The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York, April 2000.

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

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

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

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

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

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

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

[Tho98] Anne Thomas. *Enterprise JavaBeans Technology: Server Component Model for the Java Platform*. Patricia Seybold Group, December 1998. Available at `http://java.sun.com/products/ejb/white`.

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

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