DIPLOMARBEIT

A C++ Servlet Environment

ausgeführt am
Institut für Informationssysteme (E184-1)
Abteilung für Verteilte Systeme
der Technischen Universität Wien

unter der Anleitung von o. Univ.-Prof. Dipl.-Ing. Dr.techn. Mehdi Jazayeri
und Univ.-Ass. Dipl.-Ing. Dr.techn. Thomas Gschwind

durch

Benjamin A. Schmit
Österreich

Matrikelnummer: 9626231

Wien, im September 2003

Zusammenfassung

Die Entwicklung von Webapplikationen konzentriert sich hauptsächlich auf Java und einige Skriptsprachen. Die für Systemprogrammierung gerne eingesetzten Programmiersprachen C und C++ werden dafür kaum noch verwendet. Diese Arbeit präsentiert eine C++ Servletumgebung (C++ Servlet Environment; CSE), welche hohe Leistung speziell für C++ Programmierer bietet. Ein großes Problem hierbei ist die Erreichung dieser hohen Leistung bei gleichzeitiger hoher Stabilität, so daß einzelne Servlets den Server nicht zum Abstürzen bringen können. Wir stellen das Design sowie Details zur Implementierung dieses Systems vor.

Weiters haben wir eine C++ Servletprogrammierschnittstelle und eine Sprache für C++ Server Pages entworfen. Beide wurden ähnlich zu ihren Java-Äquivalenten erstellt, um Umsteigern den Einstieg zu erleichtern. Wir stellen in dieser Diplomarbeit auch den "Record Store" vor, ein umfassendes Beispiel zur Webapplikationsentwicklung mit der CSE, in welchem typische Techniken und Abläufe für Programmierer dargestellt werden. Wir präsentieren eine Benchmark-Suite und vergleichen die Leistung des CSE mit der anderer Webanwendungsumgebungen.

Abstract

Current environments for web application development focus on Java or scripting languages. Developers that want to or have to use C or C++ are left behind with little options. To solve this problem, we have developed a C++ Servlet Environment (CSE) that provides a high performance servlet engine for C++. One of the biggest challenges we have faced while developing this environment was to come up with an architecture that provides high performance while not allowing a single servlet to crash the whole servlet environment, a serious risk with C++ application development. In this thesis we explain the requirements for such a servlet environment, the architecture we have designed, as well as details about the implementation of the CSE.

To allow developers to get familiar with the CSE, we have also designed a C++ servlet API and a syntax for C++ Server Pages that closely resembles that used by Java servlet environments. To illustrate the use of the CSE, we have also implemented the Record Store, a sample web application. On the basis of this example, we describe how servlets can be developed using our environment. To demonstrate the benefits that can be gained by using the CSE, we evaluate its performance and compare it to that of other popular servlet environments.

Acknowledgements

I would like to thank the developers of the Apache HTTP Server [Apaa] and the MySQL database [WA02] for their efforts on which I could build the C++ Servlet Environment. Especially, I thank them for their decision to release their products as free software [Deb].

I thank the contributors to the CSE user forum for their valuable suggestions and reports. I also wish to thank the supervisors of this thesis, Thomas Gschwind and Mehdi Jazayeri.

I gratefully acknowledge the financial support provided by the European Union as part of the EASYCOMP project (IST-1999-14191) and our partners from the University of Kaiserslautern. I would also like to thank the USENIX Advanced Computing Systems Association for giving me a grant to attend the USENIX Annual Technical Conference.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Originally, the web has been used as a medium for presenting static text and images to a wide variety of users. Dynamic content was provided through the Common Gateway Interface mechanism (usually referred to as CGI). It offers a way to incorporate executable or interpretable files into the web server. The program or script is processed on demand, parses some input parameters, and writes its response into a stream which is returned to the web browser. CGI scripts became more and more widely used, and some common principles used in most of them were identified. The most important common task of such scripts is session management. Since HTTP is a stateless protocol [FGM$^+$99], a way had to be found by which a client could be traced over several requests. The CGI does not provide this functionality, so that every servlet programmer had to re-implement it.

Servlet environments solve this problem by automating such common tasks. A servlet server provides a uniform interface to its servlets. It usually offers much better performance than the CGI because objects used by a servlet can be kept in memory between requests. The environment commonly comprises a web server which provides static documents, a servlet server which executes dynamic code, and a database server which stores session and application data.

The comfort a servlet environment offers on the other hand implies that all server applications adhere to a pre-defined standard, which also includes the choice of the programming language. Traditionally, this programming language has been Java for most servlet environments. The Java Servlet API [Sun01a] has become a standard for servlet programming.

An advantage of Java is that it is platform-independent. Its main disadvantage, however, is that Java code is not compiled directly into machine code, but into an intermediate language which either has to be interpreted at run-time or has to be compiled on-the-fly. Both approaches usually imply lower performance than native code; a just-in-time compiler needs to compile the code quickly and thus has limited time in which to optimize the code for the current platform.

Another disadvantage is that Java code cannot use C code efficiently. This makes it especially cumbersome to use existing C code or the platform's standard libraries. Therefore, we decided to create a servlet environment written entirely in C/C++. This system is aimed at overcoming the difficulties mentioned above. It does not rely on a virtual machine or a just-in-time compiler, but is executed directly on the destination platform. It allows the inclusion of previously written C/C++ code easily and thus helps porting existing code to the web.

The C++ Servlet Environment provides everything that is necessary for servlet execution. The API it offers has been designed similar to the Java Servlet API [Sun01a]. We also have avoided to make it overly complicated by trying to re-implement the underlying Java classes. Also, our servlet environment offers a language similar to Java Server Pages [Sun01b]. Such an easy-to-use method of integrating dynamic functionality into previously static documents is important because it greatly eases development for servlet programmers. It also helps beginners by allowing step-by-step learning. These documents are processed by the servlet server requiring little configuration.

The system performs well in comparison to current servlet environments. Even though C++ is used, it is equally secure as Java systems. To gain the same amount of stability, in C++ a new architecture was necessary. We have taken care to provide ease of use as well, else most programmers would avoid it. Finally, the CSE offers enough parallelism to be able to compete with the performance of other servlet environments.

In Chapter 2 we will identify and discuss the main requirements for a C++ Servlet Environment. The CSE's architecture will be described, together with some design decisions that we have made, in Chapter 3. Chapter 4 is concerned with the implementation of this architecture, its components and its interfaces.

Chapter 5 gives an introduction on servlet programming and explains both the servlet and the CSP interface (C++ Server Pages) and discusses the internal mechanisms used to transform a CSP into a servlet. Chapter 6 presents a comprehensive example which explains the practical aspects of servlet programming by describing and listing the servlets needed for a web store.

Chapter 7 presents an overview on related work and explains their differences. The performance of some concrete servlet environments is evaluated in Chapter 8. Finally we draw our conclusions in Chapter 9.

Chapter 2

Requirements

This chapter addresses the requirements for a servlet environment. It also presents the advantages of C++ as compared to typical implementations in Java, which poses additional difficulties not encountered in other servlet environments. In this chapter, we limit our discussions to identifying possible solutions. Their details and the solutions we have chosen will be presented in Chapters 3 and 4.

During the design of the C++ Servlet Environment, we identified five main requirements for a servlet environment:

Security: An important topic of interest for all Internet applications. Every day new security holes are found that could have been avoided with a more thorough design.

Stability: Special mechanisms are required to guarantee the system's stability because of the large-scale utilization of user-supplied code (the servlets).

Ease of Use: A servlet environment is worthless if it is too hard to administrate. We do not just want to present a theoretical concept which is complicated to use in practice. Hence, setup and maintenance should not be harder than that of other successful servlet environments.

Parallelism: On a server machine, parallelism is a necessary prerequisite for both performance and scalability. The upper bound of the access time to a service can be high and thus forbids handling requests in serial.

Performance: For Internet services, more performance can be gained either by writing more effective software, or by buying more (usually expensive) hardware. Therefore, implementing more performant algorithms may save a lot of expenses later.

2.1 Security

Many existing servlet environments as well as other network services try to reach the goal of a more secure system by introducing additional security mechanisms. The Java virtual machine in particular provides a standardized run-time system, the sandbox, in which all Java code is executed [LY99]. Security permissions are required in order to interact with system components outside the sandbox, and these permissions must be set up independently of the underlying operating system (which is one of the reasons why Java programs run on so many platforms). The C++ Servlet Environment, on the other hand, was written explicitly for UNIX operating systems and reuses many of their security features, which eases the CSE's administration.

Within the CSE, servlets are executed within separate tasks called sandboxes (a concept different from the Java sandboxes mentioned above). Since they are normal UNIX tasks, they can be executed with different user IDs in order to minimize the impact of servlet programming errors, they can be restricted to only a part of the UNIX filesystem using `chroot`, and they can make use of more secure memory allocation libraries — in short, the whole range of the UNIX security tools is available to them.

The disadvantage of this approach, and also of choosing C++, is low portability. The CSE depends on the C++ programming language, the C++ Standard Template Library, and the UNIX libraries. However, because good C++ compilers are available on almost all platforms and even the UNIX libraries are provided by more and more systems, this problem has ceased to be an insurmountable obstacle.

For most UNIX operating systems, as well as for the web server and the default database server used by the CSE, security patches are usually provided shortly after weaknesses become known. The active maintenance of these components also helps to increase the CSE's security: Even a completely secure servlet server won't help much if the generated HTML output can be changed by an intruder due to an insecure web server, or if the stored data can be altered on an insecure database server.

The choice of C++ as a programming language also helps to eliminate one of the most common security flaws: buffer overflows. The servlet server uses the C++ Standard Template Library's container types (including the `string` type) [Str00]. These errors should therefore only be able to happen at the interface to the web server (which is written in plain C, but only contains a small number of lines of code) or within the STL itself. The STL, on the other hand, is used by a large number of programs, so that it is constantly being tested by many different programmers, and security problems should be patched quickly.

Additional security can be gained by applying firewalls and intrusion detection systems. A correctly configured firewall enhances the security of the servlet server by denying any direct requests to its port, so that all attacks first have to pass the web server. Since the servlet server does not use any authentication mechanisms on its port, this configuration is suggested for production systems. An intrusion detection system may be able to warn the administrator of known attack methods.

2.2 Stability

A servlet environment can never be sure about the stability of the code it has to load for executing servlets. Since the C++ Servlet Environment has to load such code on a large scale, the old concept of loading the code into the server itself needs to be replaced in order to provide sufficient stability.

The concept of the sandboxes is able to manage this: The main server task never touches a servlet and its stability can therefore never be compromised by an unstable servlet. It delegates this task to the sandboxes, separate parts of the server which do not store any vital information. A dedicated mechanism restarts the sandboxes if a servlet crashes them. The crash is logged so that the servlet programmer has some guidance in fixing the error.

This concept is also able to maximize the stability of a web application: Servlets should be placed first into an "unstable" sandbox and moved into the "stable" sandbox only after some time of testing.

2.3 Ease of Use

A servlet environment that requires reading a thick manual before it can be used may be helpful for some special applications; however, most users probably prefer a system where they can start deploying simple servlets quickly, and ignore the more advanced features until they are needed. The CSE has been designed to make the transition from another servlet environment (or from a static web server) as easy as possible.

According to [Net03], the most widely used web server today is the Apache HTTP Server, which is also used by the CSE. A site administrator that already uses this server only has to add a module in order to incorporate it within the CSE; the server configuration and the static documents provided can serve as a basis for a dynamic web site. HTML pages can be changed to C++ Server Pages by setting their extension to "`.csp`" and adding CSP tags as needed. At this point, the servlet programmer does not need to know about the internal workings of the rest of the servlet environment.

Both the servlet interface and the C++ Server Pages tags have been designed similar to their well-established Java equivalents. A Java servlet programmer should be able to understand the purpose of a C++ Server Page without much help. Java differs from C++ in some ways, which means that Java servlets still need to be rewritten in order to work with the CSE, but Java Server Pages are easier to transform to C++ than servlets — there are even some JSPs that will run without any changes.

Ease of use has also been implemented on other levels: The servlet server configuration has been designed so that it can be changed while the server is running, and the changes trigger a dynamic re-configuration without the need to shut down any part of the system. Servlets, C++ Server Pages, and even sandboxes can be added, removed, and changed on-the-fly. Also, servlet parameters like the persistence level used for session-dependent data can be changed by altering only a single line in the servlet source code.

2.4 Parallelism and Performance

A server that just processes all requests serially scales badly, whereas the ideal scalable server would handle all requests in parallel, with no synchronization whatsoever. The latter approach, however, is often impossible in practice. During implementation, we focused on finding a good compromise between the two extremes, introducing parallelism on many levels.

First, all requests coming in through the socket are accepted serially. After determining where a request should be handled, it is quickly passed on to one of the sandboxes. There can be several sandboxes which run in parallel, so that the server scales well with the number of sandboxes (or, in other words, the number of web applications on the server). Between the sandboxes, there is no need for synchronization whatsoever, which means that an ideal form of scalability is accomplished here.

However, the requests which go into a single sandbox are (currently) processed one at a time. If they were processed in parallel, many kinds of accesses would need to be synchronized: Reading a request from a socket, accessing a session, using the database, and writing log messages. Some performance can probably be gained by parallelizing sandbox requests, but there is a natural limit to the number of parallel threads in a single sandbox if performance is to be maximized.

Also, a sandbox request is usually processed without the need of waiting for external shared resources. On a single processor system, several such threads cannot be executed faster through (pseudo-) parallelization, which means that only a true multi-processor system would reap some benefits here. On the other hand, several distinct sandboxes have no need for synchronization between them, which means that it is much better to assign a sandbox task to each CPU than to split up the sandboxes into smaller units.

If distinct sandboxes are not an option, a possible solution towards more parallelism would be the replication of the whole server host (with web server, servlet server, and, if necessary, database server) along with a mechanism that redirects requests according to some property of the request source, such as the session ID or the client's IP address. It needs to be a property of the request source because all requests coming from the same source might require the same session, and therefore need to be handled by the same sandbox. This approach, however, is not yet supported with the C++ Servlet Environment.

Chapter 3

Architecture and Design

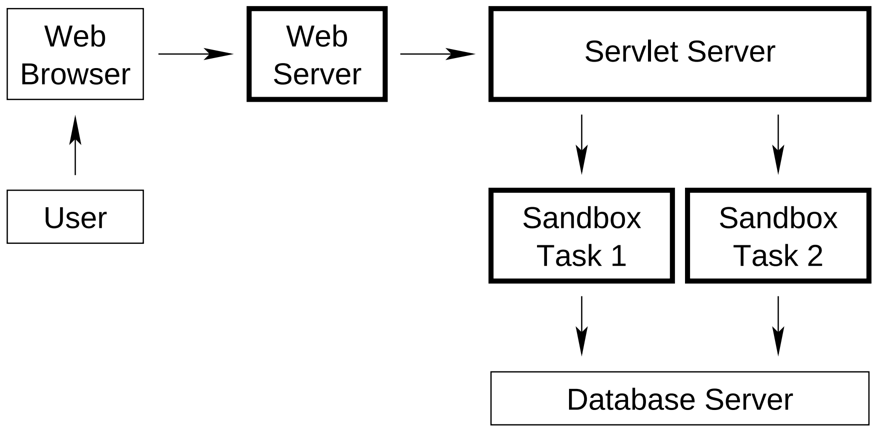

Figure 3.1: Design of the C++ Servlet Environment

The C++ Servlet Environment consists of a web server, a servlet server, and several sandboxes that execute the servlets. Figure 3.1 shows an overview of these parts, their environment and their connections (the arrows here indicate the direction of a request). Additional figures showing the internal organization of the web server, the servlet server, and the sandboxes are available in Chapter 4. A briefer introduction to the CSE can be found in [GS03].

The core components each have their own internal structure. They run as separate tasks, and there are well-defined interfaces between them.

The web server module receives requests for pages from the web server and forwards them to the servlet server using a socket. The servlet server keeps track on pages and their association with servlets or C++ Server Pages. If a URL has not been registered, it returns the failure to the web server, which then tries to fulfill the request by other means, e.g. by delivering a static page to the client.

When the servlet server receives a registered URL, it forwards the request to the servlet's sandbox using a pair of sockets. The sandbox processes the request, optionally contacting a database server. The response from the sandbox is then returned to the browser directly.

The sandboxes are tasks that hold a number of servlets. They are separated from each other so that a servlet that crashes can only bring down its own sandbox. Hence, servlets running in other sandboxes are not affected by the crash. The sandboxes also manage sessions, and they each have their own garbage collector.

3.1 The Web Server

The web server processes each request it receives from a web browser. It serves static pages by itself and contacts the servlet server only when a dynamic page is requested. The alternatives we had for this part of the servlet environment were to either reuse an existing web server or to write a new one. A new web server would have had the advantage of a close integration with the servlet server, eliminating the overhead of communication between those servers. On the other hand, an existing web server reduces the complexity of the implementation — and it supports more options than we would have implemented otherwise.

There are already many web servers in existence that meet the requirements for a servlet environment. We decided to take a web server that is both used widely and that supports loading third-party modules into the server. The most widely used web server [Net03] currently is the Apache HTTP Server [Apaa]. It supports modules that can be loaded by listing them in the server's configuration file — a process that the web server administrators are already used to. The C++ Servlet Environment uses the module interface of Apache, version 2.0, but it should not be hard to write similar support modules for other web servers.

Since the Apache HTTP Server is written in plain C, the interface between the web server and the servlet server must translate between Apache's C structures and the C++ Servlet Environment's C++ objects. This can be done without an additional overhead because both requests and responses need to be serialized when they are passed between the web server and the servlet server. Therefore, the module has two parts:

The C subsystem implements the Apache HTTP Server's module interface, which requires plain C code. During module initialization, it processes the servlet server's basic configuration (the part within the web server's configuration file, `httpd.conf`), and registers functions that will be called by the web server.

The C++ subsystem (actually C++ code with C bindings) is responsible for communication with the servlet server. It sends the requests to the server through a socket. Then, it returns the servlet's answer as a response structure to the web server (or, if there was an error, it declines the request, which causes the web server to deliver a static page or a *not found* error).

3.2 The Servlet Server

The servlet server consists of the coordination server, the servlet registry and the CSP manager. These three classes are coupled tightly, hence, they call members of the other two classes. All of them are static. They manage the registered servlets and CSPs, the sandboxes in which these are stored, and the interface to the web server module. The servlet server uses several threads for its operation: The main thread (with the coordination server's main loop), the guardian thread which monitors the sandboxes, and a number of compiler threads that can be generated on demand.

The architecture of the C++ Servlet Environment supports multiple sandboxes to increase both stability and scalability. The servlet server receives requests and assigns them to the individual sandboxes. This main loop must be small enough so that the servlet server does not become a central bottleneck. If it contained too much code, it would hinder scalability because the sandboxes can execute servlets only after the requests have been dispatched by the main loop. The requests are therefore forwarded as quickly as possible in order to avoid these inefficiencies. In the future, we plan to parallelize this loop so that it can be executed by several threads at once, which will remove this requirement.

A mechanism within the coordination server makes sure that all responses to several (almost) simultaneous requests arrive at the right client: It contacts the responsible sandbox and passes the file descriptor of the socket the request came from down to the sandbox. The sandbox is then able to deliver the servlet's response directly to the web server. If there is no servlet to handle the request, the coordination server itself writes an empty response to the socket.

While the coordination server is responsible for handling and forwarding requests and responses, the servlet registry manages information about servlets and sandboxes. Each time a servlet is requested by the web server, the servlet registry might have to be updated. To identify whether this is necessary, the timestamp of the configuration file is stored by the registry, which minimizes the number of updates.

On each update, the servlet registry processes the configuration file. These updates occur rarely (at least on production servers, where such times matter). As a rule, only servlets and sandboxes that have changed are updated. Also, servlets are only removed when they are no longer in use or need to be reloaded. The update functionality could also have been implemented as a separate thread, but that would have meant that the registry would sometimes lag behind the configuration file's content.

The servlet registry manages also C++ Server Pages (HTML documents with special tags embedded). The CSP manager is able to automatically perform the transformation from CSPs to servlet source code and then to servlets, and it presents CSPs to the servlet registry just like normal servlets. For this transformation, it spawns compiler threads whenever this is necessary.

The possible crash of sandboxes is automatically compensated by the servlet registry's guardian thread which monitors the sandbox tasks and restarts them if they fail. By design, sandboxes may go down when an unstable servlet crashes.

3.3 The Sandboxes

Sandboxes are the only part of the C++ Servlet Environment that *by design* is allowed to crash. If a servlet crashes, it brings down the sandbox it runs in, all servlets within the same sandbox, all non-persistent sessions within the sandbox, but nothing else. The server's guardian thread (Section 4.2.5) notices the crash and restarts (and re-initializes) the sandbox. All requests for servlets inside the crashed sandbox are kept within the input socket and executed as soon as the sandbox is up again.

The reason for this design is that C++ code has to be compiled before it can be used, and if complex servlets (servlets executing more than just a pre-defined set of functions) are to be used, it is practically impossible to judge the quality of that code from within the program. But if servlets should be loaded into a running process and be actually executed (as opposed to interpreted), they are able to crash the whole task. Java has this problem too, but there it is only a single thread that crashes, which can easily be compensated. In C++, the whole task goes down when some code within it fails, which makes a different approach necessary.

Therefore, sandboxes have been included in the design of the C++ Servlet Environment. Sandboxes are designed as processes that are running on their own. They separate possibly dangerous code from the rest of the system and thus improve the stability of the server. The main thread of a sandbox reads commands by the servlet server from the socket, processes them by calling a servlet, and returns the responses directly to the web server.

Sandboxes can also be used to separate web applications from each other, since each of them provides its own session management mechanism. The number of concurrently running sandboxes is only limited by the maximum number of concurrently running processes[1] and the amount of available memory.

3.4 Servlets and C++ Server Pages

C++ Server Pages (in short: CSPs) are HTML documents in which special tags have been embedded. They must be compiled before they can be used like regular servlets. The CSP compiler within the servlet server is responsible for this task.

When the servlet server receives a request for a C++ Server Page it does not yet know about, the server asks the CSP Manager about the servlet. The CSP Manager then looks for a compiled

[1]This number is usually set by the operating system.

shared object with that name and a valid timestamp. If this is successful, there is usually no need to re-compile the CSP.

Else, the CSP Manager starts a new compiler thread. Compiling takes some time, and the server must not delay all other requests while waiting for the compilation process to finish, so all requests to CSPs that are currently in the compilation phase are queued, with one queue per CSP. Such a queue is flushed as soon as the compiler thread finishes.

The process of compilation itself is described in Section 5.2.3.

3.5 Database Access

Within the C++ Servlet Environment, persistent data storage is managed by the database interface. From the point of view of the CSE, a database can be almost anything that allows reading and writing of persistent data. However, the current implementation allows only SQL databases to be used for session management data.

3.5.1 SQL Databases

The SQL database interface allows the interaction of the CSE with most current databases. It is designed so that, in theory, a switch from one database to another one (for internal use — servlets can specify their own database drivers) only requires re-compiling the CSE. However, different databases usually interpret the standard differently, so that SQL commands often need to be changed after switching the database.

The interface itself is designed as a set of template classes. They provide general SQL database functionality without actually establishing contact with a database server. The template parameter — a traits class — encapsulates all database-specific code. After parameterization with this database driver class, the SQL database interface classes work together with a given database server.

3.5.2 Other Approaches

In the future, we are planning of enabling the CSE to access more generic databases which do not understand SQL, or which can be used more efficiently in another way. One such generic database is the Persistent Standard Template Library [Gsc01], a persistent implementation of the C++ Standard Template Library [Str00]. This library can be used in the same way the C++ container classes are used, and will give C++ programmers a more straight-forward method of using persistent data.

However, integrating the PSTL and other generic databases into the CSE will make a re-design for persistent session management necessary, which currently uses standard SQL queries. One way to solve this problem would be the introduction of a traits class for persistent session management.

In servlets and C++ Server Pages, non-SQL databases can already be used via their usual interface. In C++, this means including the database's header files and loading the necessary libraries into the sandbox.

Chapter 4

Implementation

This chapter discusses the technical details of the components of the C++ Servlet Environment introduced in Chapter 3: The servlet server, the sandboxes, and the interfaces to the web server and the database server.

4.1 The Web Server Module

The Apache HTTP Server supports the use of third-party code through its module interface. A module is a piece of software that implements this interface and has been linked as a shared object (see sidebar). Such modules can be loaded into the running web server by listing them in its configuration file, thereby adding new functionality to the server. The interface between the web server and the CSE's servlet server has been implemented as such a module.

Sidebar: ***Dynamically Shared Objects***

Almost all C programs use some so-called shared libraries. These libraries are files containing machine-code which is accessible via a well-known interface. The usual way to access such a library is to include its header file(s) in the source code of a program, and tell the compiler to perform dynamic linking to the library file. Upon execution, the library is loaded automatically and its functions are accessible to the program. As the name implies, the libraries can be shared, i.e. loaded by several programs.

A shared library is the most common example of a dynamically shared object (DSO), but it must be linked to the program at compile time. A more flexible way of using shared objects is loading them on demand. To do this, the programmer must know the location of the shared object file and the name and signature of the functions that should be used.

The system call `dlopen()` opens a DSO and returns a handle. This handle is, together with a function name, used by `dlsym()`, which returns a function pointer. The generic function pointer is casted to a function pointer matching the symbol's signature and can then be used like any local function. To unload a shared object, `dlclose()` is used.

This is done e.g. by the Apache HTTP Server [Apaa]. This web server uses so-called modules to extend its functionality. The modules are listed in the server's configuration file, so that the web server can be re-configured for additional purposes like using the HTTPS protocol [RRST99]. Because of the module interface, re-compilation of Apache is not necessary in this case.

The CSE's web server module comprises two parts: One is written in plain C and implements the Apache module interface, and the other connects the module to the servlet server and is written in C++, linked with C bindings so that its functions can be called from the first part.

This architecture eases the use of C++ in an environment which is written in plain C, while still providing a clean separation between those two languages.

Figure 4.1 shows the basic architecture of the Apache HTTP Server with the servlet module loaded.

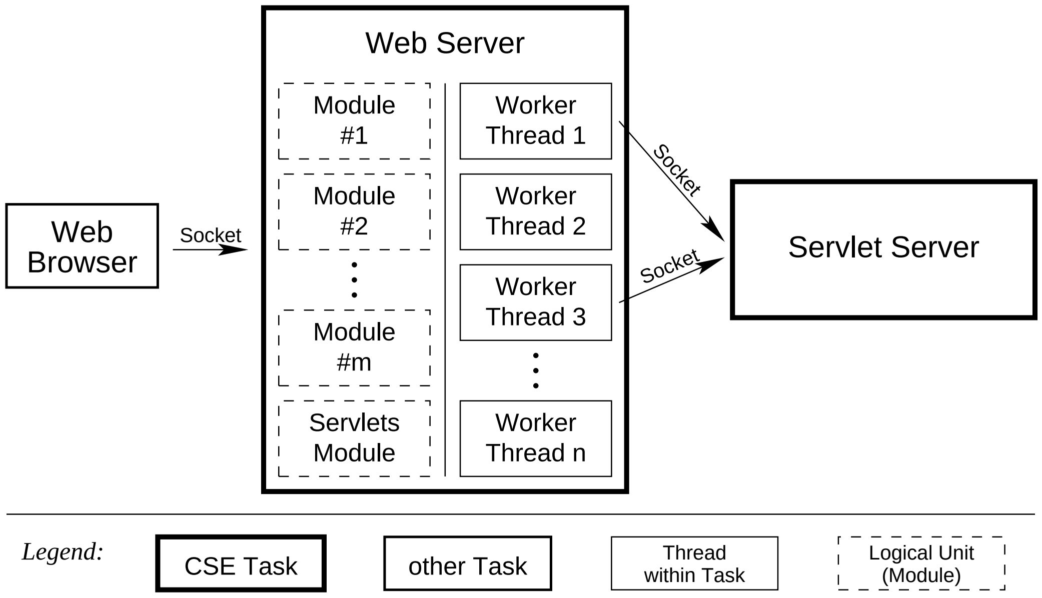

Figure 4.1: The Apache HTTP Server in the CSE

The Apache HTTP Server is itself a very complex piece of server software. Since version 2.0, parallelism has been encapsulated into so-called Multi-Processing Modules. In order to adapt the server for a wide range of host operating systems, several such modules have been written. The main problem with Multi-Processing Modules from the point of view of the C++ Servlet Environment is that it cannot be determined where the web server's requests come from: All from the same task, from a set of threads within a single task, from a set of tasks with a single thread each, or even from sets of threads within a set of tasks. This feature, together with the concrete thread implementation used, makes the synchronization of web server threads and the use of common state hard.

Because of this problem, we decided to have as little code as possible within the web server, with no necessity to keep any state there. A clean interface exists between the servlet module in the web server and the servlet server: The servlet server listens on a socket, which automatically queues requests coming from the web server's tasks and threads, and thereby synchronizes them. After this synchronization, the requests are distributed into concurrently executing tasks and threads.

4.1.1 Implementation of the Module Interface in C

The C part defines the Apache module. It adds a hook and a new directive to the web server. An Apache hook is a callback function which is executed when the web server is in a given state which determines the type of hook (e.g. when a URL is parsed, or when a new request comes in). An Apache directive defines a new configuration file option. Both are used by add-on modules to expand the web server's functionality.

The hook is called whenever a request to the web server is made. The directive is called `ServletConfig`. It sets the servlet server's configuration file. This file can also be set by the server's startup script, but administrators might prefer the option of configuring them from within the web server.

4.1.2 C++ Implementation

The C++ part defines the implementation of the hook and the handler for the directive. Both are linked with C bindings so that the code can be integrated into the web server as a shared object.

The handler for the directive invokes the command to set the configuration file at the Servlet Server, which will then re-configure itself.

The hook first determines whether a request to an URL fits a pre-defined pattern. This pattern defaults to `/servlets/*` for general servlets and `*.csp` for C++ Server Pages. Servlet requests are then passed on to the servlet server. However, we plan to modify this interface so that in the future the hook will be executed only for a given MIME type. A MIME type can be assigned to directories or extensions in the web server's configuration file, which is more flexible than a compiled-in default.

The information the servlet server needs for handling a servlet is extracted out of the web server's request structure and passed on to the servlet server. Afterwards, the thread waits until a response is written to the socket. This does not imply a performance loss because the Apache HTTP Server's Multi-Processing Modules support sufficient parallelism to assign a web server thread to each pending servlet request. The response is transformed back into data the web server can use, after which the hook returns execution.

If, for any reason, this process fails (e.g. the servlet server might not be started), the web server module declines handling the request. This behavior may cause the web server to return a *not found* error, or to deliver a static document if there is one with the same name. It also means that the source code of a CSP will be shown if the servlet server is down or no response is generated. Since some programmers like to put plain-text passwords into servlets, this might not be desirable. The standard UNIX security mechanisms are able to prevent that. To deny delivering the source code, the permission bits of the CSP files must be set so that the servlet server can read them, but the web server can't.

4.2 The Servlet Server

The servlet server comprises the coordination server class, the servlet registry class, and the CSP manager class. The following sections describe the implementation of these classes. Figure 4.2 shows an overview of the server's organization.

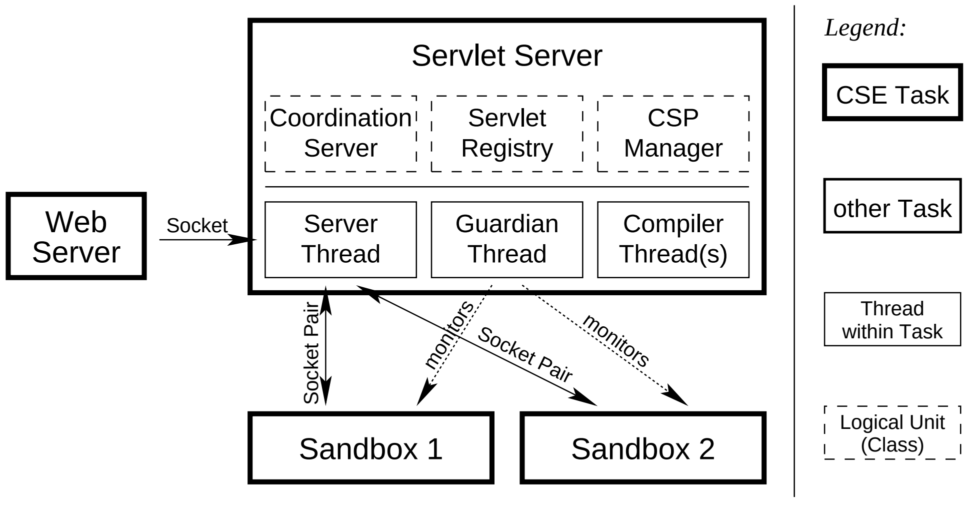

Figure 4.2: The Servlet Server

In those three classes, there is only a single server thread. It listens on the socket the server was assigned to and dispatches requests to the sandboxes, which behave like worker tasks. The server thread is also responsible for keeping the registry where the server's configuration is stored up to date. Finally, it spawns compiler threads when a C++ Server Page needs to be compiled.

The second permanently running thread within the server task is the guardian thread. It monitors the sandboxes and restarts them if they are crashed by a servlet.

4.2.1 The Coordination Server

The coordination server class is responsible for serving requests (together with the web server module and the sandboxes). The server thread is implemented as a loop that reads a request from the socket, handles it (usually by delegating it to a sandbox), and starts over again. At this point, some performance may get lost currently because this loop is executed by a single thread only. In the future, we plan to re-implement the coordination server so that several threads are able to handle requests.

When the web server receives a request for a servlet, the servlet module within the web server opens an Internet socket to the servlet server's port. The coordination server receives the request and queries the registry whether the requested URL is already assigned to a servlet. If it is, the registry informs the coordination server which sandbox is responsible for handling the request, and the coordination server forwards the request to the destination sandbox via the corresponding socket pair. The file descriptor (FD) of the socket on which the request came in is passed to the sandbox so that the response can be returned directly by the sandbox, as shown in Figure 4.3. Then, the coordination server is ready for another request.

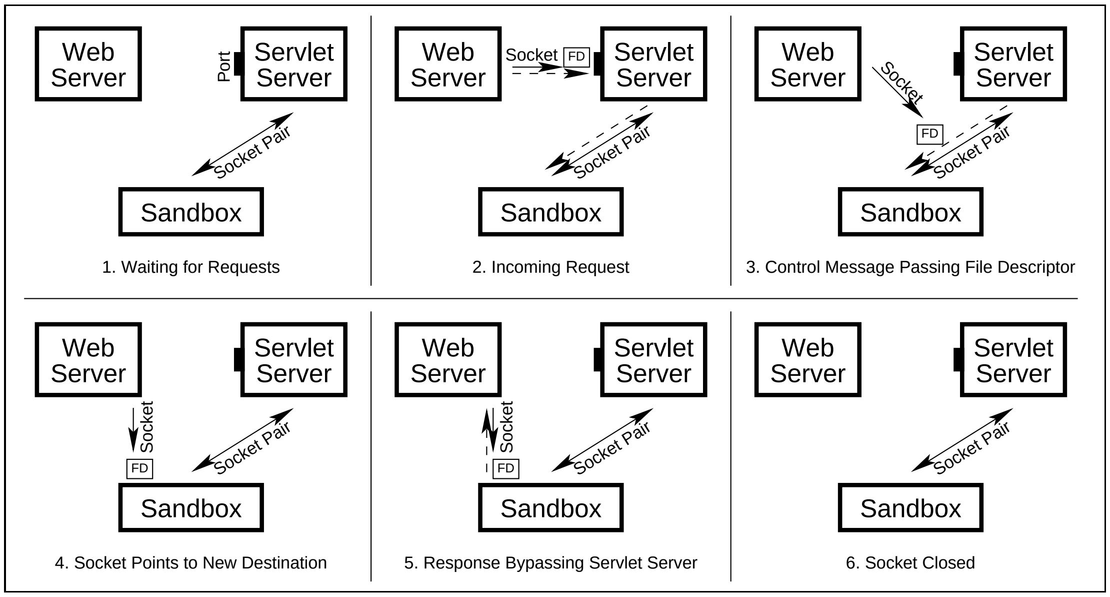
Web Server
Port
Servlet Server
Socket Pair
Sandbox
1. Waiting for Requests
Socket FD
2. Incoming Request
3. Control Message Passing File Descriptor
4. Socket Points to New Destination
5. Response Bypassing Servlet Server
6. Socket Closed

Figure 4.3: Servlet Request and Response

If the URL is not stored within the registry, the coordination server responds by declining the request.

If the URL requested by the web server points to a C++ Server Page (extension `.csp`), there is one more stage to that process. After asking the registry and receiving a negative result, the server notifies the CSP manager that a new CSP has been requested. If there is a CSP document with that name, the compiler is invoked as a separate thread.

4.2.2 The CSP Manager

The CSP manager class basically provides a mapping from C++ Server Pages (embedded C++ code in HTML documents) to servlets, so that the coordination server can neglect that difference.

When a request to a CSP is received by the coordination server, the CSP manager is contacted. It then performs a number of checks to deduce whether a compilation is necessary:

First, if another thread is already at work compiling this CSP, the request is simply added to a queue. Next, the CSP source file is located. If it is missing, the request has either been misspelled or the servlet has been removed by the administrator. In the latter case, the sandbox hosting it is informed of the removal. If the source file has been found, the directory that will be used for the shared object file is created. If this directory already exists and contains a compilation status file newer than the source code, the servlet is either up to date or the last compilation attempt has failed. In these cases, no compilation will be performed.

If a compilation is still necessary after these tests, a request queue is created in which all requests for that URL will be stored. The current request is placed on top of that queue. Then, a new thread is created, which will attempt to compile the CSP. The process of compilation itself will be described in Section 5.2.3, after the CSP syntax has been introduced.

After a successful compilation, the new shared object is registered like a normal servlet, and the responsible sandbox is informed. The requests that have been stored in the meantime are then read from the queue and executed as normal servlet requests, after which the compilation thread terminates.

4.2.3 The Servlet Registry

The servlet registry is responsible for mapping URLs to servlets. It also stores which sandboxes should be running and how they should be configured. It synchronizes its state with the configuration file, and accepts additional input from the CSP manager which adds CSP information to the registry as needed.

However, the registry does not store all information on the servlets. Detailed servlet information (like the location of the shared object file) is stored at sandbox level. The registry only holds enough information to be able to dispatch requests to sandboxes and to identify whether servlets have been added to or removed from the configuration file. This means that information only available at sandbox level must be re-created out of the configuration file if a sandbox crashes.

The registry also implements the guardian thread (which is responsible for restarting sandboxes and obtaining this information) and provides dynamic re-configuration.

4.2.4 The Server Thread

The server thread is the servlet server's main thread which may spawn other threads. It uses members of all three server classes. The thread initializes the startup procedure described below, and it contains the server's main loop. The main loop's purpose is to wait for incoming connections and forward servlet requests to the responsible sandboxes, as described above. It also keeps the servlet registry's information up to date, which is described in the remainder of this section.

Server Startup and Shutdown

The startup process is invoked by an independent task. It first starts the servlet server's main thread in a new task, then sends its configuration information (passed at the command line) to the server. The server thread first starts the guardian thread, since it doesn't require any other running tasks. Then, the server's main loop is executed, and the servlet server starts listening on its assigned port (the port number is configured during compilation) for commands from the web server or the startup process.

Since the server is not yet configured, the next step in the startup process is setting the configuration file. It is set either by the startup task or by the web server module. (In the latter case, the web server must be started after the servlet server.) The servlet registry is generated empty, and therefore the initial configuration can be done by calling the update functionality described below. As soon as the server knows the name of the configuration file, this functionality parses the configuration file, translates it into an object structure, generates the sockets for the sandboxes and writes the servlet information for the sandboxes to the output sockets.

The sandboxes are started by the guardian thread, as part of its normal operation. Sandboxes that are used only by C++ Server Pages are not known at that time and will be created on demand, when a CSP belonging to the sandbox has been requested for the first time. Whenever a sandbox is created, it first starts up its garbage collection thread (see Section 4.3 for an overview on sandbox architecture). Then, the sandbox enters its main loop.

If there is a request stored within the socket before the sandbox has been initialized, it is delayed until after initialization, since the sandbox does not have the necessary information yet. This happens when a sandbox is started because a CSP has been requested for the first time. Configuration commands stored in the input socket (from parsing the configuration file above) are processed first, causing the sandbox to load servlets. At the end of the configuration commands, the servlet server sends a notification that the initialization is complete, after which the sandbox can start executing servlets. When all the sandboxes known to the servlet registry have completed their initialization, everything is up and running.

The shutdown command is sent to the servlet server by a separate shutdown task. When the server receives this command, it stops accepting input from the web server. Next, it informs the sandboxes of the shutdown. Then the server process shuts down the guardian thread and leaves its main loop. Any running compiler threads are finished normally. The server process exits after the last thread has terminated, causing the sockets to be closed by their wrapper classes' destructors (Appendix B).

A sandbox receives the shutdown command from the server on its input socket. Because the servlet server has stopped accepting requests when sending this command and sandboxes process commands serially, the sandbox will not need to serve any more requests at that time. Since it does not need to save any state at shutdown (which would hinder automatic restarts), it terminates both the garbage collector thread and its main thread as soon as the shutdown command is received. Even when sandboxes have some unprocessed commands left to execute after the servlet server terminates, the pending responses will not get lost because the sandbox has already established direct communication with the web server then and does not need the servlet server for its responses.

The termination of the main thread causes the servlets' destructors to be called (a servlet programmer might want to log successful shutdown of a servlet). Also, it causes deletion of all non-persistent session objects in a sandbox (persistent sessions are stored by a database server), and it unloads all servlets and shared objects in memory. Finally, the garbage collection thread is terminated.

The web server (and, along with it, the servlet module) must be shut down separately. As long as the servlet server is inactive, the servlet module declines all requests, which causes the web server to behave as if the servlet module were not loaded.

Automatic (Re-)Configuration

When the servlet server starts up, the registry is initialized by parsing the configuration file. Before each access to its stored information, it checks the configuration file's timestamp to make sure that its information is current. If the timestamp has changed, it parses the configuration file again, adding or removing servlets as they are added to or removed from the file.

This comparison of the timestamp helps to keep the process efficient. A modern operating system will keep the timestamp in its cache memory where it can be retrieved quickly. On the other hand, this approach provides the semblance that the internal configuration is *always* up to date.

At the start of the update function, the registry clears a tag on all of its stored servlets and sandboxes. As an entry is encountered in the configuration file, the tag is set again, as it is set on new servlets or sandboxes. Finally, any servlets or sandboxes that have not been tagged are removed. Currently, this procedure removes all stored CSPs from the registry at every update. In the future, CSPs will get an extra tag that will cause the update function to ignore them. Another thread will be assigned to verify the registry periodically for CSPs that have been deleted from the document root directory.

The update function sends the information that a sandbox must know (URLs it manages, location of its shared object files) to the new sandboxes. When a sandbox's configuration is complete, it receives an initialization signal that causes it to start processing requests from the socket, which is only possible for an initialized sandbox.

The update function is also called when a sandbox has crashed and has been restarted by the guardian thread. The sandbox needs to be initialized again, and the registry alone does not have enough information to do this, so the guardian resets the coordination server's last known timestamp of the configuration file. This effectively causes the update function to be executed.

The update function should almost never have to be invoked on a production server. First, the configuration file will only change during a major upgrade (testing should be done on other systems). Second, servlets running on the production server should have been sufficiently tested, so that the server's administrator can judge the quality of the code loaded into the sandboxes (which is exactly what the program can not do by itself). For these reasons, the impact of the update function on the C++ Servlet Environment is small on systems where access times matter.

4.2.5 The Guardian Thread

The guardian thread, located within the servlet registry class, is responsible for the sandboxes' health. The sandboxes are also started by the guardian thread (at server startup).

As outlined in Section 3.3, sandboxes are allowed to crash when a servlet they host contains an error. The guardian thread periodically checks the state of the sandbox tasks. If it finds one that is down, it starts it again. Then, the guardian thread executes the servlet registry's update functionality which causes crashed sandboxes to be re-initialized, in the same way as during the server's startup phase. The socket that has been used by the failed sandbox task is reassigned to the new instance, so that pending inputs can be processed by it. In this case, the sandbox stores incoming requests until it has been initialized. The only unusual result the users may notice is a slightly longer response time, but, except for non-persistent sessions which are stored in the sandbox's local memory, no information is lost.

The guardian thread also helps to protect the servlet server against a certain kind of denial of service attack: If such an attack comes in the form of crashing a servlet by exploiting a bug in it, the sandbox is restarted quickly, and only non-persistent data are lost. Additionally, such events are logged in case somebody tries to use this bug to gain access to the machine.

4.2.6 The Communication Protocol

The servlet server reacts to a set of command codes (32 bit integers) on its port, which are mainly used by the web server module. The integer values are defined in `include/cse/defaults.h`, which sets up the symbolic values used here (within the namespace `CSE`). Each socket connection to the servlet server is used for a single command; successive commands by the same client would rarely be useful because they are not used by the web server module. The following commands are implemented at the servlet server:

`SR_CONFIG`: This command takes a string parameter which is used to set the name for the configuration file. The change takes effect at the next request to a servlet.

`SR_QUIT`: Causes the server to shut down immediately.

`SR_LOG`: Reads one parameter and discards it. This command is only included for backward compatibility, it was once used to set the server's log file. In the current version, the syslog is used for logging, so there is no more use for this command.

`SR_STATUS`: Returns a short status message. This command can be used to monitor the servlet server.

`SR_URL`: This command causes the server to process a servlet request. It reads in the URL and the filename of the corresponding document within the web server's document root (which is needed for C++ Server Pages). Further arguments are the request method, a list of cookies, request parameters, and the request body. These arguments, however, are not used by the servlet server; they are read by the responsible sandbox, which at that point has taken over the input socket (see below).

The sandboxes expect similar commands from the servlet server via their socket pair connections. These commands are implemented at the sandboxes:

`SB_CONFIG`: This command takes two parameters: The URL to be serviced and the location of the shared object file. It configures the sandbox to host a new servlet, to use a different servlet for a given URL, or to reload a shared object file.

`SB_QUIT`: Shuts down the sandbox, unloading all shared objects.

`SB_INIT`: Initializes the sandbox. The sandbox will start executing servlets only after this initialization command. This is necessary if a request to a servlet is stored within the pipe before the configuration commands, which can occur after a sandbox restart. The command's parameter sets the sandbox, which will be used for session management and for logging.

`SB_NEW`: This command takes the same two parameters as `SB_CONFIG`. Its main difference to that command is that it loads a new servlet without trying to locate an old version, i.e. it skips some validity checks.

`SB_DELETE`: Removes a servlet from the sandbox. The servlet's URL is read as the parameter. This command also unloads the shared object if it is no longer in use.

`SB_LOAD`: Loads a generic shared object into memory, without adding a servlet to the sandbox. This can be necessary if a servlet depends on a shared library. The path to the shared object file is passed as a parameter.

`SB_URL`: This command executes a servlet request. It takes several parameters in this format: First, the URL assigned to the request is read. Next, a socket's file descriptor is retrieved and used for reading the remaining parameters. This socket will also be used for the response, which will then bypass the servlet server for performance reasons. The remaining parameters are the request method, a list of cookies, the request parameters, and the request body.

A command to unload a generic shared object is currently not implemented. However, as, this command would include the necessity to keep a record of all generic shared objects within the sandboxes at the servlet server, we plan to provide a command to unload all non-servlet shared objects from a sandbox. This command could be executed by the update functionality when a sandbox section is opened within the configuration file. The command will probably be called `SB_UNLOAD`.

4.3 The Sandboxes

The sandboxes are the worker tasks of the C++ Servlet Environment. As shown in Figure 4.4, within each sandbox task two threads are running simultaneously. The execution thread contains the main loop, in which the sandbox reads requests from the servlet server, executes a servlet, and returns the output to the web server. The garbage collection thread is responsible for removing out-of-date session information from memory.

The sandbox code is structured into the servlet engine which loads and executes servlets, the session manager which is responsible for generating and retrieving session information, and the garbage collector which deletes obsolete sessions. The session manager stores persistent sessions in a database and therefore keeps a connection to a database server open.

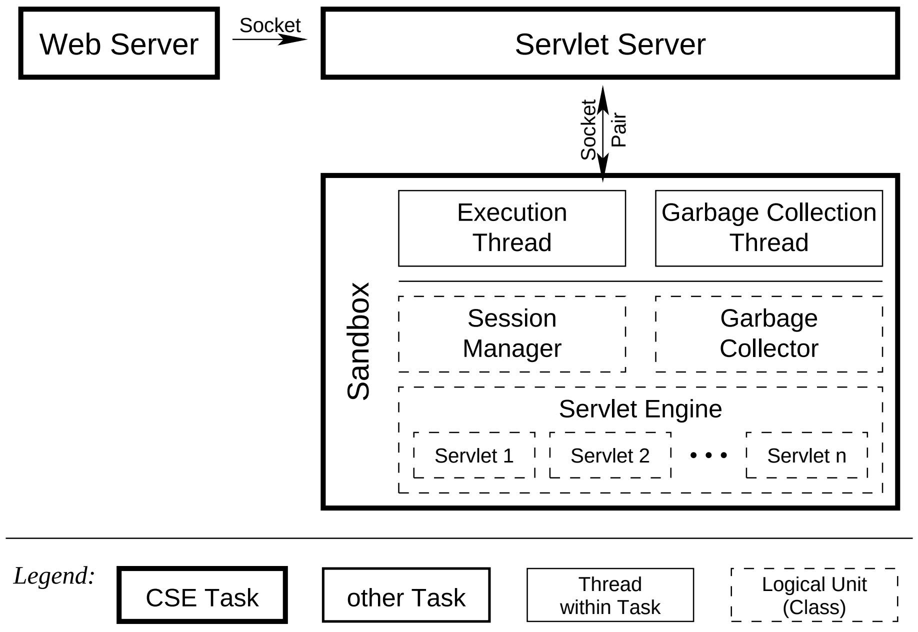

Figure 4.4: The Sandboxes

All sandboxes run in parallel, as separate tasks, without any coordination between them. That means that a failure of one sandbox has no consequences for the other sandboxes, and the failing sandbox can simply be restarted by the server's guardian thread. Within the sandboxes, however, requests are executed one at a time. Still, each servlet has a flag that tells the sandbox whether accesses to the servlet may be parallelized. The behavior of the sandboxes may be changed in the future so that they can handle servlet requests in parallel (Section 2.4 discusses possible performance gains).

4.3.1 The Servlet Engine

Within the C++ Servlet Environment, servlets are implemented as dynamically shared objects (DSOs). When such a shared object is loaded, its factory function (a function with a well-known name and signature within each DSO implementing a servlet) is called. It returns an instance of the servlet stored within the DSO. This servlet will be used to handle all requests to a single URL. In future versions of the CSE, we plan to expand the factory function by an URL-dependent parameter so that multiple servlets can be compiled into a single shared object, which is currently impossible.

When a sandbox is asked to process a request, it first checks the timestamp of the servlet file against the timestamp stored within the sandbox. If the timestamps differ, the servlet is reloaded before processing the request. This mechanism ensures that a new version of a servlet is used after re-compilation as soon as it is requested. After checking the timestamp, the request is stored in a request object, an appropriate response object is created, and the servlet is executed. It writes its output along with other information (including e.g. the MIME type) into the response object. After execution, the response object's contents are written to the output socket (which then points to the web server) and the sandbox is ready for another request.

4.3.2 The Session Manager

The session manager is a static class (a single instance exists within each sandbox). It services both the sandbox itself and the servlets located within the sandbox. Hence, sessions generated in one sandbox cannot be used by servlets running in other sandboxes. This implies that sandboxes can form a border between different web applications provided by a single server, or a separation between several virtual web servers.

Session management can be done automatically. This is requested from within a servlet or C++ Server Page by setting the `session` flag to true (the default) in the servlet's constructor. This flag is inherited by every servlet from the `Servlet` class. The automatically generated session is passed to the servlet at the entry point of the request functions. The session information still must be passed on when following links within the servlet, links to another servlet, or when filling in forms. A good way to do this is to set a cookie [KM97] containing the session's designation at the client's host. However, this does not always work since some clients don't support cookie management (or have turned it off). Within the C++ Servlet Environment, there are three support functions that generate a cookie, a hidden field within a form, or an URL-encoded argument, each storing the session information. If the servlet cannot rely on cookies, one of the latter two functions must be called at every link.

It is also possible to request a session manually. This is done by contacting the Session Manager directly. Once the session is generated, it can be used exactly like an automatically generated session. Although we think that automatic session management should be the ideal solution for most servlets, there might be special cases in which manual session management is preferred by the servlet programmer. Therefore, we have provided this option within the C++ Servlet Environment.

4.3.3 The Garbage Collector

Each sandbox provides a garbage collector. It is run as a separate thread within the sandbox, and its accesses to session data are synchronized using a binary semaphore.

From time to time, it tells the session manager to check the timestamp on the sessions kept at the sandbox. Every non-persistent session has an expiry date which is updated on each request, and if that point in time has been reached, the session is deleted.

The behavior of the sandbox's garbage collector can be adjusted before compilation of the CSE: The interval between two runs of the garbage collector as well as the default session timeout for persistent and non-persistent sessions may be set. These default times can be changed for individual sessions by the servlets using them.

4.4 The Database Interface

An important service that servlet servers have to provide is a database interface. It is used for everything from keeping simple connection state to huge catalogs containing a web application's memory. In the case of connection state, the database interface should not be visible. In other cases, an explicit interface for database accesses is needed. The C++ Servlet Environment's database interface, along with its session manager, tries to fulfill both of these goals.

The database interface is implemented in two layers: The top layer is a general-purpose database interface which defines how databases can be accessed from within the C++ Servlet Environment. It is implemented as a set of template classes and provides functionality common to all supported databases, e.g. for creating a database connection.

The bottom layer is the concrete database driver. There can be several such drivers that may also be used simultaneously. A driver is implemented as a traits class (see below) that is used as the template parameter for the general-purpose interface classes.

We also plan to offer the possibility to use non-SQL databases (such as text files) in the CSE's next version. The interface to those databases will not provide the full range of options available in SQL, but it will offer the possibility to store session data there.

4.4.1 General-Purpose SQL Database Interface Classes

The general-purpose SQL database interface consists of a number of classes, most of which must be parameterized with a concrete database driver when they are instantiated.

The following servlet code fragment shows a use case of the database interface, using the concrete database driver "`MyDriver`". We will frequently refer to this example in the following paragraphs. It first creates a simple database table designed to store a hit count of each authenticated user visiting a servlet page. Note that there are easier ways to do this, e.g. by storing the hit count within a session, but this is a demonstration of the database interface and not one of session management mechanisms. Here is the example:

```
 1 string user;
 2 unsigned int threshold= 50;
 3 try {
 4   DB<MyDriver> db("visits", "localhost", "me", "mypwd");
 5   db.execute("create table visits (
 6                  userid varchar(8) not null primary key,
 7                  hits integer default 0);");
 8   [...]
 9   DBQuery<MyDriver> query(db);
10   query << "select * from visits"
11     << " where hits > " << threshold << ";" << ends;
12   DBResponse<MyDriver> response(query);
13   re << response.size() << " records found." << endl;
14   DBIterator<MyDriver> i;
15   for (i= response.begin(); i != response.end(); ++i) {
16     re << (*i)[0] << ": ";
17     DBObject* hits= (*i)["hits"];
18     re << int(*hits) - threshold << endl;
19   }
20 } catch (exception e) {
21   re << "Exception: " << e.what() << endl;
22 }
```

Starting with line 9, visitors that have come to the page more than 50 times are selected from the database. The number of these users is printed first, then for each user the number of requests surpassing these 50 times is written to the response object. Should an exception occur (e.g. if the connection to the database server cannot be established), an error message is printed. Some of this code is unnecessarily complicated because we want to show what is possible, not what is sensible.

The interface classes are:

`DB`: This is the basic database class. Its main responsibility is opening and maintaining a connection to the database. The constructor takes the database name, the database host, the user name, a password, and a port on which to connect to the database server. All these parameters have reasonable defaults: They are set to the values the database creation scripts use for a default installation. In the example, the port parameter has been left at the default, which is a good idea in most cases.

The class `DB` contains the method `execute()`, which executes an SQL command that does not return data. In the above example, lines 5–7 create a table called visits which stores the number of page hits for each user.

`DBQuery`: This object eases the formulation of SQL queries. The constructor takes the `DB` object as a parameter, whose database connection is used for the query. The class supports write operations in the same way the `ostream` class does [Str00], as shown in lines 10–11. For static queries, the query string can also be passed directly to the constructor. The current content of the query can be extracted by the function `str()`. The method `clear()` prepares the object for a new query.

`DBResponse`: This class acts in many aspects like a C++ container class [Str00]. It is instantiated by the constructor, which takes an initialized `DBQuery` object as its parameter. On creation, the SQL query is executed, and the number of records found is stored. It can be accessed, as usual for a C++ container, by the `size()` function (line 13). The function `getCols()` returns the number of columns the query returned.

The class keeps a history of returned data objects, which is automatically written using the method `addToHistory()`. The member `clear()` deletes the objects in the history list when the `DBResponse` class is deleted. This history is necessary because the database interface does not return data items, but pointers to a base class of the concrete object class. This implementation facilitates user-defined datatypes, but it also makes it impossible to know when a pointer is no longer in use. Therefore, the history is created and maintained. The programmer can use the `clear()` method in order to help the environment freeing unused memory, especially when using large databases, or he can wait until the destructor of a `DBResponse` object is invoked.

`DBIterator`: The functions `DBResponse::begin()` and `DBResponse::end()` return C++ iterators that can be used in a `for` loop like the one in line 15 of the example. This statement iterates over all records of the `response` object. The `DBIterator` is a C++ input iterator and supports all of the basic input iterator operations [Str00]: Reading (`*i`), accessing (`i->`), iterating (`++i`, `i++`), and comparing (`==` and `!=`).

`DBRow`: The read operation of the iterator returns a `DBRow` object, which is usually not stored. This object again supports the `size()` function, which returns the number of columns in the row. The index operator which is defined on integer and string returns a single data field. Since the `DBRow` object is usually not stored, accessing a column at a `DBIterator`'s current row looks generally like line 16 and 17 in the example.

`DBObject`: The return type of `DBRow`'s index function is a pointer to one of the subclasses of `DB-Object`. The subclasses currently used within the C++ Servlet Environment are: `DBString`, `DBInt`, `DBLong` (64 bit integer), `DBFloat`, `DBDouble` (64 bit floating point number), `DBBlob` (binary large object), and `DBDateTime` (date, time, or both). These classes may contain a `NULL` value instead of an actual data item. In that case, an exception is thrown if the value is used. This can be prevented by first evaluating the function `isNull()`.

The classes can be converted to their corresponding builtin and to other datatypes such as `int`, `long long int`, `float`, `double`, `string`, and `char*`. If a specific conversion is not possible, an exception is thrown. The `DBObject` subclasses can be written to a stream. Also, the pointers to these classes (as returned by `DBRow::operator[]`) can directly be written to a stream. Therefore, a command like line 16 of the example is possible.

Since the database interface does not know by itself when a pointer to a `DBObject` is deleted, and since some objects hold pointers or references to other objects (e.g. an implementation of `DBQuery` usually holds a pointer to a `DB` object), a simple rule must be followed in deleting the database interface objects: An object may only be deleted when there are no more objects in existence that were created, directly or indirectly, from the object in question. This rule is more general than the actual dependencies, but easier to understand and to follow. (The actual dependencies also vary depending on the concrete database driver implementation.)

This restriction is easy to follow by adhering to a good programming style: In the example introduced above, instantiation is done by defining the objects or calling member functions, which forbids any other order of declaration. Also, the objects used here are not deleted explicitly, but implicitly when the control flow leaves the closure opened by the `try` statement in line 3. This is either done by successfully leaving the block after line 19, or by raising an exception from within the block. (Implicit deletion is in itself a good reason to use this technique because else the programmer would not know what objects need to be deleted in the `catch` block.)

4.4.2 The Concrete SQL Database Driver

A concrete database driver is implemented as a traits class. The classes `DB`, `DBQuery`, `DBResponse`, `DBIterator`, and `DBRow` use it as the template parameter. Since `DBObject` and its subclasses only encapsulate data items, there is no need to parameterize these classes.

Table 4.1 sums up the types and functions that have to be implemented for a concrete database driver.

Sidebar:* *Traits Classes

C++ introduces traits classes [Mye95, Str00] as classes encapsulating different implementations of a given concept. In that way, they are very similar to the concept of an implementation class in Java which implements one or more (abstract) interfaces.

Unlike Java, C++ does not have a construct to declare an interface — an abstract class with only virtual method comes closest, but nothing prevents programmers from placing code within this abstract class. Since virtual methods have an additional overhead in finding the concrete implementation for a given object, they can be much slower than static methods. The best solution in C++ is a template class which can be parameterized with another class providing the necessary implementation (the traits class). This latter class can be replaced by another class supporting a different implementation of the same interface.

In C++, a traits class does not require a special declaration. Its usual members are type definitions and inline functions (Appendix B), because these members can be inlined by the compiler, eliminating the overhead of a regular function call. When the template class is parameterized with a concrete traits class, the C++ compiler checks for the presence of all symbols that are used in its code. These symbols must be defined in the traits class as well.

Type/Function	Description
ConnectionType	Object/structure holding database connection information
ResponseType	Object/structure holding response information
CursorType	Type of a database cursor
getConnection()	Create a new connection to a database
removeConnection()	End a connection to a database
executeCommand()	Execute an SQL command without returning data
getResponse()	Execute an SQL command that returns data
removeResponse()	Delete a response object/structure
getRows()	Return the number of records in a database response
getCols()	Return the number of columns in a database response
getCursor()	Return a database cursor
copyCursor()	Copy a database cursor
incrementCursor()	Advance a database cursor
removeCursor()	Delete a database cursor object/structure
getData()	Read a data item out of a database response

Table 4.1: Types and Functions of a Concrete Database Driver

The functions could easily have been designed as three objects (connection, response, and cursor), but we think that a single traits class is easier to use as three of them. Due to this design, the traits class contains three groups of functions: constructor-like functions that return new objects or structures, destructor-like functions that delete them, and member-like functions that work on them. The type definitions declare which concrete class or structure is used by these functions.

The following functionality must be implemented by a database driver class:

- The constructor-like functions are `getConnection()` (which takes the same parameters as the constructor of the `DB` class), `getResponse()` (taking a connection type object and a string), and `getCursor()` (which takes a response type object).
- The destructor-like functions are named `removeConnection()`, `removeResponse()`, and `removeCursor()`.
- The function `executeCommand()` works on a connection type object. It also takes the string to be executed as a parameter.
- The functions `getRows()` and `getCols()` work on a response type object. They return the number of records in the query and the number of columns in such a record.
- The function `getData()` also works on the response type object, but takes three more parameters: A cursor, and the row and column id of the data item that should be retrieved. The row id is necessary for database drivers that work without the use of cursors. The column id can be either an integer or a string (the column name). Therefore, there are two functions called `getData()` that differ only in the type of the last parameter. Usually, one of them calls the other one, or both functions call a third function containing the common code.
- The function `copyCursor()` works like a copy constructor for the cursor type object. It is useful in `for` loops which iterate over a database response.
- The function `incrementCursor()` tells the database driver that it should advance a cursor to the next record of the response.

Servlet programmers may want to use the database interface for their servlets as well, but they are also free to use any third-party database connectivity software. Persistent session management within the sandboxes, on the other hand, requires a standardized interface, and the connection to the database server is best opened during the startup phase of the sandboxes. This requires that the sandboxes are able to contact a "default" database server without external assistance.

This problem is solved by providing special header files that define the macros `DefaultDB`, `DefaultDBQuery`, `DefaultDBResponse`, `DefaultDBIterator`, and `DefaultDBRow`. Such a header file is provided by every database driver implementation. When the C++ Servlet Environment is compiled, one of these files is included and thereby sets the default driver. This default is configured before the compilation of the CSE.

The MySQL Driver

At the time of writing, we support a concrete database driver for the MySQL database.

MySQL [WA02] is an open source database that is used for many open source projects and is among the most popular Internet databases. Because of the free availability and the wide use of MySQL, we decided to implement the first database driver for this database. MySQL has backends for many different programming languages, among them also one for C++ called Mysql++ [AMW01]. The concrete database driver for MySQL uses this C++ driver.

The file `cse/db/mysql/default.h` (in the CSE's include directory) includes everything necessary for the driver and defines MySQL as the default database if there is no default database yet. If MySQL is configured as the default database before compilation, the file `cse/db/default.h` will include this file, else it will point to another database driver.

Of the types introduced in Table 4.1, `CursorType` is set to `void` because MySQL does not (yet) use cursors. `ConnectionType` points to the `MysqlConnection` class, which manages the connection to the MySQL server. `ResponseType` uses the `MysqlRes` class (which contains results from the MySQL server).

The scripts `createuser.sql`, `createdb.sql`, and `createtables.sql` (contained in Appendix A) initialize the database for use as the C++ Servlet Environment's default database. They create the user `cse` with an empty password (who can only connect from the local host), the database `cse` which belongs to this user, and the tables necessary for session management. These scripts are provided by the CSE installation in the directory `share/cse/db-scripts/mysql`.

4.4.3 Use of Non-SQL Databases

Currently, only SQL databases are supported for session management. Servlets, however, are free to use whatever database interface the servlet programmer prefers.

In the future, we plan to implement a more general approach that will make it possible to store session data in non-SQL databases as well. The design of this interface has been outlined in Section 3.5.2.

4.4.4 Persistence

In many web applications persistence of data is of great importance. Session data, for instance, need to be stored for some time so that a user does not have to log in over and over again. Other data have to be stored longer, such as orders in a web store which must be stored as long as the deliverance process lasts, and here it is very important that these data must not be lost after they have been accepted, even if the server host crashes immediately after the response to the web browser has been delivered.

The C++ Servlet Environment distinguishes between several persistence levels, which are described in the following sections.

Volatile Data

First, some data do not need to be stored within a database, but will be retrieved from another interface and/or are stored within the servlet itself. For these data, no special infrastructure is necessary.

Local variables used for processing a request can be declared and used anywhere within the servlet's `service` function, as usual in C++. These variables are deleted as soon as the `service` function finishes and are re-created on the next execution. Variables that need to be stored within the servlet are simply declared in the `Servlet` subclass and are visible as long as the servlet remains in memory. System or third-party libraries are accessed by including their header files and (if they are not already used) loading the libraries into the sandbox.

Memory Sessions

The second persistence level are session data that do not need to be stored within a database. These data are stored in the sandboxes within a `MemorySession` object. It is requested by setting the servlet's `session` flag to `true` and the `sessionType` variable to `MemorySession`. (These variables will in the future be merged into a single one.) These data are lost when the sandbox goes down, either by normal termination of the sandbox or the server, or by an unstable servlet that crashes.

Database Sessions

More safety can be gained, at the cost of some performance, by using a `DatabaseSession` object. It provides the same interface as the `MemorySession` does. The necessary access to the database interface is hidden by the session manager. The switch from a `MemorySession` to a `DatabaseSession` is done by setting the servlet's `sessionType` variable to `DatabaseSession`.

Sessions, whether they reside within the main memory or within a database, have a timeout. A default value for that timeout is set when the server is configured, and it can be overwritten by a servlet for a given session. The timeout is measured starting with the last access of the servlet's session. Once it is reached, the session can be deleted by the sandbox's garbage collector.

Besides keeping memory and database usage low, the timeout also improves data security. On a session based web application server there is always a chance that somebody might discover the session ID of someone else, e.g. by entering random session IDs. The C++ Servlet Environment minimizes this danger by assigning session IDs in a random order, by keying session IDs to a single sandbox, and by deleting them when they are not used for some time. This mechanism is essential for data security, especially on a big web site, and should not be circumvented.

Permanent Data

The top persistence level can be accessed by the C++ Servlet Environment's database interface. It provides a uniform way to access SQL databases using C++. The database connection can be established on loading the servlet, which provides the best performance, or only when it is needed.

By using the `DefaultDB` macro, the database beneath a web application can, in theory, be changed without much effort. In practice, however, different databases usually have a different view on the interpretation of the SQL standard, understand different additions to the standard, and recognize different forms of persistence. So, this persistence level depends mainly on the database that is used by a servlet.

Chapter 5

Writing Web Applications

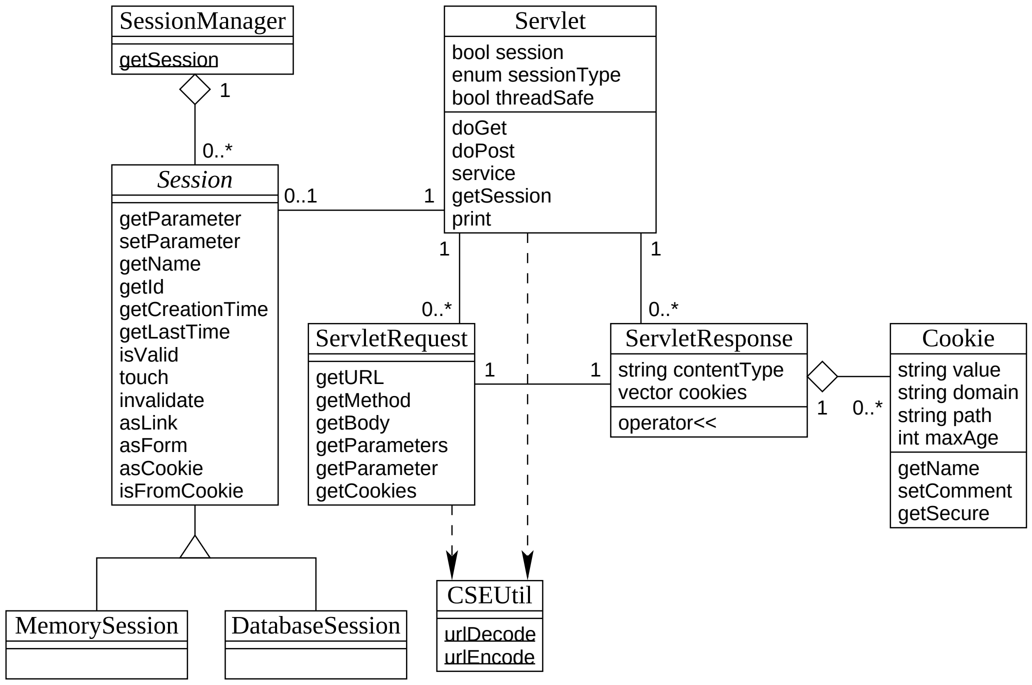

Figure 5.1: The Servlet API

The C++ Servlet Environment allows developers to implement their web applications either as servlets in C++ source code, or as C++ Server Pages, which are HTML documents extended by special tags. Both approaches have their advantages, and both will be explained in detail in this chapter.

The servlet programming interfaces have been designed similar to their Java equivalents, which means that the main problems Java programmers have to overcome are the difficulties of the C++ programming language rather than that of a new interface. C++ programmers, on the other hand, will have no problems in understanding the example servlets and can start immediately by modifying them, using the API documentation as a reference when necessary. The CSP syntax might be new to them, but their understanding of the compilation process should help them to learn the use of the CSP tags quickly. Programmers who have previously written CGI scripts only, however, will have to change their view on web application development. On the other hand, the shortened development and maintenance times which a complete servlet infrastructure makes possible should more than make up the time used for learning the new concepts.

5.1 The Servlet API

The Servlet Application Programmer's Interface (API) consists of the Servlet class, which serves as the base class for all servlets, the Servlet Request and Response classes, which encapsulate requests to and results from servlets, the Session Manager class, which manages sessions within a sandbox, and the two smaller classes Session and Cookie. The namespace `CSEUtil` contains some useful functions which may also be used by the servlet programmer. Figure 5.1 shows these classes and their connections within the CSE.

5.1.1 The Servlet Class

Every servlet is implemented as a subclass of the Servlet class. This means that the servlets inherit the functionality of the parent class. For most members, a default which usually does not need to be changed is defined in the base class. Other members, such as the `service()` function, need to be implemented by most servlets. Besides the Servlet class, a servlet needs to use at least the Servlet Request and Response classes in order to provide dynamic server pages. These classes provide access to HTTP requests and responses and are explained in Section 5.1.2.

The Interface

First, a constructor and a destructor may be implemented by subclasses of the Servlet class. They can set up and delete data structures and initialize the servlet. However, in some cases the destructor might not be executed when the servlet is unloaded, because sandboxes are allowed to crash when hosting unstable code. If state must be kept permanently, it has to be placed outside the sandbox the servlet runs in (e.g. in a database).

The most important part of the Servlet class is responsible for processing servlet requests. There is the `service()` function, which is called every time there is an access to the servlet. Its default implementation executes either the `doGet()` or the `doPost()` function, handling the two most important types of requests. The programmer has the option of overriding these two functions, or of overriding the `service()` function itself, which allows handling different types of requests within a single function. All three functions take an `HTTPRequest`, an `HTTPResponse`, and a pointer to a `Session` object as parameters.

`Session* getSession(const string& name)`
: Creates a new session with the given session name within the current sandbox. The session ID is assigned randomly. If the new ID is already in use, another random ID is chosen.

`Session* getSession(const string& name, const long int& id)`
: Retrieves an existing session with the given name and ID from the current sandbox' session manager. The function returns `NULL` if the session cannot be found.

`Session* getSession(const string& name, const ServletRequest& rq)`
: Evaluates the given ServletRequest object in order to retrieve or create a session. The function first looks for a cookie, then for an HTTP request parameter with the given name. If this is successful and a session corresponding to the result exists, it is returned. Else, a new session with the given name is created (see above).

Table 5.1: `Servlet::getSession(...)`

As shown in Table 5.1, there are three overloaded `getSession()` member functions implemented within the servlet class. These functions are used by the Session Manager class, and it should usually not be necessary to call them directly. Automatic session management is activated by setting the `session` flag within the servlet's constructor. Section 5.1.3 explains session management in detail.

The programmer has the option of overriding the servlet's `print()` function. This function prints some information about the servlet to an output stream. It might be used e.g. for automatically generating an overview on the servlets loaded within a sandbox. The `print()` function is called by an overloaded instance of `operator<<()`, which makes it possible to write the servlet object to an output stream, as in "`cout << servlet;`".

Another function that is not actually part of the Servlet class is the `factory()` function. It must be provided with C bindings by every shared object, so that the loader is able to find it. It returns a new instance of the servlet that is implemented within the shared object.

Special Member Variables

The C++ Servlet Environment recognizes several member variables and flags on a servlet. They can be set by the programmer within the constructor (and can even be changed later — evaluation always happens before the execution of the `service()` method). This section explains how these variables are interpreted. Table 5.2 lists them briefly.

Variable	Description
`bool session`	Servlet needs session management
`enum sessionType`	Type of requested session management
`bool threadSafe`	Multi-threaded execution enabled

Table 5.2: Special Servlet Class Variables

These variables are implemented:

`session`: If set to false, session management is deactivated for this servlet. Default: `true`.

`sessionType`: Tells the Session Manager which kind of session to provide whenever the `service()` function is executed. Depending on the persistence level needed for a session (Section 4.4.4), the values `MemorySession` and `DatabaseSession` may be used. If a session has already been created for this connection, it is retrieved, else a new session is created (see Section 5.1.3 on session management). Default: `MemorySession`.[1]

`threadSafe`: On servlet level, multithreading is implemented by calling the `service` function whenever a request arrives, which may result in multiple simultaneous executions by different threads. If this flag is set to false, every sandbox the servlet runs in will provide a binary semaphore protecting the servlet so that only a single instance is allowed to run at any time. This can be necessary if the servlet does not provide synchronization by itself. If a servlet keeps any state within the servlet class, accesses to that state need to be synchronized manually when `threadSafe` is set. Since it slows down execution, the flag should not be used excessively. Default: true.[2]

5.1.2 Servlet Requests and Responses

In order to provide dynamic servlet output, it is also necessary to access functionality implemented within the Servlet Request and Response classes.

[1]This variable will be merged into the `session` flag in the next release. It will then offer the values `MemorySession`, `DatabaseSession`, and `NULL` for no session management.

[2]This flag is not used yet, since servlet execution within the sandboxes currently is not yet happening in parallel. Still, it should be set to the right value, since this behavior may change in the future.

Servlet Requests

`const string& getURL()`	The Request URL, without encoded parameters.
`const string& getMethod()`	The HTTP request method, e.g. `"GET"`.
`const string& getBody()`	The HTTP request body, unprocessed.
`const map<string, string>& getParameters()`	The request parameters.
`void getParameter(const string& name, string& par)`	Retrieves a single request parameter.
`const vector<Cookie>& getCookies()`	Cookies sent with the request.

Table 5.3: Servlet Request Access Functions

The Servlet Request class contains information on a request to the web server. This request object is generated by the sandbox the servlet runs in and passed by calling the servlet's `service()` function. The programmer accesses the stored information by a number of access functions (Table 5.3):

`getURL()` and `getMethod()` return the HTTP request URL and the HTTP request method, respectively.

`getBody()` returns a string containing the HTTP request body (e.g. for a `POST` request). The parameters entered into an HTML form can be retrieved easier using `getParameters()` (see below). This method is meant for raw access to the request body.

`getParameters()` returns a map containing request parameters from an HTML form. If the request is a `GET` request, they are extracted from the request URL (after the question mark). In the case of a `POST` request, they are taken from the request body. The parameters are automatically decoded from the *URL-encoded* format. The map returned by `getParameters()` maps request parameter names to parameter values. An easier way to retrieve the value of a single parameter is provided by the function `getParameter()`.

`getCookies()` returns a vector of Cookie objects that have been passed to the server with the HTTP request. In the next version, this vector will be replaced by a map which will make retrieving a given cookie easier. If one of these cookies contains session management information, it is evaluated automatically, as explained in Section 5.1.3.

An easy way of accessing request parameters that may be set to multiple values (e.g. from a list in an HTML from in which multiple choices may be selected) has not yet been implemented, as well as a method for retrieving a file submitted by an HTML form. Methods for these and possibly some more simplified parameter queries will be added in the future. In the meantime, the functions `getParameters()`, `getURL()`, or `getBody()` can be used to emulate this functionality.

Servlet Responses

The Servlet Response class object is generated with default values by the sandbox and passed to the servlet (in writable state). Similar to the Servlet Request, this class stores parameters that will be passed to the client of the web server. All of these parameters can be set by the servlet developer.[3] Table 5.4 shows a list of them.

Variable	Description
`string contentType`	MIME type of the response body
`vector<Cookie> cookies`	Cookies sent with the response

Table 5.4: Servlet Response Class Access Functions

Here is a longer explanation of the parameters:

`contentType`: A string that tells the client how to interpret the response body. The default type is "`text/html`".[4]

`cookies`: All these cookies will be passed to the client. It is not necessary to pass cookies that have been read into the Servlet Request object to the client again unless they should be changed. By default, no cookies are sent.

The response body is written to the Servlet Response object in the same way that output is written to the console or to a file in C++: by using the operator "<<". Any object that can be written to a stream can, in the same way, be written to the response object.

5.1.3 Session Management

Session Management is implemented in the three overloaded functions called `getSession()` and in the Session Manager class.

Every sandbox contains a static instance of the Session Manager class. It is responsible for storing Session objects, creating new sessions, assigning existing sessions to new requests, and deleting sessions on request or when they are no longer accessed.

Session management is usually done automatically: When the `session` variable on a servlet is set to a non-null value (Section 5.1.1), a new or existing Session object of the requested type is passed to the servlet's `service()` function on every access. It can then be used just like any other local variable. Changes to the Session object are stored within the Session Manager or within a database (Section 4.4.4).

The Session Manager also implements the sandbox' garbage collection mechanism. This functionality is periodically called from a separate thread and deletes obsolete sessions.

The session parameters are accessed by two functions, `getParameter()` and `setParameter()`. Both functions take a string (the name of the session parameter) and the object that should be used. The storage function stores a copy of the object either within local memory or within a database, depending on the session's type. The retrieval function returns a local copy of the stored object. If the object is not found within the session, it is created and initialized with the object given to the retrieval function.

The `Session` class supports storage of `string`, `int`, `long long int` (64 bits), `float`, and `double` (64 bits) type objects, which can be stored both within local memory and within a database. The `MemorySession` subclass supports all kinds of objects by using template functions for storage and retrieval. This added flexibility can only be used by casting the general `Session` pointer to the more specific `MemorySession` pointer, and it leaves the responsibility for type checking to the servlet programmer.

A common way to use variables stored within a session is to first declare the variables, then call the `getParameter()` function for each variable, use it like a local variable, and at last call the `setParameter()` function for the variables. This approach is shown in Figure 5.2. Note that if

[3]The current version does not provide access functions since they are not necessary. For completeness, `setContentType()` and `addCookie()` will be implemented in the next version.

[4]Binary content is not yet supported, because the web server interprets a `NULL` character as the end of input. We are working towards a solution of this problem.

```
// Servlet::session must not be set to NULL!

void MyServlet::service(
  const ServletRequest& rq,
  ServletResponse& re,
  Session* session)
{
  string name= "Anonymous Coward";
  session->getParameter("name", name);
  // do something with name
  session->setParameter("name", name);
}
```

Figure 5.2: State Within a Session

the session parameter is not yet existing, it is initialized by the given object (in the example, by the string "Anonymous Coward"). Functions like `const string& getParameter(const string& name)` are not available because `string` is not the only class supported by `getParameter()`.

Every session has a unique key, composed by the sandbox name (not needed within the sandbox), the session name, and the session ID. Session name and ID are managed automatically, but can be retrieved from the Session object using `getName()` and `getID()`.

A session usually also has a timeout, which is the interval past the last access after which it may be deleted by the garbage collector. The interval itself is configured prior to compilation of the CSE, for each session type separately. The function `isValid()` can be used to see whether a session is still safely within the interval, or is already marked for removal. The `invalidate()` function immediately marks the session for removal by the garbage collector. The `touch()` function updates the access timestamp and is automatically called when the session is retrieved at the `service()` function's entry point. Mostly for debugging purposes, access to the sessions creation time and the last access time are provided by the functions `getCreationTime()` and `getAccessTime()`.

Session IDs

The session ID is a 32 bit integer, which means that there are more than 4 billion different numbers that qualify for session IDs. They are generated randomly, with a new random number chosen in the event that a generated number is already in use. Also, session IDs are valid only within the sandbox in which they have been generated, which is an improvement over existing servlet environments. These properties make it very hard to guess a session ID (important for security critical applications).

In order to use sessions, the session ID must be passed between the web server and its client (because the web server, according to [FGM+99], does not keep any connection state). The C++ Servlet Environment supports three ways to pass the session ID: In the *URL-encoded* format used e.g. in a hyperlink (`?id=4711`), as a hidden field within an HTML form (`<input type=hidden name=id value=4711> </input>`), or as a cookie. The functions that transform the session ID into the *URL-encoded* or the form format are `asLink()` and `asForm()`. A cookie is returned by the function `asCookie()`. The cookie method is the easiest, but it is not supported by every client (see below).

The programmer is free to use or not to use cookies. For example, the line

```
if (!session->isFromCookie()) re.cookies.push_back(session->asCookie());
```

sets a cookie if the current session has not been read from a cookie (determined by the function `isFromCookie()`). If cookies are not used, the session parameter must be passed on at every hyperlink or form.

It would have been possible to automate session ID management completely by either relying on cookies only, or by providing functionality for integrating the session ID into every link or form. However, both solutions have disadvantages for some application areas, so we implemented convenient functions for both cases and left the final choice to the servlet programmer.

The Use of Cookies

Cookies are not supported by every client, and even when they are supported, they can usually be turned off by the user. Servlet programmers therefore should warn their users that they need to store cookies in order to access a servlet, or — which is a better, but also more complicated solution — provide an alternative way of passing the session ID in case cookies are switched off.

Cookies can also be used to store more than just the session ID. In fact, they can be used to store all data that is needed within the servlet, in which case session management is not necessary anymore. In this case there also should be an alternative way of storing that information.

Cookies are primarily a way to store a string variable, mapped to a static string. However, they can also do more:[5] A domain name and a path can specify the cookie's area of use (e.g. "`.company.com`" or "`/servlets`"). The cookie can also have an expiry date (default is a year). After this time has passed, the cookie is deleted by the client. The cookie can contain a comment that helps the user (or the user agent) to decide if it wants to accept it.

Finally, there is a flag that tells the user agent to pass the cookie only through secure connections (which can be necessary if it contains sensitive data). If the session ID itself is considered sensitive data, this flag must be set after receiving the cookie from the `asCookie()` function. Then, the cookie can be passed on to the Servlet Response object.

5.1.4 Loading a Servlet

When a servlet has been written, it must be compiled as a shared object. How this is done depends on the compiler, the GNU Compiler Collection's C++ compiler uses the switch `-shared`.

After compilation, the shared object needs to be mapped to at least one URL. This is done in the CSE's configuration file, which is explained in Appendix A. At the next access, the servlet is loaded into at least one of the sandboxes.

Loading shared objects is done using the `dlopen()` system call. It loads the shared object into memory and prepares it for executing its functions. Then, the `dlsym()` system call looks up the factory function. This function is defined with C bindings so that it can easily be loaded (Section 5.1.1).

The factory function returns an instance of the servlet. This instance is stored within the sandbox and used to process all requests to a single URL.

5.2 C++ Server Pages

C++ Server Pages (in short: CSPs) provide an easier way to use servlets. They are essentially HTML documents enriched with special tags.

These documents must be transformed into servlets before they can be used. This is done automatically by the servlet environment: A servlet source code skeleton is completed by parsing the CSP tags. Code within the tags is inserted into the proper locations of the skeleton, and the remaining HTML code is printed as text in the servlet's `service()` function.

The following sections explain the syntax and the semantics of the CSP tags, the compilation process, and how the compiled CSPs are loaded into the sandboxes. Finally, Section 5.2.5 describes caching mechanisms so that the CSPs do not have to be recompiled on every access.

[5] The cookies used within the C++ Servlet Environment implement the Internet RFC 2109 [KM97].

5.2.1 The CSP Syntax

The syntax of C++ Server Pages has been designed similar to the syntax of Java Server Pages [Sun01b]. This has been done so that Java programmers can quickly learn how to write CSPs. Additionally, we didn't want to invent a completely new syntax as there are already enough different ways to integrate code into HTML pages. Finally, JSP tags are standardized, so that these tags should never conflict with other extensions to HTML.

Tag	Description
`<%--comment--%>`	CSP comment, not sent to the client
`<%$sandbox%>`	Sandbox the CSP is placed into
`<%#include%>`	Includes a C++ header file
`<%!definition%>`	Class member definition(s)
`<%@initialization%>`	Statement(s) within the constructor
`<%=expression%>`	Evaluates a C++ expression
`<%code%>`	Executes some C++ code

Table 5.5: CSP Tags

Table 5.5 sums up the CSP tags. They are explained in more detail in this section:

`<%--comment--%>`: Holds a CSP comment. Unlike an HTML comment, it is not passed on to the user's web browser. Of course, the programmer can also include standard C++ comments within the code tags.

`<%$sandbox%>`: This tag is not transformed into code. The servlet will be placed into the specified sandbox. If that sandbox does not exist yet, it is created.

`<%#include%>`: Includes a C++ header file. Since it is often necessary to include header files within a C++ program, this tag has been provided. Please note that one can only include a single file per tag (but there can be several include tags within a CSP) and there must be no whitespaces within the tag. The include directives, regardless of their position within the CSP, are placed at the head of the source code so that the functionality will be available within the whole CSP.

`<%!definition%>`: Defines a class variable or function. This code will be placed within the class definition.

`<%@initialization%>`: Writes code into the class' constructor. This is the place where class variables are initialized. Variables and flags of the Servlet class can also be set here, e.g. the need for session management.

`<%=expression%>`: Evaluates a C++ expression. This can be a variable, as in `<%=i%>`, or any other kind of expression, like in `<%=i - strings.begin()%>` in the next section. It is printed where the tag occurs within the C++ Server Page.

`<%code%>`: This tag executes some code. There can be one or more statements within the tag. Output is not usually generated, except if it is done by the code within the tag. This tag can also be used for conditional statements or loops embracing CSP code. For this purpose, an opening brace ({) starting in one CSP code tag creates a closure that is ended by a closing brace (}) in another CSP code tag. An example of such a loop will be given in the next section.

Please note that the code within some CSP tags (variable definition, initialization code, general code) must be terminated by a semicolon. Other CSP tags (file inclusion, expression evaluation) must not be terminated that way. This requirement results from the placement of the code within the CSP tags into the servlet skeleton by the CSP compiler. CSP tags can not be embedded within other CSP tags.

5.2.2 A CSP Example

`TableServlet.csp`, shown below, is a simple example of a C++ Server Page. Its purpose is to store strings entered by its users.

```
 1 <%#vector%>
 2 <%!vector<string> strings;%>
 3
 4 <%
 5   // check whether a string should be removed/added
 6   ServletRequest::Map::const_iterator mi;
 7   if((mi= rq.getParameters().find("remove")) != rq.getParameters().end())
 8     strings.erase(strings.begin() + atoi(mi->second.c_str()));
 9   if((mi= rq.getParameters().find("string")) != rq.getParameters().end())
10     strings.push_back(mi->second);
11 %>
12
13 <html>
14  <head>
15   <title>TableServlet</title>
16  </head>
17  <body>
18   <h1>TableServlet</h1>
19   <p>This servlet stores strings within a table.</p>
20   <form method=get action=TableServlet.csp>
21    <p>
22     Please enter a string:
23     <input type=text size=64 name=string></input>
24     <input type=submit value="Go!"></input>
25    </p>
26   </form>
27   <p>
28    <table border=1>
29     <tr>
30      <th colspan=2>Strings entered to date: <%=strings.size()%></th>
31     </tr>
32     <% for (vector<string>::iterator i= strings.begin();
33        i != strings.end();
34        ++i) {
35     %>
36      <tr>
37       <td><%=*i%></td>
38       <td>
39        <a href="TableServlet.csp?remove=<%=i - strings.begin()%>">remove</a>
40       </td>
41      </tr>
42     <% } %>
43    </table>
44   </p>
45  </body>
46 </html>
```

Line 1 includes a C++ Standard Template Library header file. Other system libraries' header files can be included in the same way, offering access to a wide range of code.

Line 2 defines a vector of string objects for storage. It is, by default, created empty and will contain the strings entered by the servlet's users.

The lines 5–10 contain pure C++ code. The names "remove" and "string" are searched in the servlet request's parameters. If the first one is found, the value is interpreted as the integer position in the vector from which an element should be deleted. The second parameter enters a new string into the vector.

Lines 13–46 contain mostly HTML code, with some CSP tags embedded. The first CSP specific tag can be found in line 30. It evaluates and prints the current size of the vector (the number of strings entered so far).

Lines 32–35 start a loop, iterating over all elements of a vector. This loop is closed in line 42, and it generates a table from the vector's contents. Line 37 is responsible for generating the first column containing the string itself, by dereferencing the loop's iterator. The second column's content is generated in line 39. It is a link to the servlet itself, with a parameter that deletes the current line from the vector.

When the CSP is compiled by the servlet server (usually at the first request from a web client), the first step generates C++ source code. For this example this code, contained in `TableServlet.cc`, is shown below in full.

```
#include <vector>
#include <cse/cse.h>
#include <string>


class CSPServlet : public Servlet {

protected:
  virtual void print(ostream& os) const;

vector<string> strings;


public:
  CSPServlet() : Servlet() {

  }
  virtual ~CSPServlet();
  virtual void service(const ServletRequest& rq, ServletResponse& re,
    Session* session= NULL);

};


CSPServlet::~CSPServlet() { }

void CSPServlet::print(ostream& os) const {
  os << "CSP servlet compiled from TableServlet.csp" << endl;
}

Servlet* factory() {
  return new CSPServlet();
}

void CSPServlet::service(const ServletRequest& rq, ServletResponse& re,
  Session* session= NULL)
{
  re << "


";

  // check whether a string should be removed/added
  ServletRequest::Map::const_iterator mi;
  if((mi= rq.getParameters().find("remove")) != rq.getParameters().end())
    strings.erase(strings.begin() + atoi(mi->second.c_str()));
  if((mi= rq.getParameters().find("string")) != rq.getParameters().end())
    strings.push_back(mi->second);
  re << "

<html>
 <head>
  <title>TableServlet</title>
 </head>
 <body>
  <h1>TableServlet</h1>
  <p>This servlet stores strings within a table.</p>
  <form method=get action=TableServlet.csp>
   <p>
    Please enter a string:
    <input type=text size=64 name=string></input>
    <input type=submit value=\"Go!\"></input>
   </p>
  </form>
  <p>
   <table border=1>
    <tr>
     <th colspan=2>Strings entered to date: ";
  re << strings.size();
  re << "</th>
    </tr>
    ";
 for (vector<string>::iterator i= strings.begin();
       i != strings.end();
       ++i) {
      re << "
     <tr>
      <td>";
  re << *i;
  re << "</td>
      <td>
       <a href=\"TableServlet.csp?remove=";
  re << i - strings.begin();
  re << "\">remove</a>
      </td>
     </tr>
    ";
 }   re << "
   </table>
```

```
90   </p>
91  </body>
92 </html>
93
94 ";
95 }
```

The first tag, including the `vector` header file, is placed at the beginning of the file. It is followed by two other inclusions that are necessary for the servlet to work.

The class `CSPServlet` is declared in lines 6–22. Since the code will be loaded from a shared object, the name of the class is unimportant, as no name conflicts can arise with classes from other shared objects. In line 11, the vector we declared in the example is placed into the class definition, making it a normal member of the class. The constructor (in lines 15–17) is empty, since we did not use the `<%@initialization%>` tag. The rest of the class declaration is straight-forward.

The lines 25–29 define simple implementations for a destructor (empty) and the `print()` method which outputs some information about the servlet. Since the CSP compiler cannot know the purpose of the CSP, this information amounts to a line stating that the CSP was compiled automatically.[6] Lines 31–33 define the factory function, which returns an instance of the new servlet for use within a sandbox.

The real work is done starting with line 35. Here, the implementation of the `service()` method is located. (Since the CSP compiler cannot know which type of HTTP requests are handled by a servlet, it uses this more general method instead of e.g. `doGet()`.) It starts by outputting a few blank lines, which can be explained by lines 1–12 of the original CSP code — some line breaks there stand outside the CSP tags, and the transformation process is not yet intelligent enough to erase them.

The code from the top of the servlet (lines 5–10 in the CSP) can be found immediately below, in lines 43–49. It has been moved there without any transformations.

Starting with line 49, the HTML document is generated. The output is written to the servlet response object. The expression tags from the lines 30, 37, and 39 can be found at lines 69, 79, and 83. The loop that generates the table has been moved to lines 73–88.

5.2.3 The Compiler

The process of compilation consists of two stages. In the first stage, the CSP is parsed by the CSP Manager and a C++ source code file is generated. The CSP tags are transformed into code, and this code is inserted into a servlet skeleton source code file at the right places, as shown in the last section.

A human-readable compilation status file is generated, and if an error is discovered at the first stage, it is written into this file, (else, the file is generated empty). The compilation status files also serve as timestamps for the CSP Manager, which uses them as values for the last time a compilation has been attempted (Section 5.2.5).

A third, machine-readable file contains the designation of the destination sandbox for the CSP (generated out of the sandbox tag). This file is needed in case the server is shut down and restarted again, in order to minimize the need to re-compile CSPs.

If the first stage has completed successfully, the C++ compiler is invoked as stage two.[7] It tries to compile the C++ source code generated in stage one as a shared object file. Any errors from this stage appended to the compilation status file.

Table 5.6 shows the files that are created by a successful compilation.

[6]This implementation will be changed in the future: A default `print()` method will be implemented within the servlet base class. Servlets providing their own identification can then overwrite the default by using the `<%!definition%>` tag.

[7]The compiler itself is not implemented within the C++ Servlet Environment. The server can use any compiler that is able to generate a shared object out of C++ source code. Per default, it remembers and uses the compiler that it was compiled with itself.

File	Contents
`file.csp`	The CSP source code
`file.cc`	C++ source code generated by the CSP compiler
`file.sb`	Caches the destination sandbox for the servlet
`file.so`	The shared object file, for use within a sandbox
`file.err`	Any transformation or compiler errors and warnings

Table 5.6: Files Used by the CSP Compiler

5.2.4 Loading a CSP

After compilation, a CSP is handled exactly like a regular servlet (Section 5.1.4). After compilation, the CSP manager enters the CSP's information into the servlet registry. Still, the server must continue to check the CSP source document for changes. This process is described in the next section.

5.2.5 Caching Considerations

Of course, a C++ Server Page that has been successfully compiled once can be used as long as the original servlet remains unchanged. Also, if the compilation failed once, there is no point in trying to compile it again every time the CSP is accessed. Finally, when a minor change in the source code introduces an error, it would be good to have a backup servlet in store.

For these reasons, a number of improvements have been introduced into the compilation process.

First, if the compilation status file that has been generated during compilation (Section 5.2.3) is newer than the CSP itself, there is no need to re-compile. Either the compilation was successful (then the compilation status file contains only warnings and there is a shared object), or the compilation failed because of an error in the CSP. Until this error is corrected (which means that the access time of the CSP is updated), the compiler has nothing to do.

If there are several accesses to a CSP that has not yet been compiled, this should not lead to several running compiler threads that each try to compile the same source code. Therefore, the CSP Manager maintains a queue that stores requests to a given URL as long as the compiler thread for that URL is running. This queue is synchronized by a binary semaphore. As soon as the compiler thread finishes, the CSP Manager flushes the queue and executes the stored requests.

The compiler puts the shared objects and intermediate files into a well-known directory. If the compiler thread returns an error, there might still be an old version of the compiled shared object file available (or even an old shared object in memory). If it is there, it is used in order to minimize application downtimes. Only if the compilation finished successfully, the new shared object replaces the old one. The destination sandbox is notified of the change and reloads the shared object.

Another optimization is the file containing the destination sandbox (Section 5.2.3). This file is necessary in the case that the server is shut down and restarted. The first access to the CSP will discover that the compilation status file is newer than the CSP file itself, meaning that there is no need to re-compile everything. The only unknown is the destination sandbox, which has been cached within this file. It can now simply be read in, eliminating another compiler run — in fact, eliminating a great number of compiler runs after the server starts up again and all the CSPs need to be reloaded as the first requests come in.

Chapter 6

A Comprehensive Example: The Record Store

This chapter introduces and describes a webstore written for the C++ Servlet Environment. It can be used for selling audio compact discs (which we hereafter simply call records) over the web. It provides a front page for advertising recommended titles, database search and browsing, a shopping cart, login and registration functionality, detailed record description pages with custom images, support for credit card payment, and some simple warehouse management functionality. Not implemented are credit card verification and processing of orders past registration; these features would go beyond the purpose of this demonstration.

6.1 Overview

The Record Store is implemented as a set of HTML files, CSP documents, graphics, and database tables. HTML files are used for showing static documents such as a business agreement or a search form. The CSP documents generate the dynamic content out of static HTML code and database data. The graphics are used only by database entries to show images of selected records — we did not create a fancy design for an online record store. The database tables contain the store's items with descriptions and links to images, user identifications and orders, and also dynamic data like shopping cart entries and sessions.

In the next section we will discuss the organization of these database tables, which form the basic structure upon which everything else is built. Section 6.3 then lists the static and dynamic documents which form the store itself, and explains the working of the CSP documents.

6.2 Database Organization

The Record Store uses both the common CSE database in which sessions are kept and a private database to store specific data.

This section focuses on the latter database. It is a MySQL database called "record" and belongs to the user "record" who can connect from the local host without a password. The following statement[1] creates the user and the database (executed as root):

```
connect mysql;
create database record;
grant all privileges on record.* to record@localhost;
```

The definitions of the database tables are explained below. In those definitions, it is assumed that a connection to the database "record" as either "record" or root has been established.

[1]All MySQL specific code has been tested with version 3.23.52.

6.2.1 `record`

This database stores the records for sale in the record store. It is defined below:

```
create table record (
  artist varchar(80) not null,
  record varchar(80) not null,
  genre varchar(80),
  label varchar(80),
  producer varchar(80),
  year integer,
  price float not null,
  store integer not null,
  record_desc text,
  cover text,
  content varchar(80),
  primary key (artist, record)
);
```

The artist and the record name uniquely identify a record. Other properties a record needs are a price tag and the number of copies left for sale (store).

The other columns show further information that may be helpful to the customers: Genre informs about the type of music on the record, label contains the record company name, producer the person(s) who produced the record, and year the production date. The field record_desc may contain a detailed description of the record. Cover may refer to a graphic file stored within the web server's document root directory, and content stores its MIME type (which is not yet used by the Record Store).

6.2.2 `artist`

This table stores information about an artist.

```
create table artist (
  artist varchar(80) not null primary key,
  type varchar(80),
  genre varchar(80),
  label varchar(80),
  producer varchar(80),
  artist_desc text
);
```

An artist is identified by a string which may contain the name of a single person, a band designation, or a pseudonym. The type field further explains the kind of artist and may be set e.g. to "Artist", "Band", or "Orchestra". The artist field must be the same for every record made by that artist or combination of artists. Artist_desc may contain information about the artist.

Genre, label, and producer may also be set in this table. They can be used as default values for entering new records.

6.2.3 `track`

This table stores information about individual tracks on a record.

```
create table track (
  artist varchar(80) not null,
  record varchar(80) not null,
  track integer not null,
  name varchar(80),
  length time,
  sample text,
  content varchar(80),
  primary key (artist, record, track)
);
```

A track is identified by the artist, the record, and the track number (track). Track information may include the track name and its length. Also, the URL of a sound file (sample) may be referenced to help customers decide on whether they would like to buy the record. The field content then determines the MIME type of this file.

6.2.4 `image`

This table is used to connect images to record descriptions.

```
create table image (
  artist varchar(80) not null,
  record varchar(80) not null,
  pos integer auto_increment,
  image text,
  content varchar(80),
  primary key (artist, record, pos)
);
```

An image belongs to a record identified by artist and record name. The position (pos) determines which image should be placed first. It is automatically assigned (appending the image) if set to a NULL value. The image field refers to the URL of the image, and content contains its MIME type.

6.2.5 `top`

This table contains items that should be placed on the front page of the Record Store.

```
create table top (
  pos integer auto_increment primary key,
  artist varchar(80) not null,
  record varchar(80) not null
);
```

A valid entry must specify all three columns. As in the `image` table, pos specifies the position that the image should appear at, starting with the smallest value on the left. Artist and record identify the record that should be placed at that position. All other data will be retrieved by the servlet.

6.2.6 `genre`, `label`, `producer`, and `type`

These tables may contain values that can be used as defaults. They may, for example, be used as pre-defined values in a drop-down list. The tables all contain only a single column:

```
create table NAME (
  NAME varchar(80) not null primary key
);
```

These tables are only provided for convenience, they are not needed for the correct functioning of the store.

6.2.7 `user`

The preceding tables together are sufficient for presenting the store's stock. Still, a functioning web store also needs to store some business logic related data, which is done in the next few tables.

The table `user` stores registration information like the password used for the account or the address records should be sent to. It is created by the following command:

```
create table user (
  uname varchar(80) not null primary key,
  pword varchar(80) not null,
  first varchar(80) not null,
  last varchar(80) not null,
  address text not null,
  email varchar(80) not null
);
```

Every field must be filled out to generate a correct entry. Uname registers the username for accessing the store on its web frontend. Pword contains an encrypted password used for authentication purposes. First and last contain the user's real name, for postal delivery. The address field is also needed for that purpose. Finally, email contains the user's e-mail address which might be used for confirming orders.

6.2.8 `cart`

This table contains the content of the current shopping carts.

```
create table cart (
  session integer not null,
  artist varchar(80) not null,
  record varchar(80) not null,
  amount integer,
  primary key (session, artist, record)
);
```

Since (different copies of) a record may be taken by several users, a shopping cart may contain several records, and every user (identified by the session ID) may use only a single shopping cart, the primary key consists of the session ID and the record identification (artist and record). The amount value contains the number of records currently in the shopping cart.

6.2.9 `request`

This table contains the processed and confirmed orders.

```
create table request (
  uname varchar(80) not null,
  artist varchar(80) not null,
  record varchar(80) not null,
  amount integer not null,
  price float,
  ccard varchar(80),
  cnumber varchar(80),
  cdate varchar(80),
  instant timestamp
);
```

The table does not have a primary key. This feature does not restrict orders in any way and implies that a single user can order the same record twice (e.g. if someone needs two separate bills for bookkeeping).

Uname contains the customer's username, from which the current postal address can be deduced. Artist and record identify the record that was ordered, and amount specifies the number of copies. The price field can contain the price at the date of the order, which means that later changes in the record's price do not affect this order.

Ccard, cnumber, and cdate contain the data necessary for ordering by credit card: The credit cart type (which company issued the card), its number, and its expiry date.

The instant field has a `timestamp` type, which means that it will be automatically set to the current date and time when a row is added or changed. Since timestamps in MySQL have a granularity of one second [WA02], also this field can not be used as a primary key.

6.3 HTML, CSPs, Servlets

This section presents the code of the Record Store in detail. CSP tags, C++ related comments, design decisions, and, programming techniques will be explained at their first occurrence.

6.3.1 `index.html` and `top.csp`

The file `index.html` only redirects browsers from `/store/` to `/store/top.csp`. It can be removed if the web server's default file is changed to `top.csp` for this directory.

```
<html>
 </head>
  <title>The Record Store</title>
  <meta http-equiv="refresh" content="0; url=top.csp">
 </head>
 <body>
  <h1>Welcome to the Record Store!</h1>
  <p>Please <a href="top.csp">click here</a> to proceed.</p>
 </body>
</html>
```

The file `top.csp` contains a sophisticated front page. It is generated dynamically from the database table `top`, which contains a number of records that should be placed here. This file will be explained in several chunks.

```
1 <%$store%>
2 <%#cse/db/default.h%>
3 <%@session= false;%>
```

The first three CSP tags tell the servlet environment how the servlet should be handled. It will, like all other Record Store servlets, be placed into the sandbox "store". This makes it independent of all other servlets on that machine and restricts sessions generated here to the webstore. Line 2 adds the header files for the default database, in this case MySQL. Since the Record Store's complete configuration is contained in the database, they will be needed in every servlet.

The third line contains code that will be placed into the servlet's constructor. The code in this tag will set the servlet's `session` flag to `false` so that the servlet environment will not provide a session when the servlet is executed.

```
4 <%
5   DefaultDB db("record", "localhost", "record");
6   DefaultDBQuery q(db);
```

```
 7   q << "select * from top"
 8     << " order by pos;"
 9     << ends;
10   DefaultDBResponse r(q);
11 %>
```

This block is contained within a CSP code tag. It will be placed at the top of the servlet's `service()` function which is executed on every request.

It first creates a connection to the database `record` defined in the last section of this chapter. Depending on the time needed for establishing that connection, line 5 could also have been placed into a `<%@constructor%>` tag. In that case, the connection would be maintained over several requests. However, the servlet then might have to be declared as non-thread-safe, depending on the database driver's ability to use a single connection for several requests. This could be done by the code fragment `<%@threadSafe= false;%>`, placed e.g. between lines 3 and 4.

Lines 6–10 create and execute a (static) command that extracts the necessary data out of the table `top`. It selects all data there, ordered by the field pos (from left to right). The response generated in line 10 will be needed to display items on the front page. Since the command within the query is a static string, it could also have been placed directly into the `DefaultDBQuery` object's constructor.

```
12
13 <html>
14  <head>
15   <title>Our Top Titles - Record Store</title>
16  </head>
17  <body>
18   <h1>Welcome to the Record Store!</h1>
19   <h2>Our Top Titles:</h2>
```

This code is transformed into a string that will be written to the servlet's output. Since it is written below the `<%code%>` tag containing the database connectivity code, it will be written after the connection has been established.

The next code chunk contains a loop that has been distributed over two separate `<%code%>` tags. It starts in line 22 and terminates in line 43. The code within the loop (both HTML code and CSP tags) will be executed once for each row of the data within the table `top`. The iterator `i` points to the line for which the loop is currently executed.

```
20   <table>
21    <tr valign=top>
22     <% for (DefaultDBIterator i(r); i != r.end(); ++i) { %>
23       <td width=192>
24        <%
25          DefaultDBQuery q2(db);
26          q2 << "select cover, record_desc from record"
27            << " where artist=\"" << (*i)["artist"] << '"'
28            << " and record=\"" << (*i)["record"] << "\";"
29            << ends;
30          DefaultDBResponse r2(q2);
31          DefaultDBIterator i2(r2);
32        %>
33        <p align=center><a href="record.csp?artist=<%=CSEUtil::urlEncode(
              (*i)["artist"]->operator string())%>&record=<%=CSEUtil::urlEncode(
              (*i)["record"]->operator string())%>">
34          <% if ((*i2)["cover"]->isNull()) { %>
35            <img src="noimage.jpg" width=128 height=128></img>
36          <% } else { %>
37            <img src="<%=CSEUtil::urlEncode((*i2)["cover"]
                 ->operator string())%>" width=128 height=128></img>
38          <% } %>
39        </a></p>
40        <p><%=(*i2)["record_desc"]%></p>
41        <p>
42       </td>
43     <% } %>
44    </tr>
```

Each iteration of the loop creates a new column within an HTML table (lines 21–42). This results in the selected items being placed from left to right on the screen. After a new column has been opened, another database request is performed in lines 24–32. This one includes dynamic data from the first request: The values for the fields artist and record are taken from the current line of the first database response. In line 27, the command `(*i)["artist"]` first dereferences the iterator `i`, which requests the new line of the first database connection. Then, it requests the value of the column artist, which is placed on the stream for the second database request. The same is done for the record name in line 28. Lines 30 and 31 execute the request and generate an iterator.

Line 33 (which occupies several lines in this printout, but is written as a single line in the CSP) generates a hyperlink to the selected record (which will be shown by the CSP `record.csp`). To do this, two `<%expression%>` tags of the same structure are necessary. They first extract information out of the first database iterator in the same way as above, but instead of writing it to a stream, they explicitly convert it into a string. This output is then used to call `CSEUtil::urlEncode()`, which converts special characters into the URL-encoded form necessary for form parameters [FGM+99]. The final output of line 33 looks like `<p align=center><a href= "record.csp?artist=`*`artist+name`*`&record=`*`record+name`*`">`.

The lines 34–38 contain a distributed `if` command. If the column cover of the current record contains a NULL value, line 35 is executed, else it is line 37. Note that the braces of the `if` command within the `<%code%>` tags are necessary even when they only embrace a single line. The servlet environment might still convert this line into several separate commands, and a syntax error could result from omitting them. Both line 35 and 37 generate an image with a fixed size. In the first case, it is taken from the file `noimage.jpg`, which contains a graphic showing the user that no image is available. In the other case, the image name is taken from the database.

Line 40 extracts the description of the current record from the database and puts it below the image, which concludes the information placed within the current column. The servlet then continues with the next iteration of the loop or, if there the last iteration has already finished, with the code below it.

```
45    <tr>
46     <td colspan=<%=r.size()%>>
47      <table width=100%>
48       <tr>
49        <td width=33% align=center>
50         <b><big>&gt;&gt; <a href="search.html">Search</a> &lt;&lt;</big></b>
51        </td>
52        <td width=34% align=center>
53         <b><big>&gt;&gt; <a href="search.csp">Browse Our Catalog</a>
            &lt;&lt;</big></b>
54        </td>
55        <td align=center>
56         <b><big>&gt;&gt; <a href="cart.csp">Shopping Cart</a>
            &lt;&lt;</big></b>
57        </td>
```

```
58       </tr>
59       </table>
60     </td>
61    </tr>
62   </table>
63  </body>
64 </html>
```

The rest of the servlet prints static data. It will be centered below the featured record descriptions, which is done by introducing a broader column which spans over all record descriptions (line 46). Lines 50 and 53 contain links to the pages introduced in the next section, which are used to search and browse the record database. The shopping cart can be reached from the link in line 56.

6.3.2 `search.html` and `search.csp`

The documents presented in this section are used to search and browse the record database. The browsing functionality is implemented simply as a search with no constraint. In order to execute a search, `search.html` is used.

```
 1 <html>
 2  <head>
 3   <title>Search - Record Store</title>
 4  </head>
 5  <body>
 6   <h1>Search Our Database</h1>
 7   <p>Please enter one or more strings below:</p>
 8   <table>
 9    <form action=search.csp method=get>
10     <tr>
11      <td align=right>Artist:</td>
12      <td><input type=text name="artist"></input></td>
13     </tr>
14     <tr>
15      <td align=right>Record:</td>
16      <td><input type=text name="record"></input></td>
17     </tr>
18     <tr>
19      <td align=right>Label:</td>
20      <td><input type=text name="label"></input></td>
21     </tr>
22     <tr>
23      <td colspan=2 align=center><input type=submit value="Search"></input>
           </td>
24     </tr>
25    </form>
26   </table>
27  </body>
28 </html>
```

This document shows a simple search form. It allows search by artist, record, and label. The parameters entered into the browser are submitted by the GET method to the servlet `search.csp`, as defined in line 9. A search is always executed for all given fields at once (AND search). OR searches are not implemented here, they can be executed by filling out the form several times.

This servlet, `search.csp`, processes the search requests:

```
 1 <%$store%>
 2 <%#cse/db/default.h%>
 3 <%#cstdlib%>
 4 <%#strstream%>
 5 <%@session= false;%>
```

Like above, the servlet is placed into the sandbox "store", needs the database header files, and requires no session management. This servlet additionally requires the files `cstdlib` and `strstream` from the C++ standard library. The header `cstdlib` includes the C standard library header `stdlib`, and `strstream` contains types that can be used both as streams and as strings.

```
 6 <%
 7
 8   string artist, record, label, page;
 9   rq.getParameter("artist", artist);
10   rq.getParameter("record", record);
11   rq.getParameter("label", label);
12   rq.getParameter("page", page);
13   int pg= atoi(page.c_str());
14   if (pg <= 0) pg= 1;
```

This code requests the search parameters `artist`, `record`, and `label` which have been filled out by `search.html`. Also, a parameter called `page` is read and converted into an integer. This parameter is necessary for requests that result in more than ten results, because these results are distributed over several pages. If the page parameter has not been specified of is invalid, it is reset to 1 and the first page of the result is displayed.

```
15
16   DefaultDB db("record", "localhost", "record");
17   DefaultDBQuery q(db);
18   q << "select * from record where 1=1";
19   if (artist != "") q << " and artist=\"" << artist << '"';
20   if (record != "") q << " and record=\"" << record << '"';
21   if (label != "") q << " and label=\"" << label << '"';
22   q << " order by artist, record;" << ends;
23   DefaultDBResponse r(q);
24   DefaultDBIterator i(r);
25
26 %>
```

This code block executes the query and requests an iterator for the result. The slightly strange condition `1=1` in line 18 is an easy way to retain the validity of the SQL statement even if no parameter is specified. In lines 19–21, conditions are added to the query if they were specified in the request parameters.

```
27
28 <html>
29  <head>
30   <title>Search - Record Store</title>
31  </head>
32  <body>
33   <h1>Database Search</h1>
34   <p>Your query was: <i>Find all records
```

```
35     <% if (artist != "") { %>
36       by &quot;<%=artist%>&quot;
37     <% } %>
38     <% if (record != "") { %>
39       named &quot;<%=record%>&quot;
40     <% } %>
41     <% if (label != "") { %>
42       from the label &quot;<%=label%>&quot;
43     <% } %>
44    </i></p>
45    <% if (r.size() == 0) { %>
46      <p>We have no records in our database matching your query.</p>
47    <% } else if (r.size() == 1) { %>
48      <p>It found 1 result.</p>
49    <% } else { %>
50      <p>It found <%=r.size()%> results.</p>
51    <% } %>
```

This code shortly sums up the requested database query in a human-readable sentence. It also displays the number of results, again encapsulated within a distributed `if` statement. This statement is necessary for displaying a different string if no results have been found, as well as for avoiding the popular programming error that shows strings like “1 results”.

```
52    <table width=70%>
53     <% if (r.size() > 0) { %>
54       <tr>
55        <td colspan=3><b><big>
56          <% if (r.size() < 10*pg) { %>
57            Results <%=10*(pg - 1) + 1%> to <%=r.size()%> of <%=r.size()%>
58          <% } else { %>
59            Results <%=10*(pg - 1) + 1%> to <%=10*pg%> of <%=r.size()%>
60          <% } %>
61        </big></b></td>
62       </tr>
63     <% } %>
```

This code prints the results show on the current page, as in “Results 1–10 of 27”. It is suppressed if there were no results.

```
64     <%
65       if (pg - 1 > (r.size() - 1)/10) pg= (r.size() - 1)/10 + 1;
66       for (int k= 1; k < pg; ++k) for (int j= 0; j < 10; ++j) ++i;
67     %>
68     <%
69       DefaultDBIterator e(i);
70       for (int j= 0; j < 10; ++j) ++e;
71     %>
```

In line 66, the iterator `i` is set to the first of the search results that should be displayed on the current page. This is done only after verifying the value submitted by the form. In lines 69 and 70, another iterator is created and set to the record immediately after the last one to be printed. This is possible because a C++ iterator always contains an end mark after the last data set. Once this end mark is reached, the iterator cannot be incremented further. Now, the loop statement in line 72 is very simple:

```
72    <% for (; i != e; ++i) { %>
73      <%
74        string thisArtist((*i)["artist"]->operator string());
75        string thisRecord((*i)["record"]->operator string());
76        string thisLabel((*i)["label"]->operator string());
77      %>
78      <tr valign=top>
79       <td width=64><a href="record.csp?artist=<%=CSEUtil::urlEncode(
            thisArtist)%>&record=<%=CSEUtil::urlEncode(thisRecord)%>">
80       <% if ((*i)["cover"]->isNull()) { %>
81         <img src="noimage.jpg" width=64 height=64></img>
82       <% } else { %>
83         <img src="<%=CSEUtil::urlEncode((*i)["cover"]->operator string())%>"
              width=64 height=64></img>
84       <% } %>
85       </a></td>
86       <td><a href="search.csp?artist=<%=CSEUtil::urlEncode(thisArtist)%>">
            <%=thisArtist%></a><br>
87        <a href="record.csp?artist=<%=CSEUtil::urlEncode(thisArtist)%>&
             record=<%=CSEUtil::urlEncode(thisRecord)%>">
             <%=thisRecord%></a><br>
88        <a href="search.csp?label=<%=CSEUtil::urlEncode(thisLabel)%>">
             <%=thisLabel%></a><br>
89        <%=(*i)["year"]%><br>
90       </td>
91       <td align=right>
92        <%
93          ostrstream os;
94          os.setf(ios::fixed);
95          os.precision(2);
96          os << (*i)["price"]->operator float() << ends;
97        %>
98        EUR <b><big><%=os.str()%></big></b>
99       </td>
100     </tr>
101   <% } %>
```

This loop displays the records resulting from the search request and the page number. A result set is displayed as a row in a table with three columns. The first column (lines 79–85) contains the record's cover (as in `top.csp`) as a small image. The second column (lines 86–90) contains the artist and record name, the label, and the production year. Artist name and label are printed as links that can be used for starting new searches. The link behind the record name leads to the detailed record description page presented in the next section. The last column of the search result table shows the record's price. It is first written to an `ostrstream` to obtain the proper format for a price.

```
102   <% if (r.size() > 0) { %>
103     <tr>
104      <td colspan=3>
105       <table width=100%>
106        <tr>
107         <td>
108          <% if (pg > 1) { %>
```

```
            <a href="search.csp?artist=<%=CSEUtil::urlEncode(artist)%>&
               record=<%=CSEUtil::urlEncode(record)%>&
               label=<%=CSEUtil::urlEncode(label)%>&
               page=<%=pg - 1%>">&lt;&lt; Previous 10</a>
          <% } %>
         </td>
         <td align=right>
          <% if (pg - 1 < (r.size() - 1)/10) { %>
            <a href="search.csp?artist=<%=CSEUtil::urlEncode(artist)%>&
               record=<%=CSEUtil::urlEncode(record)%>&
               label=<%=CSEUtil::urlEncode(label)%>&
               page=<%=pg + 1%>">Next 10 &gt;&gt;</a>
          <% } %>
         </td>
        </tr>
       </table>
      </td>
     </tr>
   <% } %>
  </table>
 </body>
</html>
```

The rest of the servlet is used to print, if necessary, links to the preceding and succeeding page of the search request. This is done by modifying the page parameter of the search servlet.

6.3.3 record.csp and *.jpg

```
<%$store%>
<%#cse/db/default.h%>
<%#iomanip%>
<%#strstream%>
<%@session= false;%>
<%

  string artist, record;
  if (rq.getParameters().find("artist") != rq.getParameters().end())
    artist= rq.getParameters().find("artist")->second;
  if (rq.getParameters().find("record") != rq.getParameters().end())
    record= rq.getParameters().find("record")->second;

  DefaultDB db("record", "localhost", "record");
  DefaultDBQuery q(db);
  q << "select artist.artist, record.record, type, record.genre,
     record.label, record.producer, artist_desc, record_desc, year, price,
     record.cover from artist, record where artist.artist=record.artist"
    << " and artist.artist=\"" << artist << '"'
    << " and record=\"" << record << "\";"
    << ends;
  DefaultDBResponse r(q);
  DefaultDBIterator i(r);

%>
```

This servlet needs the `iostream` and `strstream` C++ standard header files. It does not require session management. The servlet takes the two HTTP request parameters `artist` and `record`, which uniquely identifies a record. These parameters are read in lines 8–12.

Before doing any output, the servlet establishes a database connection and extracts the required information about the record from the tables `artist` and `record`.

```
24
25 <html>
26  <head>
27   <title>Record Description - Record Store</title>
28  </head>
29  <body>
30   <% if (r.size() == 0) { %>
31    <h1>Not Found!</h1>
32    <p>Sorry, but the record you requested could not be found within our
          database.</p>
33   <% } else { %>
```

This code first prints a common title. Then, an error message is generated if the title has not been found. Else, the code below is used to generate the page.

```
34    <table>
35     <tr>
36      <td width=128>
37       <% if ((*i)["cover"]->isNull()) { %>
38         <img src="noimage.jpg" width=128 height=128></img>
39       <% } else { %>
40         <img src="<%=CSEUtil::urlEncode((*i)["cover"]->operator string())%>"
              width=128 height=128></img>
41       <% } %>
42      </td>
43      <td width=32></td>
44      <td width=*><h1><%=artist%></h1><h1><%=record%></h1></td>
45     </tr>
46     <tr height=32></tr>
47    </table>
48    <table width=70%>
49     <tr valign=top>
50      <td width=40%>
51       <h2>The <%=(*i)["type"]%>:</h2>
52       <p><%=(*i)["artist_desc"]%></p>
53      </td>
54      <td width=10%></td>
55      <td>
56       <h2>The Album:</h2>
57       <p><%=(*i)["record_desc"]%></p>
58      </td>
59     </tr>
60     <tr height=32></tr>
61    </table>
```

Lines 34–47 generate the record's cover on the top left of the page, with artist and title in headers to the right of the cover. The cover is generated exactly like in `top.csp`, by reading the name of the graphic file out of the database or, if the record contains no such information, by displaying `noimage.jpg`.

Below the cover, two columns inform the reader about the artist and about the record. They are read out of the database, which means that these columns may contain formatting instructions in HTML code.

```
62    <table width=70%>
63     <tr valign=top>
64      <td width=67%>
65       <h2>Details:</h2>
66       <p><b><%=artist%> - <%=record%></b><br>
67        Genre: <%=(*i)["genre"]%><br>
68        Label: <a href="search.csp?label=<%=CSEUtil::urlEncode(
             (*i)["label"]->operator string())%>"><%=(*i)["label"]%></a><br>
69        Producer: <%=(*i)["producer"]%><br>
70        Production Date: <%=(*i)["year"]%><br>
71        Price: <b><big>EUR <%
72          ostrstream os;
73          os.setf(ios::fixed);
74          os.precision(2);
75          os << (*i)["price"]->operator float() << ends;
76          re << os.str();
77        %></big></b></p>
78       <p><big><b>&gt;&gt; <a href="cart.csp?action=add&
            artist=<%=CSEUtil::urlEncode(artist)%>&
            record=<%=CSEUtil::urlEncode(record)%>">Add to cart</a>
            &lt;&lt;</b></big></p>
79       <p><big><b><a href="cart.csp">View shopping cart</a></b></big></p>
80      </td>
```

This piece of CSP code gives an overview on most relevant data extracted out of the database. It shows the artist and record name again, together with the genre, the label, the producer, the production date, and the price. The price is again generated using a `strstream` class like in `search.csp`. The record label is printed as a link which allows searching for other records by the same label.

Below this information, line 78 creates a link that adds this record to the shopping cart. This is done by calling `cart.csp` with the parameters `artist` and `record`. The next link, in line 79, calls the shopping cart servlet without any parameters, which will display the cart's current status.

```
81      <td width=*>
82       <h2>Track List:</h2>
83        <ol>
84        <%
85          DefaultDBQuery q2(db);
86          q2 << "select track, name, length, sample from track"
87            << " where artist=\"" << artist << '"'
88            << " and record=\"" << record << '"'
89            << " order by track"
90            << ends;
91          DefaultDBResponse r2(q2);
92          for (DefaultDBIterator i2(r2); i2 != r2.end(); ++i2) {
93            %>
94            <% if ((*i2)["sample"]->isNull()) { %>
95              <li><%=(*i2)["name"]%></li>
96            <% } else { %>
97              <li><a href="<%=CSEUtil::urlEncode((*i2)["sample"]->
                 operator string())%>"><%=(*i2)["name"]%></a></li>
```

```
 98           <% } %>
 99           <%--<td><%=(*i2)["length"]%></td>--%>
100           <%
101         }
102         if (r2.size() == 0) {
103           %>
104           <i>Not available</i>
105           <%
106         }
107       %>
108      </ol>
109     </td>
110    </tr>
111   </table>
```

This code fragment generates, to the right of the detailed information, a listing of the tracks on the record. It queries the `track` database table and iterates over the results, using the HTML tag `<ol>` for generating the track numbers. This means that listings of records where e.g. track 13 is missing (yes, there are some) need to include the missing tracks in the database without a description. Else, the track numbers will be different from those on the record. Line 99 adds an optional length information to the track. If no track information has been stored in the database for this record, line 104 is printed instead.

Within the Record Store, it is also possible to provide downloadable audio files including e.g. a fraction of a track on the record to help customers with their decision. This is done by putting the file on the web server and entering the file name into the database table `track`, column sample. If this information exists for a given track, the track information is printed as a link to that file.

```
112   <table>
113    <tr>
114     <%
115       DefaultDBQuery q3(db);
116       q3 << "select pos, image from image"
117         << " where artist=\"" << artist << '"'
118         << " and record=\"" << record <<  '"'
119         << " order by pos;" << ends;
120       DefaultDBResponse r3(q3);
121       for (DefaultDBIterator i3(r3); i3 != r3.end(); ++i3) {
122         %>
123         <td>
124          <img src="<%=CSEUtil::urlEncode((*i3)["image"]->
                operator string())%>"</img>
125         </td>
126         <td width=32></td>
127         <%
128       }
129     %>
130    </tr>
131   </table>
```

This code optionally places additional images below the record information. The paths to these images are read out of the database table `image`.

```
132    <p><b><big><a href="search.csp?artist=<%=CSEUtil::urlEncode(artist)%>">
          Find other records by the same artist</a></big></b></p>
133   <% } %>
134  </body>
135 </html>
```

The bottom of this servlet contains a link to search for other titles by the same artist. The closing brace in line 133 closes the else branch opened in line 33 (the code between these lines is only executed if the record exists).

6.3.4 cart.csp

```
  1 <%$store%>
  2 <%#cse/db/default.h%>
  3 <%#cstdlib%>
  4 <%#ctime%>
  5 <%#strstream%>
  6 <%#unistd.h%>
  7 <%@session= true;%>
  8 <%@sessionType= DatabaseSession;%>
  9 <%!time_t last;%>
 10 <%@last= time(NULL);%>
 11
 12 <%!void cleanup() throw () {
 13   if (time(NULL) - last >= CSE::GC) try {
 14     last= time(NULL);
 15     DefaultDB dbCart("record", "localhost", "record");
 16     DefaultDB dbSess;  // default CSE database, for session information
 17     DefaultDBQuery qCart(dbCart);
 18     qCart << "select session from cart;" << ends;
 19     DefaultDBResponse rCart(qCart);
 20     for (DefaultDBIterator iCart(rCart); iCart != rCart.end(); ++iCart) {
 21       DefaultDBQuery qSess(dbSess);
 22       qSess << "select id from session"
 23         << " where name=\"" << CSE::SNAME << '"'
 23a        << " and sandbox=\"store\""
 24         << " and id=" << (*iCart)["session"] << ';'
 25         << ends;
 26       DefaultDBResponse rSess(qSess);
 27       if (rSess.size() == 0) {
 28         ostrstream os;
 29         os << "delete from cart"
 30           << " where session=" << (*iCart)["session"] << ';'
 31           << ends;
 32         dbCart.execute(os.str());
 33       }
 34     }
 35   } catch (exception e) { }  // just in case...
 36 }%>
```

This servlet uses the C++ standard library `strstream` as well as the C standard libraries `stdlib`, `time`, and `unistd`. Until now, session management has not been necessary, but the shopping cart needs to be generated with each user's individual content, so lines 7–8 cause the session manager to provide a `DatabaseSession` for the servlet.

Line 9 declares the variable `last`, which is used as a timestamp for the last invocation of the shopping cart's garbage collection algorithm. At the creation of the servlet, it is set to the current time (line 10). A separate garbage collection algorithm is necessary for this servlet because we do not want to store all shopping cart data within the session database. The shopping cart is part of the Record Store, so its contents are stored there. This design also facilitates a possible future adaption which would reserve a record as long as it is within a shopping cart.

The garbage collection algorithm is contained within the definition of the `cleanup()` function. It iterates over all entries of the database table `cart`, retrieves the session IDs corresponding to the individual shopping carts, and looks them up in the CSE's own session database. If no corresponding session exists any longer, the shopping cart is regarded as obsolete and deleted from the database. The shopping cart garbage collector is executed at most as often as the CSE's own garbage collector (it accesses the variable `CSE::GC` defined in `include/cse/defaults.h`). On the other hand, if the shopping cart servlet is not requested for some time, it does not consume any resources.

```
37
38 <%
39
40   cleanup();
41
42   if ((session != NULL) && !session->isFromCookie())
43     re.cookies.push_back(session->asCookie());
44
45   DefaultDB db("record", "localhost", "record");
46
47   string action, artist, record, amount;
48   rq.getParameter("action", action);
49   rq.getParameter("artist", artist);
50   rq.getParameter("record", record);
51   rq.getParameter("amount", amount);
```

When the servlet is requested, its `cleanup()` function is called first. Lines 42–43 generate a cookie containing the session ID if the current session has not been retrieved using a cookie. Within the Record Store, session management relies on cookies, and no alternative way to pass the session ID is used. Line 45 creates a database connection, and lines 47–51 read some parameters that the servlet uses.

```
52
53   bool success= true;
54
55   try {
56     if (action == "alter") {
57       ostrstream os;
58       os << "replace into cart values("
59         << session->getId() << ", "
60         << '"' << artist << "\", "
61         << '"' << record << "\", "
62         << atoi(amount.c_str()) << ");"
63         << ends;
64       db.execute(os.str());
65     } else if (action == "delete") {
66       ostrstream os;
67       os << "delete from cart"
68         << " where session=" << session->getId()
69         << " and artist=\"" << artist << '"'
```

```
 70          << " and record=\"" << record << "\";"
 71          << ends;
 72        db.execute(os.str());
 73      }
 74    } catch (exception) { success= false; }
```

This code processes two commands, initiated by `action=alter` and `action=delete`. The first command (lines 56–64) sets the amount of copies of a record within the shopping cart. If no such item currently exists, it is added to the cart, else the amount is set to the given value. In either case, the amount is set by the parameter `amount`. The second command (lines 65–73) deletes an item from the shopping cart.

The flag defined in line 53 is used for determining the success of the operation. If an exception occurs (which usually indicates that an SQL command failed), the flag is cleared, which will result in an error message. This error message will be written below the shopping cart's contents.

```
 75
 76    DefaultDBQuery q(db);
 77    q << "select * from cart"
 78      << " where session=" << session->getId() << ';'
 79      << ends;
 80    DefaultDBResponse r(q);
 81
 82    float sum= 0.0f;
 83
 84  %>
```

Lines 76–80 retrieve data from the database table `cart` in which the shopping carts are stored. Line 82 initializes the `sum` variable which will keep the amount of money to be payed for the cart's contents at check-out. The sum is calculated while processing these data.

```
 85
 86  <html>
 87   <head>
 88    <title>Shopping Cart - Record Store</title>
 89   </head>
 90   <body>
 91    <h1>Your Shopping Cart</h1>
 92    <% if (r.size() == 0) { %>
 93      <p><b><i>Your shopping cart is empty.</i></b></p>
 94    <% } else { %>
 95      <table width=70% border=1>
 96       <tr>
 97        <th width=35%>Artist</th>
 98        <th width=35%>Record</th>
 99        <th width=10%>Amount</th>
100        <th width=10%>Price</th>
101        <th>Delete</th>
102       </tr>
```

If the shopping cart does not contain any items, line 93 states that fact. Else, a table is created starting with line 95. This table will contain artist and record name, the amount of copies, the price of these copies, and a link to remove the item again.

```
<% for (DefaultDBIterator i(r); i != r.end(); ++i) {
  string thisArtist((*i)["artist"]->operator string());
  string thisRecord((*i)["record"]->operator string());
  %>
  <tr>
   <td><a href="search.csp?artist=<%=CSEUtil::urlEncode(
      thisArtist)%>"><%=thisArtist%></a></td>
   <td><a href="record.csp?artist=<%=CSEUtil::urlEncode(thisArtist)%>&
      record=<%=CSEUtil::urlEncode(thisRecord)%>"><%=thisRecord%>
      </a></td>
   <td align=right>
    <form action="cart.csp" method=get>
     <input type=hidden name="action" value="alter"></input>
     <input type=hidden name="artist" value="<%=thisArtist%>"></input>
     <input type=hidden name="record" value="<%=thisRecord%>"></input>
     <input type=text size=3 name="amount"
        value="<%=(*i)["amount"]%>"></input>
    </form>
   </td>
   <td align=right>
    <%
      DefaultDBQuery q2(db);
      q2 << "select price from record"
        << " where artist=\"" << thisArtist << '"'
        << " and record=\"" << thisRecord << "\";"
        << ends;
      DefaultDBResponse r2(q2);
      if (r2.size() == 1) {
        float price= (*i)["amount"]->operator int()
          *(*DefaultDBIterator(r2))["price"]->operator float();
        sum+= price;
        ostrstream os;
        os.setf(ios::fixed);
        os.precision(2);
        os << price << ends;
        re << os.str();
      }
    %>
   </td>
   <td><a href="cart.csp?action=delete&
      artist=<%=CSEUtil::urlEncode(thisArtist)%>&
      record=<%=CSEUtil::urlEncode(thisRecord)%>">Delete</a></td>
  </tr>
<% } %>
  <tr>
   <td colspan=2><big><b>End Sum</b></big></td>
   <td colspan=2 align=right><big><b>EUR
      <%
        ostrstream os;
        os.setf(ios::fixed);
        os.precision(2);
        os << sum << ends;
        re << os.str();
      %>
```

```
151         </b></big></td>
152         <td>&nbsp;</td>
153       </tr>
154      </table>
155    <% } %>
```

This loop iterates over the items within the shopping cart. For each item, it generates a line in the table created above. The columns `artist` and `record` are created as links to a search for the artist and to the record description. The column `amount` is generated as a little HTML form with only a single visible field (lines 111–116). This makes it possible to overwrite it with a new amount.

The price is selected from the database. As long as the item remains within the shopping cart, price changes will still be recognized. Only after check-out the price must remain fixed. Line 129 adds the current row's price to the end sum, and line 138 adds a link to remove the current item. The end sum itself is printed in lines 141–153.

```
156    <% if (action == "add") { %>
157      <h2>Add the following product to your shopping cart:</h2>
158      <table border=1 width=70%>
159       <tr>
160        <th width=40%>Artist</th>
161        <th width=40%>Record</th>
162        <th width=10%>Amount</th>
163        <th>Add</th>
164       </tr>
165       <form action="cart.csp" method=get>
166        <input type=hidden name="action" value="alter"></input>
167        <input type=hidden name="artist" value="<%=artist%>"></input>
168        <input type=hidden name="record" value="<%=record%>"></input>
169        <tr>
170         <td><%=artist%></td>
171         <td><%=record%></td>
172         <td align=right><input type=text name="amount" value="1"
               size=3></input></td>
173         <td align=center><input type=submit value="Add!"></input></td>
174        </tr>
175       </form>
176      </table>
```

This code fragment prints an item to be added to the shopping cart. It allows to set an amount before actually processing that requests.

```
177    <% } else if (action == "alter") { %>
178      <% if (success) { %>
179        <p><b>Your shopping cart has been updated.</b></p>
180      <% } else { %>
181        <p><b>The operation was not successful.</b></p>
182        <p>The request your browser made has failed.<br>
183         Please go back and try again.</p>
184      <% } %>
185    <% } %>
186    <p><b><big><a href="buy.csp">&gt;&gt; Check Out &lt;&lt;</a></big></b></p>
187   </body>
188 </html>
```

Below the shopping cart's contents, the result of an `alter` action is written in lines 177–185 by evaluating the `success` flag defined in line 53. Finally, a link to check out and order the contents of the shopping cart is printed in line 186.

6.3.5 `login.html` and `user.csp`

Before a user can check out, the server needs to know some data such as the delivery address. User data is managed by the HTML form `login.html` and the servlet `user.csp`. The form is displayed when a user attempts to check out without being logged in, which means that the log in is delayed as long as possible.

```
 1 <html>
 2  <head>
 3   <title>Log In - Record Store</title>
 4  </head>
 5  <body>
 6   <h1>Log In</h1>
 7   <p>If you already have an account at the Record Store, please log in here.
        <br>
 8    Otherwise, please <a href="user.csp">create one</a>.</p>
 9   <form action="user.csp" method=post>
10    <input type=hidden name="action" value="login"></input>
11    <table>
12     <tr>
13      <td align=right>Username:</td>
14      <td><input type=text size=16 name="username"></input></td>
15     </tr>
16     <tr>
17      <td align=right>Password:</td>
18      <td><input type=password size=16 name="password"></input></td>
19     </tr>
20     <tr>
21      <td colspan=2><input type=submit value="Log In"></input></td>
22     </tr>
23    </table>
24   </form>
25  </body>
26 </html>
```

Both the link in line 8 and the form in lines 9–24 call `user.csp`. If it is called without parameters, this servlet anticipates a new user. If called with a username and a password, it attempts to authenticate the user. The servlet is printed below:

```
<%$store%>
<%#cse/db/default.h%>
<%@session= true;%>
<%@sessionType= DatabaseSession;%>
<%

  if ((session != NULL) && !session->isFromCookie())
    re.cookies.push_back(session->asCookie());

  DefaultDB db("record", "localhost", "record");
  string action, user;
  rq.getParameter("action", action);
```

```
13   if ((action != "login") && (action != "create") && (action != "adjust"))
14     action= "";
15   session->getParameter("user", user);
16
17   bool valid= true;  // until something bad happens...
18   string username, password, password2, first, last, address, email;
19
20 %>
```

The servlet will be placed into the sandbox `store`, requires the database header files, and uses a session that is stored within the database. Since this servlet is stored in the same sandbox as `cart.csp` and `buy.csp`, sessions created for one of the servlets will be used by the others as well. Lines 7–8 again generate a cookie with the session ID if necessary.

The lines 10–15 parse some possible parameters. Three `action`s are recognized: logging in, creating a user, and adjusting user data (which is not implemented in this example). The `user` parameter is read out of the current session. It is found if a user has already been authenticated.

```
21
22 <html>
23  <head>
24   <title>User Management - Record Store</title>
25  </head>
26  <body>
27   <% if (action == "login") { %>
28     <h1>Log in</h1>
29     <%
30       rq.getParameter("username", username);
31       rq.getParameter("password", password);
32       DefaultDBQuery q(db);
33       q << "select first from user"
34         << " where uname=\"" << username << '"'
35         << " and pword=password(\"" << password << "\");"
36         << ends;
37       DefaultDBResponse r(q);
38     %>
39     <% if (r.size() == 0) { %>
40         <p><b><big>Login failed!</big></b></p>
41         <p>We could not find your username/password in our database.
42          Please <a href="login.html">try again</a> with the right
               username/password.</p>
43         <p>If you do not have an account yet, please
              <a href="user.csp">create one</a>.</p>
44     <% } else { %>
45       <% session->setParameter("user", username); %>
46       <p><b><big>Login successful.</big></b></p>
47       <p>Welcome to the Record Store,
            <%=(*DefaultDBIterator(r))["first"]%>!</p>
48       <p><b><big>&gt;&gt; <a href="cart.csp">Shopping Cart</a>
            &lt;&lt;</big></b></p>
49     <% } %>
50   <% } %>
```

This part of the servlet handles the login process. The parameters `username` and `password` are compared to the username and the encrypted password within the database. If the entry for the given user is found, the parameter `user` within the current session is set to the given username.

This means that from this point onward, the session ID contained within a cookie at the client's host will be the user's authentication token, and to guess a valid username is not enough to be able to gain unauthorized access.

```
<% if (((action == "create") || (action == "")) && (user == "")) { %>
  <h1>Create Your Account</h1>
  <% if (action == "create") { %>
    <p>The following problems were discovered processing your account
       data:</p>
    <ul>
     <%

       // get parameters:
       rq.getParameter("username", username);
       rq.getParameter("password", password);
       rq.getParameter("password2", password2);
       rq.getParameter("first", first);
       rq.getParameter("last", last);
       rq.getParameter("address", address);
       rq.getParameter("email", email);

       // check username:
       if (username == "") {
         valid= false;
         re << "<li><font color=800000>Please enter a username.</font>
           </li>" << endl;
       } else {
         DefaultDBQuery q(db);
         q << "select * from user"
           << " where uname=\"" << username << "\";"
           << ends;
         DefaultDBResponse r(q);
         if (r.size() != 0) {
           valid= false;
           re << "<li><font color=800000>The username you specified is
              already in use.<br>Please enter another username.</font>
              </li>" << endl;
         }
       }

       // check password:
       if (password == "") {
         valid= false;
         re << "<li><font color=800000>Please enter a password to protect
            your account.</font></li>" << endl;
       }
       if (password != password2) {
         valid= false;
         re << "<li><font color=800000>The passwords you have entered
            differ.<br>Please enter them again.</font></li>" << endl;
       }

       // check name:
       if (first == "") {
        valid= false;
        re << "<li><font color=800000>Please enter the first name.</font>
           </li>" << endl;
      }
      if (last == "") {
        valid= false;
        re << "<li><font color=800000>Please enter the last name.</font>
           </li>" << endl;
      }

      // check address:
      if (address == "") {
        valid= false;
        re << "<li><font color=800000>Please enter your address.</font>
           </li>" << endl;
      }
      if (email == "") {
        valid= false;
        re << "<li><font color=800000>Please enter your e-mail address.
           </font></li>" << endl;
      }

      if (valid) re << "None - Thank you for your cooperation!" << endl;
      session->setParameter("user", username);
      ostrstream os;
      os << "insert into user values("
        << '"' << username << "\", "
        << "password(\"" << password << "\"), "
        << '"' << first << "\", "
        << '"' << last << "\", "
        << '"' << address << "\", "
        << '"' << email << "\");"
        << ends;
      db.execute(os.str());

    %>
  </ul>
<% } %>
<% if ((action == "create") && valid) { %>
  <h2>You are now logged in as <%=username%>.</h2>
  <p><b><big>&gt;&gt; <a href="cart.csp">Shopping Cart</a>
     &lt;&lt;</big></b></p>
<% } %>
```

First, if the user has already entered some data in the servlet, they are passed through a simple verification process. The username must be entered, and it must not correspond to any existing username within the database. The password must be entered twice, and the two strings must be identical. First and last name as well as postal and e-mail address must be entered. If everything has been verified and no error has occurred, the user is created and his data stored within the database.

```
<% if (action == "") { %>
  <p>Please fill in the following form.<br>
   We need that data in order to be able to process your requests.<br>
```

```
      Everything you enter here will be kept strictly confidential.</p>
    <% } %>
    <% if ((action == "") || !valid) { %>
     <form action="user.csp" method=post>
      <input type=hidden name="action" value="create">
      <table>
       <tr valign=top>
        <td align=right>Username: </td>
        <td><input type=text size=32 name="username"
           value="<%=username%>"></input></td>
        <td><i>You can choose any username you like.<br>
          If a username is already in use, you will be notified in the
             next step.
        </i></td>
       </tr>
       <tr valign=top>
        <td align=right>Password: </td>
        <td><input type=password size=32 name="password"
           value="<%=password%>"></input></td>
        <td><i>Choose a password.<br>
          This password identifies you on our server, so make sure that it
             is hard to guess.<br>
          A good password is at least 8 characters long and contains
             UPPERCASE and lowercase characters, numbers, and special
             characters<br>
          Example (please don't use this one):
             &quot;Sr6&amp;zxpUi&gt;&quot;
        </i></td>
       </tr>
       <tr valign=top>
        <td align=right>Password: </td>
        <td><input type=password size=32 name="password2"
           value="<%=password2%>"></input></td>
        <td><i>Re-type your password here.<br>
          Since the password is not displayed, this is your way of making
             sure you really know the password.
        </i></td>
       </tr>
       <tr valign=top>
        <td align=right>First Name: </td>
        <td><input type=text size=32 name="first"
           value="<%=first%>"></input></td>
       </tr>
       <tr valign=top>
        <td align=right>Last Name: </td>
        <td><input type=text size=32 name="last"
           value="<%=last%>"></input></td>
       </tr>
       <tr valign=top>
        <td align=right>Address: </td>
        <td><textarea name="address" rows=5 cols=32><%=address%>
           </textarea></td>
        <td><i>We need your address in order to send you your purchases.
           <br>
           Please include street, ZIP code, and country.<br>
           In short, a packet with that address on it should be able to
              reach you from anywhere</i></td>
        </tr>
        <tr valign=top>
         <td align=right>E-Mail: </td>
         <td><input type=text size=32 name="email" value="<%=email%>">
           </input></td>
         <td><i>We need your e-mail address to be able to contact you in
            case there are problems with your order.</i></td>
        </tr>
       </table>
       <table>
        <tr>
         <td><b><big>&gt;&gt;</big></b></td>
         <td><input type=submit value="Create!"></input></td>
         <td><b><big>&lt;&lt;</big></b></td>
        </tr>
       </table>
      </form>
    <% } %>
  <% } %>
```

This code generates a form that registers the user's data. This form cannot be generated by a static HTML document because it is needed again in the case that some values are invalid. The form itself has three columns. The first column contains the name of the field, the second one the field for entering data, and the third column contains a helpful message for the user.

```
  <% if (action == "adjust") { %>
   <h1>Here, you will soon be able to modify your user settings.</h1>
  <% } %>
 </body>
</html>
```

Adjusting user data after entering it is not implemented in this simple example.

6.3.6 `buy.csp` and `business.html`

The HTML document `business.html` displays the business agreement for the Record Store, in our example a text stating that this is example code and no real webstore.

```
<html>
 <head>
  <title>Business Agreement - Record Company</title>
 </head>
 <body>
  <h1>The Record Company</h1>
  <h2>Business Agreement</h2>
  <ul>Please read this agreement carefully, you are bound by it when you
     order our products.</ul>
  <p>The <b>Record Company</b> web store is no real web store, it is just a
     demo application for the
     <a href="http://stud3.tuwien.ac.at/~e9626231/cse/">C++ Servlet
     Environment</a>.
  <p>No products you order will be delivered, and no money will be deposited
     from your credit card.
   Your credit card information will not be checked, and it will not be
      protected in any way, so please use a fake credit card number.</p>
  </body>
</html>
```

The servlet `buy.csp` is more interesting:

```
 1 <%$store%>
 2 <%#cse/db/default.h%>
 3 <%#strstream%>
 4 <%@session= true;%>
 5 <%@sessionType= DatabaseSession;%>
 6 <%
 7
 8   if ((session != NULL) && !session->isFromCookie())
 9     re.cookies.push_back(session->asCookie());
10
11   string user, action;
12   session->getParameter("user", user);
13   rq.getParameter("action", action);
14
15   DefaultDB db("record", "localhost", "record");
16   DefaultDBQuery q(db);
17   q << "select * from user"
18     << " where uname=\"" << user << "\";"
19     << ends;
20   DefaultDBResponse r(q);
21   DefaultDBIterator i(r);
22
23   bool valid= true;
24   float price= 0.0f;
25
26 %>
```

This servlet again uses the database header files as well as the `strstream` class (for formatting the prices as well as for formulating SQL statements). It uses the same sessions as `cart.csp` and `user.csp`. This session is written to a cookie, if necessary, in lines 8–9. In lines 15–21, an iterator is prepared that will be used to again verify the existence of the username within the database. The flag `valid` (line 23) is set to false as soon as any problem is encountered, which will immediately stop any further processing. The variable `price` will be used to calculate the sum of the items to be bought, the same way the `cart.csp` servlet does.

```
27
28 <html>
29  <head>
30   <title>Make A Purchase - Record Store</title>
31  </head>
32  <body>
33   <% if ((user == "") || (r.size() == 0)) { %>
34     <% session->setParameter("user", ""); %>
35     <h1>Please log in!</h1>
36     <p>You didn’t log in yet.<br>
```

```
37      In order to make a purchase, you must log in, because we need e.g.
           your address to be able to send you your records.</p>
38     <p><b><big>&gt;&gt; <a href="login.html">Log In</a>
          &lt;&lt;</big></b></p>
```

The above code fragment tells the user, if he is not yet logged in, to proceed to the login form (`login.html`).

```
39   <% } else if (action == "order") { %>
40     <%
41       string ccard, cnumber, cdate;
42       rq.getParameter("ccard", ccard);
43       rq.getParameter("cnumber", cnumber);
44       rq.getParameter("cdate", cdate);
45       DefaultDBQuery qCart(db);
46       qCart << "select record.artist, record.record, cart.amount,
            record.price, record.store from cart, record"
47         << " where record.artist=cart.artist"
48         << " and record.record=cart.record"
49         << " and cart.session=" << session->getId() << ';'
50         << ends;
51       DefaultDBResponse rCart(qCart);
```

This code retrieves some parameters and then again iterates over the items within the shopping cart.

```
52       for (DefaultDBIterator iCart(rCart); iCart != rCart.end(); ++iCart) {
53         ostrstream in;
54         in << "insert into request values ("
55           << '"' << user << "\", "
56           << '"' << (*iCart)["artist"] << "\", "
57           << '"' << (*iCart)["record"] << "\", "
58           << (*iCart)["amount"] << ", "
59           << (*iCart)["price"] << ", "
60           << '"' << ccard << "\", "
61           << '"' << cnumber << "\", "
62           << '"' << cdate << "\", "
63           << "NULL);"
64           << ends;
65         db.execute(in.str());
```

Line 52 starts a loop over all items within the shopping cart. The SQL insert statement below the loop head stores the request in the `request` table of the Record Store database. The price is also stored because the customer has to pay the price on his bill and is not affected by later changes in the store.

```
66         ostrstream out;
67         out << "update record"
68           << " set store=store-" << (*iCart)["amount"]
69           << " where artist=\"" << (*iCart)["artist"] << '"'
70           << " and record=\"" << (*iCart)["record"] << "\";"
71           << ends;
72         db.execute(out.str());
78       }
73       ostrstream del;
```

```
    del << "delete from cart"
      << " where session=" << session->getId() << ';'
      << ends;
    db.execute(del.str());
  %>
  <h1>Thank you for your order!</h1>
  <p>You will soon receive a packet from us.</p>
<% } else { %>
```

These statements update the number of items within the store and empty the shopping cart.

```
  <h1>Make A Purchase</h1>
  <p>Thank you for your interest in our products, <%=(*i)["first"]%>.</p>
  <p>Please verify your order once more (look for problems indicated in
     <font color=800000>red</font>).</p>
  <%
    DefaultDBQuery qCart(db);
    qCart << "select record.artist, record.record, cart.amount,
       record.price, record.store from cart, record"
      << " where record.artist=cart.artist"
      << " and record.record=cart.record"
      << " and cart.session=" << session->getId() << ';'
      << ends;
    DefaultDBResponse rCart(qCart);
    DefaultDBIterator iCart(rCart);
  %>
  <% if (rCart.size() == 0) { %>
    <p><font color=800000>You don't have articles in your
       <a href="cart.csp">shopping cart</a>.</font></p>
    <% valid= false; %>
  <% } else { %>
    <table border=1 width=70%>
     <tr>
      <th width=25%>Artist</th>
      <th width=25%>Record</th>
      <th width=10%>Amount</th>
      <th width=15%>Price</th>
      <th>Problems</th>
     </tr>
     <% for (; iCart != rCart.end(); ++iCart) { %>
       <tr>
        <td><%=(*iCart)["artist"]%></td>
        <td><%=(*iCart)["record"]%></td>
        <td align=right><%=(*iCart)["amount"]%></td>
        <td align=right>EUR
         <%
           float thisPrice= 0;
           try {
             thisPrice= (*iCart)["amount"]->operator int()
               *(*iCart)["price"]->operator float();
             price+= thisPrice;
           } catch (DBNullException) { }
           ostrstream os;
           os.setf(ios::fixed);
         os.precision(2);
         os << thisPrice << ends;
         re << os.str();
       %>
      </td>
      <td><font color=800000>
        <% if ((*iCart)["amount"]->operator int() == 0) { %>
          Amount must not be 0.
          <% valid= false; %>
        <% } else if ((*iCart)["store"]->operator int() == 0) { %>
          We are currently unable to deliver this record.
          Please try again later!
          <% valid= false; %>
        <% } else if ((*iCart)["amount"]->operator int() >
           (*iCart)["store"]->operator int()) { %>
          We only have <%=(*iCart)["store"]%> records left.
          <% valid= false; %>
        <% } else { %>
          &nbsp;
        <% } %>
      </font></td>
     </tr>
   <% } %>
   <tr>
    <td colspan=2><b><big>End Sum:</big></b></td>
    <td colspan=2 align=right><b><big>EUR
       <%
         ostrstream os;
         os.setf(ios::fixed);
         os.precision(2);
         os << price << ends;
         re << os.str();
       %>
    </big></b></td>
    <td>&nbsp;</td>
   </tr>
  </table>
<% } %>
```

If the purchase has not yet been ordered, the shopping cart's contents are once more printed, with problems indicated in red. The recognized problems are an empty shopping cart, an item of which 0 copies are ordered, and not enough copies left within the store. In each case, the `valid` flag is cleared.

```
<% if (valid) { %>
  <p>We found no problems concerning your shopping cart.</p>
  <p>Please check your delivery address:</p>
  <ul><pre><%=(*i)["first"]%> <%=(*i)["last"]%></pre></ul>
  <ul><pre><%=(*i)["address"]%></pre></ul>
  <p>Please check your e-mail address in case anything goes wrong:</p>
  <ul><%=(*i)["email"]%></ul>
  <p>Please fill in your credit card information:<br>
   We will keep this information <i>only</i> for processing that order,
      afterwards it will be purged from our databases.</p>
      <form action="buy.csp" method=post>
       <input type=hidden name="action" value="order"></input>
       <table>
        <tr>
         <td align=right>Credit Card Type:</td>
         <td>
          <select name="ccard" size=1>
           <option value="visa">Visa</option>
           <option value="mastercard">MasterCard</option>
           <option value="eurocard">EuroCard</option>
           <option value="aexpress">American Express</option>
          </select>
         </td>
        </tr>
        <tr>
         <td align=right>Credit Card Number:</td>
         <td><input type=text name="cnumber" size=16></input></td>
        </tr>
        <tr>
         <td align=right>Expiry Date:</td>
         <td><input type=text name="cdate" size=16></input></td>
        </tr>
       </table>
       <p>The button below gives us a <b>definite</b> order to process your
          request, so please make sure once again that everything is
          correct.</p>
       <p>By ordering you agree to our <a href="business.html">business
          agreement</a>.</p>
       <p><big><b>&gt;&gt; <input type=submit value="Buy It Now"></input>
          &lt;&lt;</b></big></p>
      </form>
```

If no errors have been found, a form asking for credit card information is printed. It contains a link to the business agreement and a button to order the displayed items.

```
    <% } else { %>
      <p>Please correct the error(s) and try again.</p>
      <p><b><big>&gt;&gt; <a href="cart.csp">Shopping Cart</a>
         &lt;&lt;</big></b></p>
    <% } %>
  <% } %>
 </body>
</html>
```

In the case of an error, an error message with a link back to the shopping cart is displayed. The user should then correct the errors and try again.

6.4 Summary

The Record Store is an exemplary webstore that already includes many features of its full-grown cousins. It has been implemented to show servlet programmers the possibilities of the C++ Servlet Environment. The Record Store provides a dynamic front page with featured articles, a search and browsing facility, a page containing detailed record information created on the fly from a database, a sophisticated shopping cart, a user data registration and authorization page, and a check-out page which registers and validates the orders.

However, some features of a typical web store are still missing. The search functionality lets users search for strings in different parts of the record description, but right now, only exact string matching is implemented. Substring search as well as search within the textual description of a record remain to be done.

Another problem is that records are not checked for availability or even reserved when they are put into the shopping cart. Availability is checked together with the validity of an order, before actually submitting the order. This makes a race condition possible that allows several users to order the last copy of a record, which however is visible in the `request` table of the Record Store database and can be resolved manually (either by informing some users that no more copies are in store, or by ordering more copies).

User authentication is implemented similar as in many other web applications available today, and therefore suffers from the same problems. Especially, a user who gains access to another user's session ID is not recognized as an intruder. This defect, however, only enables an intruder to view and modify the contents of the shopping cart (which is bound to be noticed), to view some user information like the e-mail address, and to pay for the user. It is currently impossible to view the complete user information, change the password, modify the destination address, or retrieve the credit card information (except in the same way as the session ID was retrieved, i.e. in transit). Another currently missing feature is the modification of the user data.

Of course, the necessary steps after placing an order (verifying the credit card information, retrieving the customer's money, and actually sending the order to the customer) are not implemented in this demonstration store. However, all the deficits mentioned here are rather easy to implement — for one, even some stub code within a servlet has been written. The further improvement of the Record Store, and probably the correction of some bugs, is left as an exercise to the reader.

Chapter 7

Related Work

This chapter first gives an overview on the existing concepts and introduces some concrete systems to illustrate them. We will also point out differences to the features the CSE provides. A performance evaluation of some of these systems will be presented in Chapter 8.

7.1 Other C++ Solutions

After we have started the development of the CSE, other developers have also recognized the need for a servlet environment written in C or C++. Their systems, however, cannot compete with the features of the CSE yet: All other systems lack its sandbox mechanism which protects the servlet server from unstable servlets and forms a barrier between individual web applications. A principal advantage of C++ servlet environments is that they can be configured to run different web applications using different system accounts in order to provide e.g. `~user`-type URLs. This is impossible as long as all servlets are executed within a single server. The C++ Servlet Environment uses a traits class which adapts a C/C++ database interface for use with the CSE, while other C++ systems currently either provide no persistence at all or work together with only a single database driver.

The Indian company Ape Software has developed *Servlet++* [Ape] under a BSD-like free license. This system still seems to be under development, and a number of typical features are currently missing. Among the most important is session management, a fundamental criterion for a servlet environment; due to this deficit, we did not include Servlet++ in our benchmarks in the next section. The servlet engine is contained completely within an Apache module, which may compromise the stability of the web server when developing servlets. The C++ servlet interface has been designed similar to its Java equivalent, so that servlets written for that system can be used with the CSE after relatively minor modifications. Simple C++ Server Pages are also supported (`<%code%>` and `<%@definition%>` tags are known), but they have to be compiled manually, first into C++ source code with an included compiler, then with the platform's C++ compiler. When a new servlet is added to the environment, the server needs to be re-started, which is not necessary with the CSE.

C Server Pages [Gon] has been designed with goals similar to those of the CSE, but it does not use a free license (commercial use is non-free). The servlet server is currently contacted via the CGI, which results in a low performance (measured in the next chapter). An Apache module is announced, but has not yet been available for download. This system does not support dynamic configuration like the CSE does — servlets as well as CSPs have to be compiled manually and work only after a restart of the server. Simple CSPs are supported, the server provides the tags `<#!header#>` (which writes preprocessor directives or definitions into the header of the C++ file) and `<#code#>`.

The commercial system *Micronovae C++ Server Pages* [Mica] works together with Microsoft's Internet Information Server. It provides support for the execution of CSPs (`<%!declaration%>`,

`<%code%>`, and `<%=expression%>` are implemented), but it does not support C++ servlets. Unfortunately, their download (still a beta version) contains no source code, so our information about this system is solely based on our experiences with it.

The commercial vendor Rogue Wave has developed *Bobcat* [Rog], a C++ servlet engine that has an API similar to that of the Java Servlet Specification [Sun01a]. Its evaluation license, however, contains a non-disclosure agreement. Hence, we cannot include their product in this paper and have to assume that it is not yet ready for a production system.

The *Weblet Application Server* [Web] seems to be a servlet engine for C++ developed by Webletworks. Unfortunately, we were unable to contact the web server of this company for several months now and were also unable to obtain a copy or other information about the system through other web sites.

7.2 Java Servlets and Java Server Pages

In their effort to make Java the default programming language for web applications, the company Sun Microsystems designed the Java Servlet API (application programmer's interface). It primarily defines a default interface for servlets in Java, and a more concrete interface for servlets using the HTTP protocol. Also included are classes for servlet requests and responses, for streams used by servlets, and methods for usual servlet related tasks. This interface greatly facilitated servlet development and is accepted today as a standard for servlet programming in Java [Sun01a].

Another API called Java Server Pages (JSP) [Sun01b] has been designed for servlets producing HTML. It consists of additional, non-HTML tags (the earlier ones using a syntax like `<%tag%>`, the later ones using the XML form `<jsp:tag>`). This code has to be changed into executable code by incorporating it into a skeleton servlet and expanding the JSP tags into Java code. This transformation is usually done at run time, followed by a compiler run. In Java, it is then relatively simple to load the newly-generated servlet into the server at run-time. The Java Servlet and Java Server Pages APIs (which were taken as a guideline for designing the CSE's equivalent interfaces) are currently used within most servlet environments and also, together with other server technologies, within the larger application servers.

The CSE's design is partly caused by the use of C++, where a crash (caused e.g. by a segmentation violation) terminates the execution of a whole task, as opposed to a thread in Java. Therefore, we introduced the sandboxes as boundaries between web applications. However, both Java systems introduced in this section configure complete web applications too. For persistent data, Java servlet environments typically use the JDBC interface so that it is as easy to switch the database in Java as it is with the CSE.

A list of Java servlet environments and application servers can be found at [Ser03] — at the time of writing, it contained 31 entries. In this thesis, we limit ourselves to presenting only two concrete implementations: the reference implementation (there used to be a reference implementation by Sun, but it has been replaced by Apache's more performant Tomcat system), and the commercial implementation Jetty by Mort Bay.

The Apache Jakarta *Tomcat* server (by the Apache Group's subproject Jakarta) [Apab] has become the official reference implementation for Java Servlets and Java Server Pages. Since it is written in Java, it runs on each platform to which the Java Development Kit has been ported. It contains its own web server, but it can also be used together with the Apache HTTP server. Web applications are configured by an XML file each, and their servlets and JSPs are put into a directory below `webapps`. Compiled Java code may be used by the servlets, either as a class file or as a Java archive. Tomcat can work with either Sun's byte-code compiler or third-party compilers like jikes [IBM]. Like the Apache HTTP Server, Tomcat is available under a free license (in the sense of [Deb]).

The Australian company Mort Bay has developed *Jetty*, which is also based on the Java Servlet and Java Server Pages standards. It works very similar to Tomcat, and (since they are both based on the same standard) can execute the same servlets without modification. Jetty also includes its own web server, and it can work both with a non-default web server and a non-default compiler. Jetty is available from [Mor] under a free license.

7.3 The Common Gateway Interface (CGI)

The Common Gateway Interface, usually referred to as CGI, has been the first concept for generating dynamic server applications. It is implemented within a web server, and is basic purpose is executing a script and delivering its output to the web server. A CGI script can be a shell script, a script in another scripting language, or an executable file (e.g. compiled C++ code).

When the web server identifies a CGI script (by its extension or because all CGI scripts belong to a common subdirectory), it first prepares an environment in which the script will be executed. Several environment variables are set to values describing the state of the HTTP server at the time of the request. For example, `QUERY_STRING` contains parameters passed to the server via a GET request. If the request contains a message body (e.g. a `POST` request), this body is placed on the scripts standard input stream. The servlet's output stream is passed to the client. The script must generate the `Content-Type` HTTP header as well as the body of the response.

A CGI script is probably the easiest way to connect an existing application to a web server. If the application's source code is available, it can be modified so that e.g. request parameters are parsed from the corresponding environment variable instead of reading them from the standard input stream, and the response can be modified so that a correct `Content-Type` header is generated. If the source code is not available, a wrapper script needs to be written for each application that needs to be placed on the web. This wrapper script has to transform the environment variables into valid application input, and the application output into a valid CGI response. Of course, this process can be applied for applications written in any programming language, and a CGI script can be used with any web server on a given platform. For C++, this technique can also be applied to the CSE.

On the other hand, the CGI is usually not the system of choice when developing a new application for the web. Shell variables may be a good means of communication for shell scripts, but usually there are easier ways in other programming languages. Generating a header allows flexible output, but requires more effort by the programmer. Session management is not implemented on CGI level, and scripts requiring it must generate and process session information on their own. Also, CGI scripts usually imply low performance. They are executed as separate processes, which conveys a constant overhead on the servlet's execution time. Database connections from CGI scripts have similar disadvantages because a database connection cannot be maintained over several invocations of the script, and transactions have to be done in a single request. Finally, CGI scripts are often low security solutions. In many cases we have seen, programmers include a database password within the script's code. If a malconfigured server does not execute the script for some reason, the source code with the plain-text password is transmitted to the client. Depending on the web server's configuration, a CGI script with a security hole may provide an attacker with the possibility to execute commands as the web server's user, i.e. it helps crackers who want to deface a web server. Of course the security problems can be solved, but inflexibility and low performance still make servlet environments like the CSE more attractive for serious web programmers.

If CGI scripts are executed as separate processes, unstable code within a servlet does not pose a stability problem, so mechanisms like the CSE's sandboxes are not necessary. When a CGI script crashes, the parent process (usually the web server) notices the crash and is able to generate an error message instead of the usual servlet output. CGI scripts usually maintain the web server's system account, which means that an intruder who finds a security hole can possibly modify the web server's content. Apache's `mod_suexec` extension allows executing CGI scripts under another user's account, which adds another layer of security to these scripts. For CGI scripts, there are no mechanisms that implement persistence. A script can use any database for which a driver in that scripting language has been written, but no standardized interface is provided.

The *Apache HTTP Server* [Apaa] includes a well-documented implementation of the Common Gateway Interface. It can be compiled into the web server itself, or as a loadable module which is called `mod_cgi`. The web server identifies CGI scripts by its MIME type (either `application/x-httpd-cgi` or `cgi-script`). This type can be set for individual file extensions or for whole directories. `mod_cgi` addresses the security problems introduced above with another (optional) module called `suexec`. It adds a wrapper around the CGI scripts that performs a number of validity checks and changes the user and group ID of the process before executing the script. This module, if used correctly, considerably lowers the possibility of web server defacement via insecure CGI scripts; on the other hand, it further decreases performance. Unfortunately, an incorrectly configured server will still deliver the script's source code, so the responsibility for server passwords is entirely up to the programmer.

Several new designs have been suggested which try to overcome the performance deficit of the CGI. The main principle of them is the re-use of a process for several requests, thus improving performance and allowing more efficient code. An implementation of this principle is *FastCGI* [Ope96], which adds another API to a CGI script. This API, which is available for several programming and scripting languages, contains functions for reading and writing the request and response streams as well as for accepting new requests. The system is an improvement over the standard CGI, and can be seen as the first step towards a servlet environment — indeed, some systems that call themselves servlet environments provide hardly more functionality than FastCGI.

7.4 Scripting Languages

As a logical improvement over CGI scripts, servlet environments have also been designed for (and written in) some scripting languages. They are much more complicated programs than mere CGI scripts. Still, they usually do not need special stability mechanisms like the CSE does because scripting languages require an interpreter. Since they do not run directly on the native hardware, the interpreter can easily catch an error within a servlet and continue execution along another path.

Because of the large number of servlet environments available, only a small number of them is presented in this section.

The *PHP Hypertext Processor* (a recursive acronym) [BAS$^+$02] provides a scripting language that can be included directly within HTML code, similar to C++ Server Pages. The language provides features for both functional and object-oriented programming, and it has been designed so that it is easy to get started. Because of its wide use PHP provides very much functionality, and it comes with a library interface which allows servlets to use third-party code. PHP is available for many platforms and works together with most web servers. However, PHP servlets which have been written for one platform may have to be modified when they are used on another one (e.g. because of different filesystem conventions).

The syntax of Microsoft's *Active Server Pages* [Micb] is very similar to that of Java Server Pages, except that ASPs may be written in any supported programming language. By default, Microsoft's VBScript and JScript are supported, but third-party modules allow the use of several other programming languages as well. The technique includes an interface to access COM components provided by the Windows platform or by the programmer. Like most Microsoft products, ASP — included in the Internet Information Server — is not licensed as free software [Deb]. The Internet Information Server itself is included in Microsoft's server operating systems and contains no source code.

For this thesis, we initially decided to include *Zope* [LP01], a servlet environment written for and in Python, in our benchmarks as an example system representing scripting languages. Unfortunately, Zope kept crashing when confronted with large numbers of requests, so that we could not execute all our benchmarks. The initial results, however, show that Zope is about an order of magnitude slower than the tested C++ and Java systems. This is probably caused by the performance penalty incurred by the Python interpreter. We assume that similar results hold for most other interpreted languages.

7.5 Different Solutions for Dynamic Web Applications

Although the usual procedure for bringing applications to the web is the extension of the web server, this is not the only possible way. SWILL [LB02] is a lightweight programming library that provides a simple web server for C and C++. Unlike a servlet environment, its goal is not to offer a full fledged server that allows the execution of multiple web applications; instead, it allows developers to embed the web server into their own programs. This web server can be used to control the embedding application and to display the application's results using a web browser.

Chapter 8

Performance Evaluation

This chapter discusses the performance of the C++ Servlet Environment in relation to that of several other systems. First, the CSE's parallelism is examined. Next, a benchmark suite for servlet environments is presented. Finally, we analyze the results of those benchmarks for the chosen systems.

8.1 Parallelism and Scalability

In the CSE's design, we have introduced parallelism on several levels. We have exploited existing parallelism within the web server and added it to the servlet server were this was reasonable.

The Apache HTTP Server contains a sophisticated mechanism that manages parallelism: At compile time, the network administrator can choose from a number of so-called Multi-Processing Modules, which each implement another approach to parallelism. For example, it is possible to make the server multi-threaded instead of using multi-tasking, or to use a combination of both, according to the destination platform. The web server has been thoroughly optimized for maximum performance over several years and thus should offer a good basis for a servlet environment.

The CSE itself currently implements parallelism mainly at the sandbox level, because this involves the least problems. Sandboxes as boundaries for web applications can be executed in parallel without any coordination, and without any dependencies. Even database requests from different sandboxes do not collide with each other since each sandbox is only interested in the data it created by itself.

The servlet server itself currently accepts requests serially, and delivers them as quickly as possible to the sandboxes. Also, different servlets within a sandbox are currently executed one at a time; partly because more parallelism would introduce undesirable conflicts, and partly because a servlet currently does not have to wait for any shared resources, which means that parallelism here can only bring new performance on a multi-processor system. In the end, we decided that the introduction of parallelism gains most and costs least at sandbox level. The amount of scalability arising from that decision can compare, as shown in the next section, with that of other current servlet environments.

8.2 Performance Benchmarks

For the evaluation of the C++ Servlet Environment's performance we have implemented a benchmark suite that tests various aspects of a servlet engine. The tests were performed with the built-in web server. All servlets were compiled using the individual engine's default servlet compiler before the execution of a benchmark. These benchmarks and their results have already been presented in [GS03].

We have included Tomcat as Sun's Java reference implementation, Jetty as an independent implementation in Java, and CSP and Micronovae as other C/C++ solutions. Since little information about the inner workings of Micronovae is available, we can only speculate why it performs better or worse than the other systems. We also wanted to include Zope, but this server regularly crashed when confronted with a large number of requests. At any rate, the results we managed to measure were worse than those of the other systems. Of the other C/C++ systems, we did not include Bobcat because of the non-disclosure-agreement necessary to download it, and we left out Servlet++ because it does not provide session management, a fundamental requirement for every servlet environment.

Static Page Access

In this benchmark we measure how long it takes to request a static HTML page together with 40 embedded images for 100 times. The primary goal of this benchmark is to measure the throughput of the servlet engine's web server.

Bulletin Board System

We have implemented a small bulletin board system that stores its entries in the file system. It allows users to create messages, read them (using a session for remembering the least recently read message), and deleting them again.

The test creates 100 messages of 320 characters each. These messages are then requested 50 times in batches of 10 messages. Finally, the messages are deleted again resulting in another 100 requests. The time taken for these 800 requests (message creation is verified with a second request) was measured. The goal of this benchmark is to check how well the servlet engine maintains the client's session state.

Dynamic Page Access

This benchmark uses a shopping cart servlet that we have implemented for each servlet engine.

For the measurement of this benchmark, we request the start page once to obtain the cookie containing the session information. Then, we add an item to the shopping cart, display the shopping cart, remove the item, and display the shopping cart again. The empty shopping cart page has 2048 bytes. A test run consists of 50 consecutive requests, with a total of 250 dynamic pages served. This benchmark is intended to measure the servlet engine's performance in a typical everyday situation.

Parallel Page Access

For this request, we used the previous benchmark and accessed the server simultaneously from a varying number of clients, each running on a different machine. We have measured the time needed by a single client to execute the same number of requests as in the previous test. The other clients in the benchmark were started before we started the client to be measured and also were terminated afterwards. The goal of this benchmark is to see how well the servlet engines can cope with increasing load.

Mandelbrot Calculation

This test measures the efficiency of calculation-intensive servlets by computing the Mandelbrot fractal. It accepts the size of the image, the area of the fractal to be calculated, and the maximum recursion depth as parameters. Although we support the generation of an xpm graphics file the output has been suppressed during this benchmark since we wanted to measure the "number crunching" performance only.

A test run consists of 10 accesses to the servlet, calculating a picture of the Mandelbrot set at a resolution of 1024x768 pixels and a maximum of 256 iterations per pixel.

System Library Call

Since a main reason for the design and implementation of the CSE was the possibility to easily integrate legacy applications into servlets, this test evaluates the inter-operation with legacy C and C++ code. For the test, we created a small servlet that calls a simple function from a shared library. A similar servlet might e.g. poll sensor data with high frequency in order to calculate a mean value. For C and C++ programs this is a straight-forward task. Java programs, however, have to use the Java Native Interface [Sun97, Lia99] which requires a JNI wrapper function to be written for each C or C++ function to be invoked. This benchmark measures how much performance gets lost at that interface.

8.3 Benchmark Results

Engine	CSE	Tomcat	Jetty	CSP	Micronovae
Static Page Access	35.79s	54.79s	48.20s	35.79s	35.03s
Bulletin Board System	20.34s	18.83s	21.73s	24.62s	25.77s
Dynamic Page Access	3.96s	2.34s	3.66s	4.74s	2.40s
Mandelbrot Calculation	10.69s	33.55s	33.22s	10.95s	14.86s
System Library Call	12.94s	209.39s	207.15s	14.16s	7.58s

Table 8.1: Evaluation Results

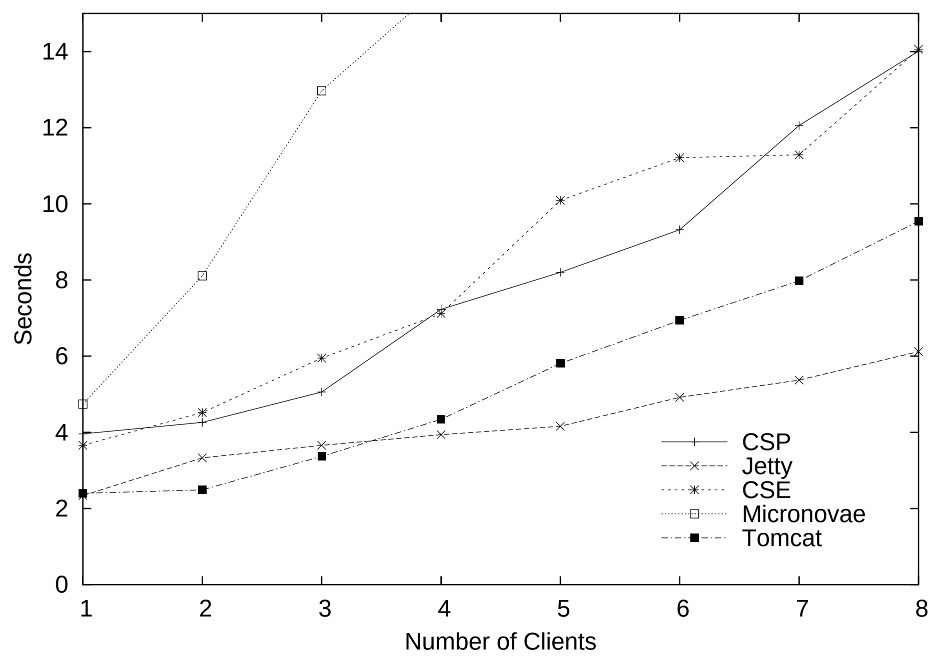

Figure 8.1: Parallel Benchmark Times

The hardware for the performance tests consisted of two computers with AMD Duron/800 MHz processors running Linux. They were linked via a 100 Mbit/s switch in order to ensure constant network bandwidth. For the dynamic and parallel tests, we used 1–8 identical Intel Pentium II/350 MHz computers as clients, with the same server as above.

The results of our tests are shown in Table 8.1. As shown by the first benchmark using Apache as front-end was the right choice for this benchmark. Tomcat and Jetty both did not perform as well. Although they can be set up to be used in combination with Apache, this setup is complicated. Mort Bay even recommends to use Jetty for static documents as well. In our evaluation, the Windows Internet Information Server was slightly faster than Apache on Linux. We assume that this effect is caused by the operating system.

The dynamic and parallel access benchmark, whose result is shown in Figure 8.1, reveals why Tomcat has become Sun's reference implementation. Both Java implementations perform much better than we would have assumed initially. Although we knew that our implementation leaves room for improvements, this was a surprise to us.

The CSE and Jetty scale equally well, but not as good as Tomcat and slightly worse than Micronovae. The CSE does not perform as well because in its current implementation the servlet server might be a bottleneck, as we have mentioned in Section 4.2.1. Jetty performs slower because it seems that its implementation has not been as well optimized as Tomcat's. CSP scales worst in our benchmark. Obviously, using the CGI for servlet execution is, at best, only a solution when C/C++ applications should be integrated into a web server whose performance is not an issue.

In the BBS test all systems were relatively close together, which indicates that bulk transfers in Java are similarly fast as in C/C++. CSP's bad result is likely caused by the fact that it creates and initializes a new task for every servlet invocation. The difference of this benchmark over the others is that the bulletin board system's entries are stored on the file system. Hence we assume that Micronovae performs worse than the other approaches due to differences in the file system implementation between Linux and Windows.

The C/C++ based systems have a clear advantage when performing CPU intensive computations as shown by the Mandelbrot calculation. It seems that the Java just-in-time compiler is unable to optimize the code as well as the GNU compiler. In the direct comparison, the compiler from Microsoft Visual C++ which is used within Micronovae performs worse than its GNU equivalent, but we do not know what optimizations are performed.

The library benchmark shows severe limitations of the Java-based systems. In our scenario, C code is more than 16 times faster when many function calls into a legacy C application need to be done. It seems that the Java Native Interface (JNI) which handles C/C++ library calls has not been optimized at all. CSP takes slightly more time than the CSE but is still an order of magnitude faster than the Java-based systems. The very good performance of Micronovae is probably caused by a different shared library mechanism provided by the Windows platform.

Chapter 9

Summary and Conclusions

In this thesis, we have introduced the architecture of a C++ Servlet Environment. We have presented the requirements of security, stability, ease of use, parallelism, and performance that a servlet environment should fulfill. Especially stability is harder to provide in C++ than in Java. Hence, we had to introduce sandboxes in which the servlets are executed and which serve as boundaries between web applications. We have shown the design of the CSE, including the interfaces to the web server and the database server, and its implementation.

Additionally, we have introduced and described both an interface for servlet programming and a language for C++ Server Pages, in which existing HTML documents can easily be extended for dynamic web programming. These interfaces have been designed similar to their well-established Java equivalents so that Java programmers can start writing C++ servlets quickly. We have explained the mechanisms used for compiling CSPs as well as the infrastructure for caching the compiler's results. We have also described how session information is managed and given examples for these techniques. Furthermore, we have presented the Record Store, a comprehensive example of servlet programming.

We have selected the most popular servlet environments and described their relation to the CSE. Finally, we have presented a benchmark suite and applied it to the top servlet environments. We have found out that the CSE performs as well as those systems. In some situations such as interfacing with legacy code it performs better, and a C++ servlet environment is the only reasonable choice for extending existing C/C++ applications.

9.1 Future Work

In several places within this thesis, we have already mentioned functionality that we would like to add, but which has not yet been implemented in the current version. Especially, we plan to further examine the possibilities and problems that a further parallelization of our system would introduce. Much performance could be gained by eliminating the servlet server and establishing direct connections between the web server's threads and the CSE's sandboxes. This design, however, would require a complete parallelization of the servlet server. Another level on which parallelism can be introduced are the sandboxes. Currently, all servlets within a single sandbox are executed serially. Parallelization would imply an overhead for synchronization too, so careful measurements must be made first.

The database interface, although it has been completely sufficient for the SQL database used, also leaves room for improvements. Currently, it is impossible to use a non-SQL database for session management. In the future, we plan to implement a traits class which enables developers to keep session data in generic databases as well. We also would like to implement more database drivers.

We further plan to implement a test environment for servlets that assists the debugging process. It will contain a minimal version of the servlet engine which can be linked to a servlet and run with any available debugger. Due to the use of multitasking and multithreading within the standard environment, current debuggers cannot be used with servlets right now.

We also plan to submit some parts of the system to the Boost web site [Boo], an Internet site which collects peer-reviewed C++ libraries for a possible future standardization. Our C++ SQL database interface might be an example of such a component.

9.2 Availability

The C++ Servlet Environment is freely available under the GNU General Public License. It is available for download at the CSE homepage at `http://www.infosys.tuwien.ac.at/CSE/` .

Appendix A

Server Configuration

This appendix describes tasks related to the setup of the C++ Servlet Environment. The CSE related configuration for the database server and the web server are covered here as well as the CSE's configuration file syntax.

A.1 The Database Server

Every database server which should be used for the CSE's session data needs to be configured first. If the servlets store only application specific data while keeping session information in memory, this setup is not necessary.

First, a database called `cse` must be created. A user account `cse` which can connect from the local machine without a password is also necessary. The database must belong to this user.

To host session information, a table `session` must be created as described in Table A.1. Session parameters are stored in the tables `session_string`, `session_int`, `session_long`, `session_float`, `session_double`, `session_blob`, and `session_datetime`. These tables are described in Table A.2, with `TYPE` standing for one of the supported session parameter types.

Column Name	Column Type	Attributes	Primary Key
sandbox	varchar(80)	not null	X
name	varchar(80)	not null	X
id	integer	not null	X
creation	datetime		
last	datetime		
validity	integer		

Table A.1: Database Table `session`

Column Name	Column Type	Attributes	Primary Key
sandbox	varchar(80)	not null	X
name	varchar(80)	not null	X
id	integer	not null	X
parameter	varchar(80)	not null	X
data	TYPE		

Table A.2: Database Table `session-TYPE`

MySQL Setup

For the MySQL database [WA02], configuration scripts that create the above database, user, and tables are included with the CSE distribution (in `db/mysql`). Possible existing data that conflicts with this setup is overwritten by the scripts, and all existing CSE session data is deleted.

The script `createdb.sql` creates the database:

```
drop database cse;

create database cse;
```

The script `createuser.sql` creates the account and sets the database permissions:

```
delete from mysql.user where user="cse";

grant all privileges on cse.* to cse@localhost;
```

Finally, the script `createtables.sql` creates the database tables:

```
create table cse.session (
  sandbox varchar(80) not null,
  name varchar(80) not null,
  id integer not null,
  creation datetime,
  last datetime,
  validity integer,
  primary key(sandbox, name, id));

create table cse.session_string (
  sandbox varchar(80) not null,
  name varchar(80) not null,
  id integer not null,
  parameter varchar(80) not null,
  data mediumtext,
  primary key(sandbox, name, id, parameter));

create table cse.session_int (
  sandbox varchar(80) not null,
  name varchar(80) not null,
  id integer not null,
  parameter varchar(80) not null,
  data integer,
  primary key(sandbox, name, id, parameter));

create table cse.session_long (
  sandbox varchar(80) not null,
  name varchar(80) not null,
  id integer not null,
  parameter varchar(80) not null,
  data bigint,
  primary key(sandbox, name, id, parameter));

```

```
create table cse.session_float (
  sandbox varchar(80) not null,
  name varchar(80) not null,
  id integer not null,
  parameter varchar(80) not null,
  data float(24),
  primary key(sandbox, name, id, parameter));

create table cse.session_double (
  sandbox varchar(80) not null,
  name varchar(80) not null,
  id integer not null,
  parameter varchar(80) not null,
  data double,
  primary key(sandbox, name, id, parameter));

create table cse.session_blob (
  sandbox varchar(80) not null,
  name varchar(80) not null,
  id integer not null,
  parameter varchar(80) not null,
  data mediumblob,
  primary key(sandbox, name, id, parameter));

create table cse.session_datetime (
  sandbox varchar(80) not null,
  name varchar(80) not null,
  id integer not null,
  parameter varchar(80) not null,
  data1 date,
  data2 time,
  primary key(sandbox, name, id, parameter));
```

A.2 The Web Server

Every web server used together with the CSE must be configured for that task. A module that sets up the communication with the servlet server must be loaded and used by the web server.

The Apache HTTP Server

For the Apache HTTP Server [Apaa], this task is done within the configuration file `httpd.conf`. The CSE specific configuration must be added to this file. The sample configuration below can be used as a start:

```
1 # Configuration for the CSE:
2 LoadFile /usr/lib/libstdc++.so.3
3 LoadModule servlets_module /.../cse/lib/cse/mod_servlets.so
4 ServletConfig /.../cse/etc/servlets.conf
```

Line 2 might or might not be necessary, depending on the destination system. If it is used, it must be changed to refer to the location of the system's C++ standard library. Line 3 connects the web server to the servlet server by loading the corresponding web server module. Line 4 sets a configuration file, which is usually not necessary from within the web server. If the default configuration file is used, it can be left out.

A.3 The C++ Servlet Environment

Once this configuration has been made, the system is ready for the C++ Servlet Environment. Before compiling the CSE, it is important to set the default database to the one that has been configured for session management. In the current version, only a single database driver is provided, so that the provided configuration should be correct in this aspect. No additional configuration is necessary for the connection to the web server.

The CSE Configuration File

The following sample illustrates the syntax of the CSE's configuration file (`servlets.conf`):

```
1 /* sample configuration file */
2 Sandbox "My Sandbox" {
3   /* configuration of the sample sandbox */
4   Load /usr/lib/libmyutils.so
5   /servlets/myservlet1      /usr/share/cse/servlets/myservlet1.so
6   "/servlets/my servlet 2"  "/usr/share/cse/servlets/my servlet 2.so"
7 }
8 Sandbox CSPs
```

Everything between a `/*` and the next `*/` is discarded as a comment. Commands must be written exactly as shown, `sandbox` is not the same as `Sandbox`. Commands, arguments, and braces are separated by whitespaces. If a command argument includes a whitespace, it must be quoted. Quotes within arguments are currently not supported. Arguments must be specified; if an empty string should be used, it is written as `""`.

The `Sandbox` command defines a new sandbox. Its argument is the name of the sandbox to be created. The sandbox argument may be followed by an opening brace, which indicates that everything up to the next closing brace is used to configure the sandbox. In the example above, lines 3–6 indicate such a configuration, while line 8 creates an unconfigured sandbox (which might be used by CSPs).

Within a sandbox configuration environment, the command `Load` indicates that the sandbox should load a shared object which will not be used as a servlet. Servlets can then use functionality provided by that library. Everything else is regarded as a servlet definition. Such a definition consists of the URL to be used (which must be unique within the servlet server) and the shared object file containing the servlet.

Appendix B

Support Classes

With the purpose of a more straight-forward implementation in mind, we have implemented a few support classes that are used by the C++ Servlet Environment. They use the C++ exception handling mechanism (`try`, `catch`) for handling errors. They are not meant to implement all the features of the underlying C mechanisms, they only provide what the system needs.

These classes use inlining for all their functions, and map a set of standard C Library calls to an object, e.g. a socket. Because of the extreme application of inlining, no performance is lost through these additional classes. The only drawback is a slight increase in compilation time.

Sidebar: Inlining

Since C++ is an object-oriented programming language, programs written in "good C++" (meaning programs that actually use the object-oriented features) tend to use a higher number of function calls than e.g. C programs. On the other hand, a function call introduces an overhead for context switching that is not always necessary, which might seduce programmers to sacrifice the notion of "good C++" for performance.

Unlike other object-oriented languages, C++ offers a solution to this dilemma: It allows programmers to declare a function as "inline". This declaration tells the compiler to include the function's body whenever it encounters a function call, eliminating the performance loss. In order to use this feature, the function definition must be available to the compiler, which usually implies that it must be placed into a header file.

The result of inlining is that the program can be written in a "good", truly object-oriented way without a performance loss. The only visible difference is that some function definitions are moved into the header files.

The C++ support classes are:

Exception (`include/cse/server/Exceptions.h`):
: The superclass of a number of exceptions thrown by the other classes. They each store a string to indicate the problem.

Pipe (`include/cse/server/Pipe.h`):
: Encapsulates a pipe, usually used for communication between a parent and a child process. In the current version, this class has been changed to use a pair of sockets which allows all operations a pipe does. In addition, file descriptors can be passed over a socket, which is used by the CSE for performance reasons.

Semaphore (`include/cse/server/Semaphore.h`):
: A generic semaphore, meaning one that can be initialized with an integer. Used for controlling shared resources. This class uses the *linuxthreads* implementation of semaphores [Ler97] — that is, it can be used for synchronization of several threads within a single task, but not for cross-task synchronization.

ServerSocket (`include/cse/server/ServerSocket.h`):
: Does everything a TCP/IP socket based server's main loop needs. Whenever the server gets a new connection from a client, it returns a Socket object (see below).

Socket (`include/cse/server/Socket.h`):
: Contains a TCP/IP socket. It is created by a TCP/IP socket based client, or returned by a server process.

Task (`include/cse/server/Task.h`):
: Creates a new task. It stores the function to call and a parameter (a generic pointer). A Task object can be copied in order to create a number of worker tasks.

Thread (`include/cse/server/Thread.h`):
: Creates a new thread. It works like the Task class (see above). The class uses the *linuxthreads* implementation of threads [Ler97].

Bibliography

[AMW01] Kevin Atkinson, Sinisa Milivojevic, and Michael Widenius. *Mysql++ — A C++ API for Mysql*, May 2001. Version 1.7.9.

[Apaa] The Apache Software Foundation. The Apache HTTP Server. http://httpd.apache.org/docs-2.0/ .

[Apab] The Apache Software Foundation. The Apache Tomcat Servlet Engine. http://jakarta.apache.org/tomcat/ .

[Ape] Ape Software. The Servlet++ Homepage. http://www.apesoft.net/servlet++/ .

[BAS$^+$02] Stig Sæther Bakken, Alexander Aulbach, Egon Schmid, Jim Winstead, Lars Torben Wilson, Rasmus Lerdorf, Andrei Zmievski, and Jouni Ahto. PHP manual, March 2002. http://www.php.net/docs.php .

[Boo] The Boost Homepage. http://www.boost.org/ .

[Deb] The Debian Project. *Debian Free Software Guidelines.* http://www.debian.org/social_contract#guidelines .

[FGM$^+$99] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. *Hypertext Transfer Protocol—HTTP/1.1*, June 1999. Internet RFC2616.

[Gon] Teodoro González. C Server Pages. http://www.cserverpages.com/ .

[GS03] Thomas Gschwind and Benjamin A. Schmit. CSE — a C++ servlet environment for high-performance web applications. In *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference*, pages 15–27. USENIX Association, June 2003.

[Gsc01] Thomas Gschwind. PSTL—A C++ Persistent Standard Template Library. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems*, pages 147–158. USENIX, January 2001.

[IBM] IBM Corporation. Jikes. http://www.ibm.com/developerworks/oss/jikes/ .

[KM97] Dave Kristol and Lou Montulli. *HTTP State Management Mechanism*, February 1997. Internet RFC2109.

[LB02] Sotiria Lampoudi and David M. Beazley. SWILL: A simple embedded web server library. In *Proceedings of the 2002 USENIX Annual Technical Conference*. USENIX, June 2002.

[Ler97] Xavier Leroy. *Linuxthreads — POSIX 1003.1c kernel threads for Linux*, 1997.

[Lia99] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification.* Addison-Wesley, June 1999.

[LP01] Amos Latteier and Michel Pelletier. *The Zope Book.* New Riders Publishing, July 2001.

[LY99] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification.* Addison-Wesley, 2nd edition, April 1999.

[Mica] Micronovae. The Micronovae CSP Engine Homepage. http://www.micronovae.com/CSP.html .

[Micb] Microsoft Corporation. Active Server Pages. http://msdn.microsoft.com/asp/ .

[Mor] Mort Bay Consulting. Jetty—Java HTTP server and servlet container. http://www.mortbay.org/jetty/ .

[Mye95] Nathan C. Myers. Traits: A new and useful template technique. *C++ Report*, June 1995.

[Net03] Netcraft. Netcraft Web Server Survey, August 2003. http://www.netcraft.com/survey/ .

[Ope96] Open Market, Inc. *FastCGI: A High-Performance Web Server Interface*, April 1996. White paper.

[Rog] Rogue Wave Software. *The Bobcat Servlet Container: Using C++ to Integrate Applications and Business Logic with the Web.* White paper.

[RRST99] E. Rescorla, RTFM, Inc., A. Schiffmann, and Terisa Systems, Inc. *The Secure HyperText Transfer Protocol*, August 1999. Experimental RFC 2660.

[Ser03] The Server Side. Application server matrix, May 2003. http://www.theserverside.com/reviews/matrix.jsp .

[Str00] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3rd (special) edition, February 2000.

[Sun97] Sun Microsystems. *Java Native Interface Specification*, May 1997.

[Sun01a] Sun Microsystems. *Java Servlet Specification*, August 2001. Version 2.3.

[Sun01b] Sun Microsystems. *The JavaServer Pages Specification*, August 2001. Version 1.2.

[WA02] Michael Widenius and David Axmark. *MySQL Reference Manual.* O'Reilly & Associates, June 2002.

[Web] Webletworks. The Weblet Application Server Homepage. http://www.webletworks.com/ .