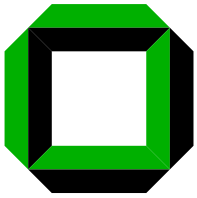


Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme

Lehrstuhl Systemarchitektur

Simplifying Server Configuration Management

Björn Tackmann

Studienarbeit

Verantwortlicher Betreuer: Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter: Joshua LeVasseur

11. Mai 2005

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 11. Mai 2005

_____

Björn Tackmann

Abstract

Network services have gained great importance during the last years. Following the current trend, the number of computers and applications that collaborate to provide these services will grow steadily in the future. The dependencies between the collaborating systems along with heterogeneous management interfaces will continue to increase the complexity of the administration task. This thesis presents a new approach to server configuration management. The integration of the applications into diverse local environments is simplified, and the availability of services is improved by freeing the applications from error-prone tasks.

Contents

List of Figures

Chapter 1

Introduction

Virtually every software product can be customized by means of a runtime configuration. The attributes of the configuration define interconnections between the programs and control their behavior. For example, a common workgroup server provides web- and file services, which require authentication. Via the configuration, the customer declares the files to serve and the authentication service to use.

Server Configuration Management

At present, software vendors integrate the configuration subsystem tightly into the software product. The application stores the configuration data using a particular format, which restricts the customer to employing a specific kind of management interface. On the target machine, the program to be configured parses the configuration data at startup. Thus, errors in the configuration data are not detected until runtime so the applications cannot provide services. In a composite system, the dependencies of the configuration parameters have to be considered to configure the collaborating applications properly. In the case of the workgroup server previously mentioned, the configuration of the file service depends on the configuration of the authentication service.

We propose to separate the configuration subsystem from the application. In our approach, several collaborating systems share one configuration data base. When the configuration data changes, administration utilities parse the data centrally. They generate application-specific objects from the unique data model and deploy these objects to the target machines. The utilities handle dependencies concerning the attributes of different applications automatically. Thus, the configuration data are managed and checked independently of the machines that execute the programs. The proposed approach suggests providing the parser in a separate program, which generates a runtime-compatible object file containing the configuration data. Since the program runs supervisedly, errors in the data can be detected before trying to execute the production system on the configuration. By means of adjusting the parser utility, the configuration task can be adapted to individual needs. Therefore, the integration of applications into the existing software system is simplified.

Some applications have to achieve a high degree of availability. A restart for

configuration updates is unacceptable due to the disruption of the service. Thus, reconfiguring need be done at run time. Nevertheless, the consistency of the application must be preserved. This thesis proposes a solution to replacement of configuration data, which is transparent for the client applications. The described techniques were implemented extending the `thttpd` web server [12].

Outline

This thesis is structured as follows. The related work is reviewed in Chapter 2. In Chapter 3, deployment of configuration data and dynamic reconfiguration of server applications are discussed. The reference implementation is described in Chapter 4. Finally, Chapter 5 summarizes the results of this thesis and gives a survey of the future work.

Chapter 2

Related Work

2.1 Management of Configuration Data

The Kolab groupware environment [8] consists of several independently developed open source applications, including a mail server, a directory server, and a web server. Each of these programs uses proprietary configuration files. A unified web-based interface is used for specifying the configuration data and for generating the files that are parsed at the startup of the server applications. The management interface does not detect errors in the data. Thus, the programs are supplied with erroneous configuration files and cannot provide services properly.

Managing configuration data of workstations and servers centrally is the objective of [2]. Analogous to Kolab, the approach proposes to build the configuration files of the configured applications from a unique data model. During the boot process of the operating system, each machine retrieves the configuration data and creates the application-specific files. Since the data is stored centrally, the administrator manages the applications independently of the target machines, and utilities can check dependencies of the configuration attributes. Nevertheless, the configured applications parse the data, so parsing errors can decrease the availability of the services. Furthermore, the approach constrains the changes to be absorbed at boot time.

2.2 Dynamic Reconfiguration

The separation of the implementation and the configuration of services increases the flexibility of applications. Kramer's Configuration-programming approach suggests implementing a distributed application at a *programming* and a *configuration level* [9]. Endler extends this approach by equipping the components with a *reconfiguration interface* to support consistency-preserving reconfigurations [7]. The enhanced components provide *state predicates* and *consistency operations*, which are accessed by *reconfiguration scripts* to synchronize the reconfiguration with the internal state of the component. In Lira, the concepts of network management are applied to the reconfiguration of distributed applications [6]. The definition of a management protocol is based on the corresponding techniques. *Managers*, which embody the logic for monitoring and reconfigu-

ration activities, use this protocol for accessing attributes of components by means of *reconfiguration agents*. These agents can build composite components by acting as managers and distributing the requests to other reconfiguration agents. A *host agent* runs on each machine and is responsible for installing and activating the components locally. Warren and Sommerville present a model for consistency-preserving reconfiguration of component-based applications [13]. The individual components can be reconfigured as well as added, deleted, and replaced. Synchronization is achieved by using a central runtime kernel for the communication among the components. All of these approaches focus on dynamic reconfiguration of component-based applications. The affected components, however, have to be suspended when they are reconfigured. In this thesis, the components proceed throughout the reconfiguration.

Upgrading the software of long-lived distributed systems is difficult, since halting the system is unacceptable [1]. If the nodes of the distributed system retrieve the new software version independently, two different nodes that need to communicate will possibly run different versions. Therefore, the approach suggests using *Simulation Objects* to map the incoming requests to the interface of the running software version. The K42 operating system kernel supports updates of both code and data structures at run time [3]. In the kernel, all object invocations are made through an *object translation table*, hence adding a level of indirection. A mediator object is interposed by adjusting the entries of the object to be replaced. To assure quiescence, which is important for a consistent update, the mediator traces the generations of the threads that access the object. Both of these approaches use an identification of the caller's version to preserve consistency. The *epoch* described in Section 3.3 serves the same purpose.

Chapter 3

Proposed Solution

Server applications are often assembled to build composite systems, such as described in [8]. Therefore, the customer must be able to adapt these programs to a variety of situations by specifying a runtime configuration. The integration of the applications into the local environment is desired to be of minimal effort. We will use the setting shown in Figure 3.1 to illustrate the problems and techniques throughout this chapter. Consider an internet service provider running

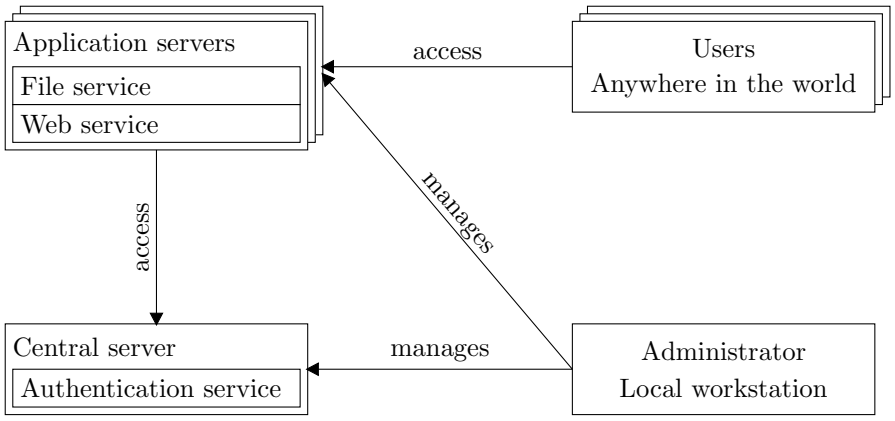


Figure 3.1: An internet service provider's data center

a private data center. Many application servers execute an instance of a file- and a web service each. The servers run in a load-balanced environment and use similar configurations. They access a central authentication service located at a different server. The administrator uses his workstation to manage all of the servers. Lots of users located anywhere in the world access the web- and file services with high demands for availability.

3.1 The Traditional Approach

In current software products, one executable file contains both the configuration subsystem and the functional implementation, since the parser of the configuration data is statically linked to the program. At startup, the application converts the data into an internal representation, which is used at run time. For example, the `thttpd` web server obtains the attribute values from a plain text file, which contains a key-value pair for each parameter [12]. `thttpd` must be restarted to adopt changes of the configuration data, since the parser runs only at startup.

The tight integration of the configuration subsystem into the application has several disadvantages. Parsing the configuration data at startup implies that errors, which prohibit the program to provide services properly, are not detected until runtime. Therefore, the program remains in an undetermined state from which an autonomous recovery is impossible. The administrator must supply the machines that received erroneous configurations with revised data and restart the programs. An application downtime is inevitable. In addition to these issues, the code of the configuration subsystem must be installed on the machines that read the data. Both the parser and the infrastructure that is necessary to obtain the data are required by the target machine. Depending on

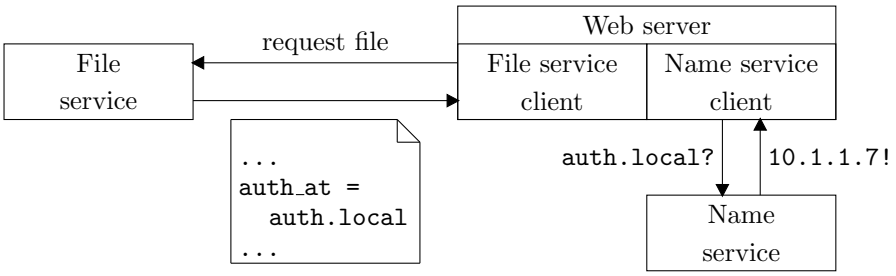


Figure 3.2: Web server retrieving configuration data at startup

the particular implementation, the configuration subsystem uses a file system, a database client, or a similar service for retrieving the configuration data. The additionally installed software increases the management effort. In Figure 3.2, the web server requests the configuration file from the file server. Parameters included in the configuration data require a look-up in an external data base, since the authentication server is referred to by its host name `auth.local` in the configuration file. On the one hand, this denotation of the host is convenient for the user. On the other hand, the current network address, which is `10.1.1.7` in the example, must be requested from the name service. Thus, a name service client has to be installed on the machine that parses the configuration data. The look-ups are also a potential source of runtime errors. For instance, if a required service is not available, the configured application will not be able to run.

When designing a software system, one must often combine components supplied by several vendors. At present, each of these components includes a specific configuration subsystem, which cannot be replaced. The configuration data are distributed among several data bases, which possibly use different formats. The

administration task becomes more complicated, since several management interfaces are used. Furthermore, a customer may require additional languages of the management interface or platforms the interface runs on, which are not implemented by the application vendor. Since the user cannot adjust the configuration subsystem, the integration of the application into the local environment is difficult. For example, if several users may adjust only particular attributes, the simple plain text file used by `thttpd` will not be sufficient to enforce these directives. Moreover, the components that the composite software system consists of are assembled by defining bindings, which are specified in the particular configuration data. In the basic example depicted in Figure 3.1, the file server accesses the authentication service. When the network address of the authentication server is changed, the configuration data of the file servers have to reflect this change. If the applications use independent configuration subsystems, the administrator will have to adjust the configuration data manually.

Several current approaches propose to store the configuration data of multiple collaborating application in one central data base [2, 8]. A utility builds application-specific configuration files from these data. The applications read these files at startup. These approaches simplify the administration task, since they provide a unique management interface for the applications. Nevertheless, the programs parse the configuration files, so the configured machines must supply the infrastructure required to retrieve and parse the files.

3.2 The Configuration Linking Approach

System configuration can be charactarized as a collection of components, such as the server applications in Figure 3.1, interconnections between these components, which are indicated by the network addresses, and values of particular attributes. For the web server, such attributes include the directory that the served documents reside in and a maximal age of pages for client-side caching. All configuration parameters have to be initialized before an application can provide services. The values can be specified by including them in the executable file, requiring the application to be rebuilt whenever the configuration changes. Alternatively, they are obtained at startup, deferring the configuration task from the developer to the user. Both ways result in the same runtime representation of the parameters. In the first case, the developer includes constant values in the source code of the application. The compiler transforms these values into a representation that is specific to the target platform. In the second case, the user creates a runtime configuration that contains the values. The application performs the transformations at run time.

Runtime configuration is used to adapt software individually. Figure 3.3 depicts the model of the configuration data lifecycle used in this thesis. By means of a *configuration editor*, the user generates a specification `spec` of the desired configuration, which is stored persistently in a *configuration data base*. The format of the specification does not depend on the particular application, since the defined requirements can be fulfilled by several implementations. The *configuration parser* retrieves the specification from the data base and generates a *runtime representation* `RR` of the configuration. This form is specific to the particular implementation and the runtime environment. For instance, a list contained in the specification can be transformed into a binary tree, a hash map,

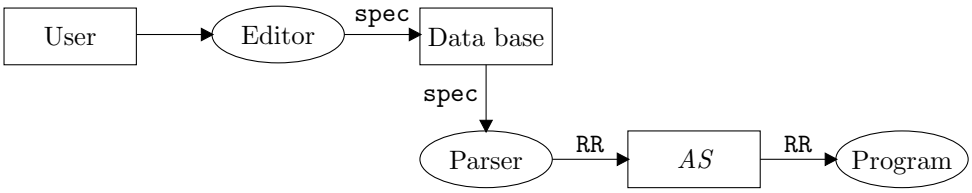
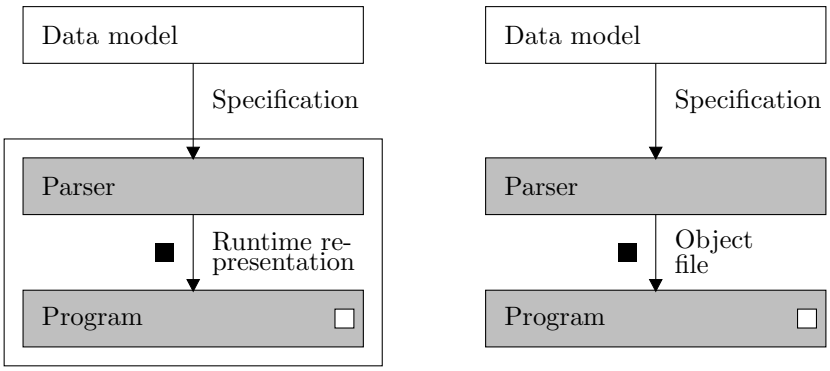


Figure 3.3: Configuration data lifecycle

or a similar data structure, according to the configured application. The byte order that is used depends on the target platform. At run time, the generated representation is accessible in the address space *AS* of the configured program.

3.2.1 Separating the Parser from the Application

Applications developed with respect to the traditional approach contain the complete configuration subsystem, as depicted in Figure 3.4(a). At startup, the parser converts the configuration data into a runtime-compatible representation, which is used by the running program. The conversion has finished before the generated data are accessed. More precisely, the parser creates a data object, and passes a reference to this object to the application. The data object could



(a) Parser linked to application

(b) Separated executable files

Figure 3.4: Deploying an object file

be generated by an external program to achieve the same result. We separate the parser from the application and execute it as a configuration utility. This program retrieves the specification from the configuration data base and stores the generated runtime representation to an object file, as in Figure 3.4(b). The starting application loads and links the object file. Therefore, this approach is called *configuration linking*. The application and the configuration utility can

run on different machines. The generated object file is deployed to the *target machine*, which executes the configured application. Since the transferred file is relocatable, the application can change the original address of the data used by the parser. Relocation is mandatory for dynamic reconfiguration as described in Section 3.3.

Seperating the parser from the functional implementation of the application and providing an independent configuration utility solves several problems stated in Section 3.1. The administrator can execute the configuration utility using his workstation, supervising the creation of the object file. Thus, errors that occur when the configuration data are parsed can be removed before deployment to live servers. Moreover, the application does not require the infrastructure for obtaining the data from the configuration data base, since the configuration utility fulfills this task. Values for parameters that require an additional look-up in an external data base are determined by the configuration utility as well. Thus, less software must be managed on the target machine. Furthermore, linking the generated object file to the application is a straightforward task. The object is loaded into memory and, if necessary, relocated. Afterward, symbols contained in the file are resolved. In contrast with the loader, parser code tends to be complex and long. Even in case of `thttpd`, which uses a simple configuration file, the parser code is about twice the size of the loader code of the extended version. The parser must be adapted to support changes of future versions of the application. In contrast, the loader is identical for multiple versions and applications.

3.2.2 Configuration Data Bases

Using configuration linking, one central data base can store the configuration data of multiple collaborating applications. An example is shown in Figure 3.5. When the configuration changes, the specification `spec` is parsed centrally, for instance on the administrator's workstation. The configuration utility generates the application-specific object files and deploys them to the target machines. Changing the applications to retrieve the configuration data from the central data base is not required, since the configuration utility containing the parser is a separate program. Storage of configuration data for several collaborating

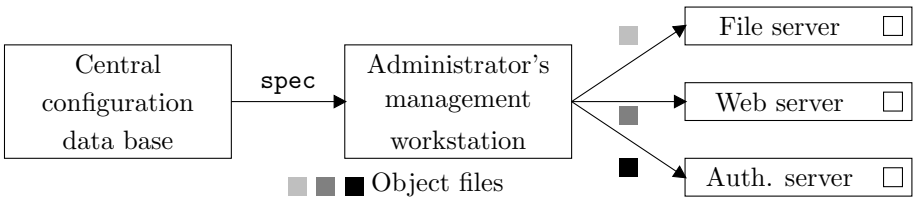


Figure 3.5: Using a central configuration data base

applications in one central data base has a lot of advantages. The configuration utility can derive the values of the attributes used to bind the components automatically. Other dependencies occur as well. In the example depicted in Figure 3.1, the web- and file servers running on the same machine must not use the same network port. If these attributes are stored in a central data base,

they can be checked independently of the target machine and application. The customer can design the configuration data base with regard to the local workflow. For example, the data base can provide fine-grained access control. Thus, the users may change only those attributes they require to perform their tasks. Additionally, a single management interface for all applications simplifies the administration task. The development of such an interface will be easier for the customer, if the data are stored using a unique format. Additional features such as local language support can be implemented. Finally, by providing an adjusted parser, the customer can integrate new programs seamlessly into existing environments and replace components without changing the management interface.

3.2.3 Generating the Object Files

For an application using configuration linking, the software vendor delivers the functional implementation and defines an interface for the deployment of runtime-compatible configuration objects. The customer designs the configuration data base and creates a parser, which generates objects complying with the interface of the application. For convenience, the vendor can provide a library that builds the object files from the data obtained by the parser. This library contains methods to set the parameters derived from the configuration data and performs the checks that are done at the startup of the application in the traditional approach. The vendor can specify the object file format, but the generation of binary object files is a cumbersome task and their interface is more specific to a particular version of the application than the interface of the library. Thus, using a library simplifies the development of the parser. Of course, a default implementation of the configuration utility, emulating the current handling of the configuration data, can be provided by the vendor. A configuration utility that uses the default data base generates the object file before the application is started.

3.3 Dynamic Reconfiguration

Availability of services is one major criterion for service providers. The service level agreements usually define a minimum rate of service uptime. The suspension of a subsystem for reconfiguration decreases the uptime, since client requests cannot be processed meanwhile. Therefore, server applications are desired to adapt to new configurations at run time.

3.3.1 A Classification of Configuration Parameters

A server application has to be reconfigured transparently for the clients. When a server receives a client request, state information representing the new connection is generated. After processing is finished, these data are deleted. The lifetime of the state information defines a *session*. A session can span multiple requests such as in web applications that preserve the state information across several network connections [5]. The server must use the same configuration throughout a session to achieve transparency for the clients. Each individual configuration specifies a consistent state of the application. The configurations

change at *epochs* with transitions between these consistent states. Each session is assigned to a particular epoch, which determines the related configuration. The client request initiating the start of a session and the reconfiguration defining an epoch are independent. Thus, a session does not depend on the particular epoch it is started in.

A configuration parameter can either be *local* to a session in a sense that its value may differ between several sessions, or *global* with the same value for all sessions. For a web server, the maximum age of pages for client-side caching is a local parameter, since separate sessions may use different values without interfering. In contrast, the maximum number of sessions handled concurrently by the server is unique in the application and therefore global.

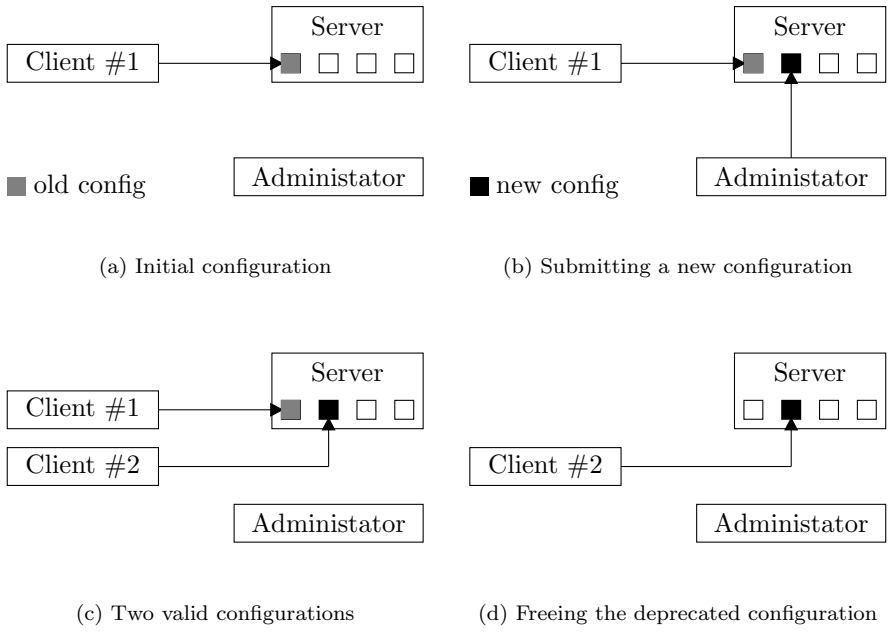
3.3.2 Local Parameters

Server applications can handle multiple client requests concurrently. The threads processing the requests read the configuration data. The reconfiguration of the application at run time, which marks a new epoch, results in changes to the same data. These accesses must be synchronized to preserve the consistency of the application. Since read accesses to the configuration data are expected to occur much more frequently than changes, the minimization of the synchronization effort for the readers is desired. Moreover, changes to local parameters must not affect sessions assigned to former epochs. The read-copy update approach [11] can be adapted to solve these issues.

The server application obtains an initial configuration object at startup. When a client request is received, the server generates state information representing the created session. Additionally, a pointer to the current configuration data is stored. The server accesses the configuration data by means of this reference when handling the session. Figure 3.6(a) depicts a server using the initial configuration to handle a client request. In Figure 3.6(b), the administrator deploys a new configuration object. The server application loads this object, marking a new configuration epoch. The configuration of the former epoch, which becomes deprecated, is still required by the first client. Thus, multiple versions of configuration data are valid. The reconfiguration does not affect clients started during former epochs, since the server refers to the deprecated configuration data when handling their requests. As shown in Figure 3.6(c), the server uses the new configuration object when handling sessions initiated after the reconfiguration. The deprecated configuration objects will be abandoned, but the server must assure that all sessions referring to this object are closed first. Therefore, the references used by the active sessions are checked. In Figure 3.6(d), the old configuration object is deleted after the first client closed its session. Using this technique, the synchronization effort for the readers may be neglected. The server stores a reference to the current configuration object in the state information when receiving a client request. Throughout the session, the thread reading the configuration data need not perform additional write accesses for synchronization. Moreover, the technique does not use waiting loops.

3.3.3 Global Parameters

The server cannot deal with global parameters as described in Section 3.3.2. These parameters represent dependencies of the sessions and have to be handled



(a) Initial configuration

(b) Submitting a new configuration

(c) Two valid configurations

(d) Freeing the deprecated configuration

Figure 3.6: Replacing local parameters

consistently for all sessions. In general, the clients either use the server for collaboration, such as a file service, or any interference between them must be avoided. A finer classification distinguishes between

1. parameters that are transparent for a client,
2. parameters that influence the communication among clients,
3. and parameters that are global because of the implementation.

The first class of parameters does not influence *what* is done from the perspective of a client, only *how* it is done. An example for this class is the maximum number of CGI[1] programs running concurrently on a web server. If the current number reached the upper bound, an additional request may be deferred. Alternatively, the server will send an exception that tells the client to repeat the request later, if this reply is valid according to the protocol. The client cannot determine whether the parameter has been changed or additional clients have connected to the server. Different policies can be used for replacing values of global parameters. The read-copy update technique is suitable where stale data can be either tolerated or suppressed [11]. A thread updates the data when a passing a *quiescent state*. Thus, different threads absorb the change independently. If this lazy handling of the configuration update is appropriate, the implementation can be done similar to the one described in Section 3.3.2. The creation and the deletion of a session are artificial quiescent states. Additional

[1] Common Gateway Interface

ones result in an earlier adoption of new configuration data. In the example, when the upper bound is decreased, the running programs are not affected, but the server defers the execution of additional ones until the current number is lower than the upper bound. If lazy handling is not appropriate, the threads that potentially access the configuration data will have to be at a consistency point [7] when the data are changed. Thus, there is no progress throughout the reconfiguration.

Several server applications, such as file servers enable clients to communicate. To achieve transparent reconfiguration, the server must assure that clients are able to communicate, even if they have been started in different configuration epochs. The data sent by a client are, at least temporarily, stored in the server. The representation of the data, for both transmission and deposition, may depend on the current configuration. In this case, the application is aware of the data format and converts the received data into a temporary form, which is stored. Data requested by a client will be converted into an output format and sent. The parameters specifying the input and output formats are local to the sessions and handled in the way described in Section 3.3.2. If the server-internal representation is changed, the existing data will have to be converted. Approaches to consistent transformation are described in [3, 6, 13].

Implementation details or the runtime environment of the application can enforce the uniqueness of a configuration parameter. Thus, only one configuration may be valid at a time. For example, the user id of a thttpd process running on UNIX is specified in the configuration file. Two sessions that are served concurrently may have different user ids without affecting consistency. But they are served by the same process, and the user id of a UNIX process is unique. Thus, the implementation of thttpd enforces that the user id is unique for all sessions. Therefore, when such a parameter is changed, a new component has to be created and to exist in parallel with to the old one. Nevertheless, requests must be handled by the appropriate component. The program abandons the old component as soon as all of its sessions are closed. Servers used for the communication among clients have to ensure that sessions started in different epochs are still able to collaborate. Therefore, persistent data have to be stored accessible to all components running concurrently.

Chapter 4

Evaluation

For evaluation, we extended the thttpd web server, version 2.25b [12], with an implementation of the techniques described in chapter 3. thttpd is a small and comparatively simple server, which supports the HTTP/1.1 protocol and runs on POSIX-compliant systems. Our extensions were developed and tested using GNU/Linux.

4.1 The thttpd Web Server

The thttpd web server has several features. URL-traffic-based throttling controls the transfer rates of files specified by using pattern matching on filenames. Moreover, status information about all requests can be written to a special logfile, and file transfers can be blocked depending on the referrer attribute. Finally, UNIX signals are used to initiate the re-opening of the connection logfile, to generate status log messages, and to terminate after all sessions that are active at the moment are closed.

thttpd is completely written in C. The build process, which uses GNU autoconf and make, creates one executable file containing the server application. Default values of the configuration attributes are specified in an include file. Most of the attributes can be overwritten by passing values as command line parameters. A configuration file, which is a simple text file that contains a key-value pair for each parameter, can be specified by means of a command line parameter. For the URL-traffic-based throttling, thttpd can obtain patterns for file name matching as well as minimum and maximum byte transfer rates from an additional file.

At startup, the application obtains the configuration data. The global variables containing these data remain unchanged at run time. After the configuration stage, the server initializes the environment. The connection logfile is opened, the root directory of the process is changed, and the network ports are requested. Finally, the process changes its user id. The program enters the main loop and checks the file descriptors repeatedly for incoming data and connections. thttpd maintains a connection table that contains the state information of all sessions. When a new connection is created, the server inserts a record into the connection table. At run time, except for running CGI programs and delivering directory listings, one thread serves all connections. The protocol

handling code for client requests is implemented in a library and hence separated from the server specific features. The methods provided by the library present models for servers and connections. An appropriate handle is passed at every function call. The requested files are read by using memory mappings. The library `mmc`, which is specific to `thttpd`, handles these mappings. Because of the single-threaded design of `thttpd`, accesses to data that have not been mapped yet block the whole application. Thus, the server performs polling before sending data to a client. The library `fdwatch` contains the corresponding code. After opening a file, the program registers the file descriptor at the library. On each pass, the main loop calls a method that performs polling on the registered files. The build process determines the particular system call, which depends on the operating system. Information about available data can be retrieved by using library calls. The UNIX alarm signal is used as a watchdog to detect application failures. A proprietary timer library supports scheduling of periodical tasks such as updating the throttling table and resetting the watchdog flag.

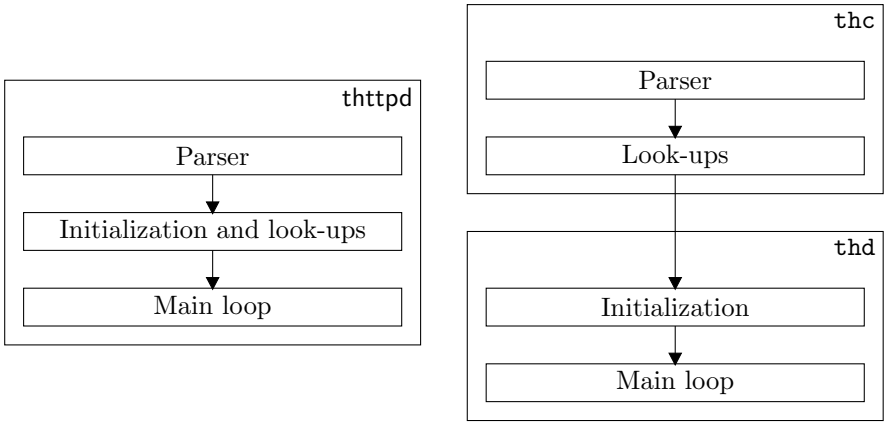
The server obtains the patterns used to match filenames and the minimum and maximum transfer rates for throttling at startup. These values are stored in the throttling table, along with current statistics, such as the average transfer rate. A client request, which initiates a new session, is analyzed. The server compares the requested filename with the patterns in the throttling table. For each matching pattern, a reference to the corresponding row of the throttling table is stored in the state information of the session in the connection table. Before sending data to a client, `thttpd` checks these entries and adjusts the transfer rate to the value specified in the throttling table.

4.2 The Extended Version of thttpd

Using the techniques described in chapter 3, we enhanced `thttpd` to use runtime-compatible object files for the deployment of configuration data. Adaption to configuration changes at run time without interrupting services is supported as well. Figure 4.1 depicts the tasks of the particular applications. A seperate utility called `thc`, the thttpd configuration utility, contains the parser code, along with code for checks on the attributes and look-ups for several parameters. `thc` obtains the configuration data and generates a relocatable object file containing the data. `thd`, the thttpd with dynamic reconfiguration, reads this object file at startup. During the server loop, `thd` listens on a named pipe for objects that replace old configurations.

4.2.1 Object Files Containing Configuration Data

An object file contains the data that are passed from the configuration utility to the server application. The reference implementation uses the simple a.out OMAGIC [10] format, which supports relocation and symbols. Since UNIX has used the a.out format for many years [4], there are many programs that handle a.out files and simplify debugging. The object file contains the runtime representation of the attribute values obtained from the configuration data base. For some attributes, look-ups have been performed, and the results are included. In particular, `thc` derives the network addresses from the host name and infers the user and group ids from the user name. Additionally, a throttling table, if



(a) Structure of thttpd (b) The extended version

Figure 4.1: Splitting thttpd into two applications

Symbol Name	Contents of Section
config	Local parameters
globalc	Global parameters
th_b	Throttling table
throttles	Reference to th_b
numthrottles	Number of values in th_b

Table 4.1: Contents of the configuration object generated by thc

specified, will be inserted into the object file. Table 4.1 lists the five symbols contained in the file and the sections they refer to.

4.2.2 The Configuration Utility

The relocatable object files containing the configuration data are created by thc. The command line arguments and the configuration files are identical with those of thttpd. Figure 4.2 depicts the design of thc. The parser reads the configuration file and passes the attribute values to libthc. This library generates the runtime representation RR of the configuration data. The cspace module finally creates the a.out file. In the following, the modules will be explained in detail.

The cspace module provides functions to create relocatable a.out object files. cspace is not specific to thc and can be used to generate arbitrary a.out files that contain only data. Figure 4.3 shows the structure of the objects managed by the module. In the beginning, a *configuration object* must be initialized to create the necessary data structures. *Symbols*, which correspond to symbols in the object file, can be added to and deleted from the configuration object.

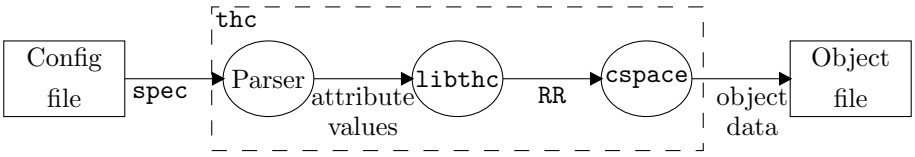


Figure 4.2: The structure of `thc`

Allocation of memory is done using function calls that are similar to those of C. Each of the allocated *chunks* is assigned to one symbol and consists of a contiguous part of memory. Multiple chunks can be assigned to the same symbol. The configuration data reside in this memory. Nevertheless, pointers must be stored relocatably in the object file. Since the library cannot determine the types of the variables, the application must mark the memory locations that contain pointers by calling a particular method. After the data are completely stored to memory, `cspace` is advised to write the object to a file. For each pointer, a relocation record is stored in the object file. This record depends on two symbols: The one containing the pointer and the one containing the target of the pointer. `cspace` determines these symbols automatically when relocating the object. The *text* and *text relocation* sections of the generated object file are empty.

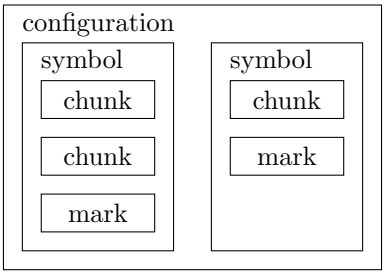


Figure 4.3: Objects used in `cspace`

The mapping of the configuration data of `thttpd` to the generic concepts of `cspace` is the task of the `libthc` module. This library does not depend on a specific configuration data base and implements the concepts described in Section 3.2.3. Thus, the implementation of a custom parser can be based on `libthc`. For each configuration parameter of `thttpd`, the library provides a specific method for setting the desired value. `libthc` does not perform look-ups such as the conversion from the user name to the user id, since this mapping depends on the particular configuration data base. Therefore, `libthc` requires the configuration utility to provide the resolved value for parameters that necessitate look-ups. Using the default `thttpd` configuration file, the user name and the host name have to be transformed.

The `thc` application parses the configuration data and obtains the attribute values. In contrast to `thttpd`, `thd` does not store the values to global variables

but writes them to the configuration object by means of the `libthc` module. We adapted the parser code of `thttpd`. Thus, for the look-ups that are performed additionally, `thc` uses the same data bases as plain `thttpd`. In particular, the user name and the host name are transformed by means of the regular UNIX functions. Thus, the current implementation of `thc` must run on a host that uses the same data for these mappings as the target machine. Exchanging the configuration data base requires to provide a custom implementation of the parser. Thus, the code of `thc` has to be adapted, but there is no need to change `libthc` and `cspace`.

4.2.3 Statically Binding Configuration Data

For debugging purposes, there is a slightly modified version of `thttpd` called `ths`, which complies with the interface of the objects generated by `thc`. The object file is statically linked to the program during the build process. Therefore, the replacement of the configuration requires to relink the application. The parser code is removed from `ths`, resulting in a smaller executable file.

4.2.4 Dynamic Reconfiguration

`thd` is a version of `thttpd` that implements dynamic reconfiguration and accepts the object files generated by `thc`. The administrator deploys the configuration data by writing the object file to a named pipe placed in the local file system. Because of security limitations of the UNIX runtime environment, `thd` must consist of two separate processes. One part, the `thd` server, handles the requests sent by the clients. The other part, the `thd` configurator, manages submission of configuration data. An unnamed pipe connects both processes. The structure of `thd` is depicted in Figure 4.4. The configurator, which does not handle client requests, has root privileges. Otherwise, the coarse-grained UNIX rights management prohibits reconfigurations that change the user id of the server process. At the startup of the `thd` application, the configurator obtains an initial configuration object and opens the named pipe. Afterward, a `thd` server is created using the initial configuration object. When `thd` receives new configuration data, the configurator checks whether the running `thd` server can adopt the changes. In this case, the configuration object will be passed through the unnamed pipe. The object must be relocatable, because the `thd` server eventually loads the data to a different address.

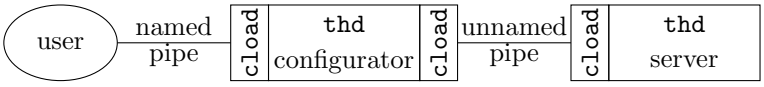


Figure 4.4: The structure of `thd`

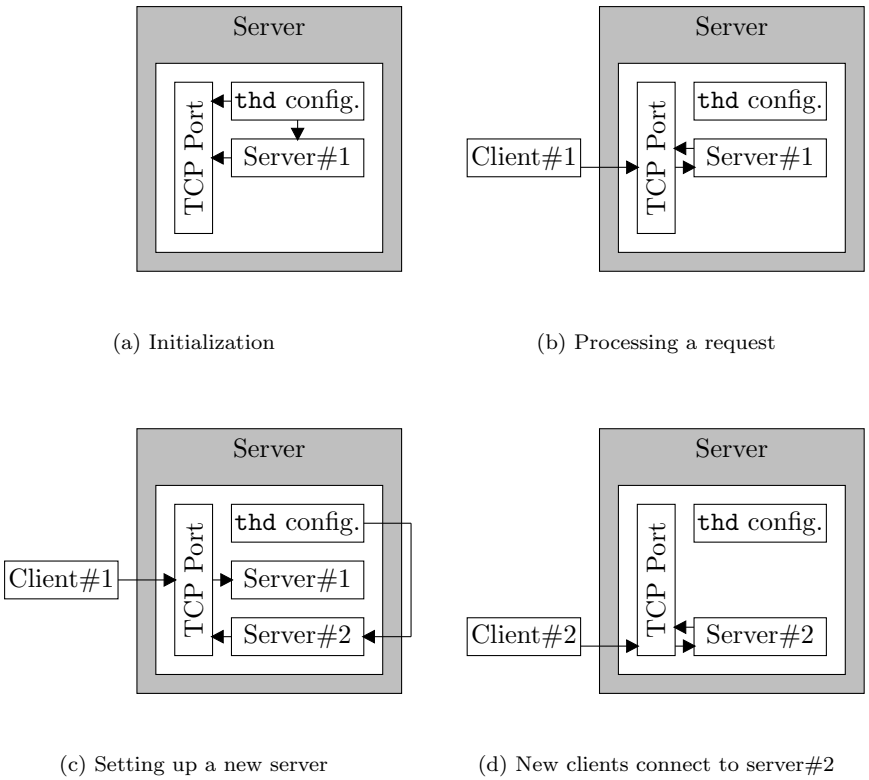
The `thd` server changes local parameters by the technique described in Section 3.3.2. Therefore, the connection table, which was mentioned in Section 4.1, contains an additional column containing a reference to the local configuration data. When a connection is initiated and registered in the connection table, the address of the current local configuration data is stored additionally. The

server determines the configuration to be referred to by using the connection table when handling a request. When receiving a new configuration object, the server stores the address of the local configuration data, which is obtained by resolving the symbol `config`, globally to mark the new configuration as the current one. Sessions started afterward will refer to these data. In the main loop, `thd` checks whether multiple versions of the configuration data exist. If there are old data and there are no active sessions that refer to these data, the deprecated configuration object will be released. Several parts of `thttpd` have been changed to support dynamic reconfiguration. For instance, the original implementation requires the served documents to reside in subdirectories of the current working directory. Since the parameter determining the location of the documents is local in `thd`, `libthttpd` has been adjusted to infer the document directory from the local configuration of the session.

As previously mentioned, the maximum number of CGI programs running concurrently is a global parameter. `thd` applies the lazy handling described in Section 3.3.3 when this parameter is changed. In particular, this attribute is only referred to before starting a CGI program. The server checks whether the number of running programs is lower than the upper bound. Thus, if the maximum number is decreased, running programs will not be affected. Execution of additional programs is blocked until the current number is adequate. In contrast, an eager replacement technique is applied to updates of the throttling table. The connection table, which contains references to the throttling table, has to be adjusted to preserve consistency, since the indices of the throttling table may have changed. Because of the single-threaded design of `thttpd`, no additional synchronization is needed.

Several additional configuration parameters are global because of the UNIX environment. For example, the user id is unique for a process. Because of the single-threaded implementation, the id is unique for all sessions in the case of `thttpd`. The values of these parameters are updated as follows. Figure 4.5(a) depicts the initialization phase of `thd`. The configurator obtains the initial configuration object, opens the TCP port, and starts a server instance, which uses the initial configuration data. The server, which is created by the `fork` system call, inherits the file descriptor of the TCP port. In Figure 4.5(b), a client initiates a request, which is handled by the `thd` server. When a configuration that changes the user id is submitted, the configurator signals the current server to close the file descriptors of the network ports and to exit gracefully. Additionally, a new instance of the server is created, which is shown in Figure 4.5(c). When the new instance is started, the current configuration, which was loaded and relocated by the configurator, is present in the address space. Thus, the new instance uses the new user id. The old instance exits after all sessions have been closed, such as in Figure 4.5(d).

Nevertheless, the server must not lose any requests during reconfiguration. Therefore, the `thd` configurator allocates the network ports. The servers are created by using the `fork` system call and inherit the file descriptors. Since the ports are never closed throughout the runtime of the configurator, no requests are lost. At all times, at most one server accepts new connections. This handling is appropriate, since, for `thttpd` and `thd`, each session consists of exactly one network connection. After the configurator has obtained a new configuration object, relocation is applied. Before the data are passed to the current server instance, the attributes can be adjusted. For example, if the server uses a



(a) Initialization

(b) Processing a request

(c) Setting up a new server

(d) New clients connect to server#2

Figure 4.5: Changing the user id

different root directory, the path of the connection logfile is adapted. If the changed object is transmitted by means of the unnamed pipe, the object must be converted to a valid a.out file again.

The `cload` module contains an object file loader for the a.out file format. A loaded file is completely stored in the memory and can be relocated to an arbitrary address. The addresses of pointers are either absolute or relative to the symbol that the related data are assigned to. The appropriate mode can be selected by passing a parameter to the relocation method. This option is necessary for passing a valid a.out file from the configurator to the server. Furthermore, `cload` provides methods that resolve the name of a symbol into the address of the data referred to. Thus, the configuration data contained in the object file can be composed of sections, which are addressed independently.

UNIX signals initiate several activities of `thttpd` [12]. In the case of `thd`, using the process id of the configurator is convenient for the user. The configurator must relay certain signals to the `thd` server. For instance, the application can be instructed to re-open the connection logfile. Since this logfile is maintained by the `thd` server, the configurator forwards the signal. As previously mentioned, the configurator adapts the path of the logfile to the root directory of the `thd` server. Nevertheless, if this path is not visible to the server, re-opening the logfile

will be impossible. Thus, the configurator will spawn a new server instance, which opens the logfile before changing the root directory. The configurator signals the old server to exit after all sessions have been closed.

Chapter 5

Conclusion and Future Work

This thesis investigated several ideas toward simplification of server configuration management. This last chapter summarizes the contents dealt with in the previous sections. Moreover, a survey of the future work is given.

5.1 Conclusion

Management of server configuration data is a complicated task. In practice, a small group of administrators manages many machines, and each of the machines runs multiple programs. Server applications collaborate with other services and therefore depend on the corresponding configuration data. After all, service uptime must be maximized. By using a central data base, values of dependent parameters can be derived and attributes can be checked for consistency automatically. If the configuration subsystems are separated from the applications, the parsers will be able to run supervisedly on an administrator's machine. Therefore, errors in the configuration data can be eliminated without decreasing the availability of the servers. The application uptime can be increased by reconfiguration at run time, which particularly depends on error-free configuration data. An approach to dynamic reconfiguration without suspending the application is presented in this thesis.

Extending the thttpd web server, we implemented the techniques described in this thesis. The enhancement of the application to deploy and accept runtime-compatible object files has required only slight changes of the server code. We expect other server software to be adaptable as well. Adjusting the configuration utility to another kind of data base can be done by exchanging the parser code. The application can remain unchanged. Dynamic reconfiguration has been implemented as well, and, for thttpd, all attributes specified in the standard configuration file can be replaced at run time without interrupting services. Nevertheless, there is no persistent state management in thttpd, so reconfiguration can be performed easier. Parameters that must be unique, such as the user id, and the coarse-grained rights management of UNIX require one process to have root privileges.

5.2 Future Work

This thesis describes the dynamic update of configuration data for one application. Composite systems must be reconfigured consistently for all components. Therefore, the servers must synchronize their configuration epochs. A client request may trigger subsequent requests to other servers. All of these requests can be assigned to the same session and thus belong to the same configuration epoch. Each of the servers contains multiple valid configurations, but only one configuration is regarded current. The insertion and the deletion of configurations, as well as the substitution of the current configuration, could be regarded as transactions. A technique based on the two-phase-commit protocol could be used for distributed transactions. By handling the insertion of new configuration data and the substitution of the current configuration as distributed transactions, the servers would only regard a configuration current after all servers have adopted the new version. Moreover, the servers would not release a configuration until all sessions that were started in the corresponding epoch have finished. Thus, the consistency could be preserved globally throughout a session.

The approach proposed in [7] specifies that the components receive reconfiguration scripts and interpret them locally. Reconfiguration must be synchronized with the internal state of components, which is exported by means of state predicates. A reconfiguration script depends on the values of certain predicates, which indicate a consistent state of the component. Throughout the execution of the reconfiguration script, the component cannot proceed. The scripts are more general than the configuration objects regarded in this thesis. The read-copy update technique described in Section 3.3 could be applied to reconfiguration scripts as well. The effects on the composite system have to be analyzed.

In Lira [6], a host agent runs on each machine. The manager installs and removes components on the host by means of this agent. The approach proposed in this thesis could be extended by the concept of the host agent. When a new configuration is deployed, the host agents would check for components that are not installed yet. The corresponding applications would be retrieved from a code repository and configured.

The current approach cannot easily be mapped to an arbitrary kind of component. Uniqueness of parameters interferes with the approach of regarding multiple configurations as valid. For instance, parameters affecting devices attached to the machine must be handled differently. Furthermore, this thesis focuses on exchanging configuration data. However, replacement of code at run time to adopt patches without restarting a program helps to prevent downtime. The approach from [3] could therefore be applied to server applications.

Bibliography

[1] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Scheduling and Simulation: How to Upgrade Distributed Systems. In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems*. USENIX Association, 2003.

[2] Paul Anderson. Towards a High-Level Machine Configuration System. In *Proceedings of the Eighth Systems Administration Conference*, pages 19–26. USENIX Association, 1994.

[3] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewsky, and Jeremy Kerr. Providing Dynamic Update in an Operating System. In *Proceedings of USENIX'05*. USENIX Association, 2005.

[4] Bell Telephone Laboratories. *UNIX Programmer's Manual*, seventh edition, 1979.

[5] Ed Burns and Justyna Horwat. *Introduction to JavaServer Faces*, 2004.

[6] Marco Castaldi, Antonio Carzaniga, Paola Inverardi, and Alexander L. Wolf. A Lightweight Infrastructure for Reconfiguring Applications. In *Lecture Notes in Computer Science*, volume 2649/2003, pages 231–244. Springer Verlag, 2003.

[7] Markus Endler. Support for Consistency-preserving Dynamic Reconfigurations in Distributed Systems. In *Proceedings of the Third Workshop on Future Trends of Distributed Computing Systems*, pages 185–191. IEEE, 1992.

[8] Tassilo Erlewein, Achim Frank, and Martin Konold. *Kolab Server: Technical Description*, 2003.

[9] Jeff Kramer. Configuration Programming—A Framework for the Development of Distributable Systems. In *Proceedings of the International Conference on Computer Systems and Software Engineering*, pages 374–384. IEEE, 1990.

[10] John R. Levine. *Linkers and Loaders*. Morgan-Kauffmann, 1999.

[11] Paul E. McKenney and John D. Slingwine. Read-Copy Update—Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, pages 509–518. IASTED, 1998.

[12] Jef Poskanzer. *thttpd Man Page*, 2000.

[13] Ian Warren and Ian Sommerville. A Model for Dynamic Configuration which Preserves Application Integrity. In *Proceedings of the Fifth European Software Engineering Conference*, pages 81–88. IEEE, 1996.