CSE — A C++ Servlet Environment for High-Performance Web Applications

Thomas Gschwind Benjamin A. Schmit
Abteilung für Verteilte Systeme
Technische Universität Wien
Argentinierstraße 8/E1841
A-1040 Wien, Austria, Europe
`{tom,benjamin}@infosys.tuwien.ac.at`
`http://www.infosys.tuwien.ac.at/Staff/{tom,benjamin}/`

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. We have developed a C/C++ Servlet Environment (CSE) that provides a high performance servlet engine for C and 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 and C++ application development. In this paper we explain our architecture, the challenges and trade-offs we have faced, and compare the performance of our environment to that of top servlet environments available today.

1 Introduction

Every day, the web is applied to more and new application domains. In some cases people try to convert services that historically have been handled by a different mechanism, such as USENET News, to the web. In other cases, such as email, they are complemented by a mechanism that allows users to access these services via a web front-end. Due to this success and the increasing number of application domains web server performance is of major importance [8, 11] and unresponsive and slow web sites may send users seeking for alternatives [5].

Traditionally, web services have been implemented using CGI programs in the form of compiled C and C++ programs or Perl scripts that were executed by the web server. Although these approaches work perfectly fine for simple dynamic web content they are cumbersome to use if a whole business process should be modeled as a web application. This stems from the fact that they do not take care of the state-management between subsequent web requests. These issues, however, are taken care of by newer approaches such as the Java Servlet Technology [22], the Python Zope Server [14], or dedicated application servers (e.g., Bea Weblogic or JBoss).

One advantage of servlets is that they are executed as part of a servlet environment that, unlike CGI scripts, needs not be restarted at each invocation. To protect one servlet from another, these environments use programming languages that take care of memory management issues. Another advantage is that these languages come bundled with standardized libraries providing support for numerous different tasks.

These advantages, however, have a price. Legacy applications that make use of C or C++ cannot be easily integrated. Although technologies such as SWIG [4] or JNI [16, 25] simplify the integration of legacy applications into scripting languages they still require developers of such systems to deal with different systems. Sometimes the choice of language is mandated by management or the developers simply prefer the use of C or C++.

Another advantage of C and C++ is that these languages provide better performance and a finer grained integration of the operating system's security mechanisms. This is probably one of the reasons why the Apache HTTP Server and the Microsoft Internet Information Server, the two most prominent web servers with a combined market share of over 85% [20], are written in C and C++ respectively.

In this paper, we present a servlet environment that is completely implemented in C++. To the best of our knowledge, our C/C++ Servlet Environment (CSE) is the first servlet environment that uses the potential of C++. Similar to the servlet engines implemented in Java, the architecture of our servlet environment offers adequate protection between the individual servlets to be executed. Hence, our servlet engine provides the following advantages over those using Java or scripting languages:

- It supports the integration of existing C/C++ code into web applications without mixing different programming languages and converting data types.

- It provides a better integration into the security mechanisms provided by the operating system.
- The C++ programming language allows developers to write more efficient code since they can choose from a wider range of implementation choices [24].

This paper is structured as follows. In Section 2 we present the requirements for a servlet environment as well as the advantages and challenges of using C++. Section 3 shows how these challenges have influenced the design and implementation of the C/C++ Servlet Environment, and application development for the CSE is discussed in Section 4. Related work is presented in Section 5 and in Section 6 we compare the performance of our approach with that of other approaches. Future work is discussed in Section 7 and we draw our conclusions in Section 8.

2 Requirements

The main goals during the design of the C/C++ Servlet Environment were security, stability, ease of use, parallelism, and performance. Before we can have a closer look at these requirements, however, it is necessary to give a brief overview of the typical architecture of a web site. This architecture is depicted in Figure 1.

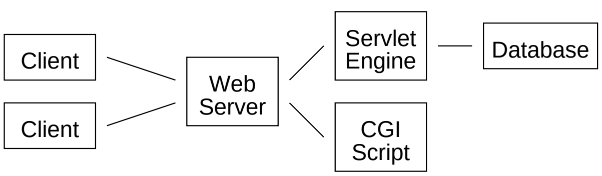

Figure 1: A Typical Web Site Implementation

A web site must include a web server that handles requests from multiple clients. Requests that cannot be handled by the web server itself are forwarded to a CGI program or a servlet engine where a web service is executed. The web service in turn may contact a database or application server for persistent data storage. Since HTTP is stateless [7], the client has to transmit the application's current state or a session identifier with each request. In the latter case a mapping from session identifier to state has to be maintained by the servlet engine.

CGI programs have the drawback that they have to be executed anew for each client's request. Hence, they need to be started at each request and then have to read in their configuration data and session state from persistent storage. A servlet engine, on the other hand, is running all the time and keeps several servlets as well as their session state in memory. Servlets need to maintain the client's session state (for instance, the contents of a shopping basket) because web clients only provide limited functionality to maintain this kind of data.

Since many servlet engines execute several different servlets within the same process *stability* is a major concern. If one servlet crashes, the other servlets have to continue to run unaffectedly. For a servlet engine, it is very important to recognize and handle this kind of failure since there is no way of judging the stability of a user-supplied servlet from within the servlet engine.

Stability is one reason why Java and interpreted languages in general are popular for this task. If a Java servlet crashes only its thread of execution is terminated and all the other servlets continue to run. Hence, the worst that can happen is that a servlet consumes overly much resources such as processor time, memory in the form of unused but still referenced Java objects, or network bandwidth. The price for these benefits is that all servlets are executed with the same privileges and that the operating system's security mechanisms need to be re-implemented as part of the servlet engine. Another drawback is a slight performance overhead since Java does not allow developers to write code on a level as low as it can be written with C and C++.

The disadvantage of C and C++ is that these programming languages are not as safe. Bugs in C and C++ programs typically take down the whole process and they tend to get apparent at a much later time (typically at a point of execution that is unrelated to the place that caused the error). Therefore, even if the application is catching the signal that a segmentation violation has occurred, recovery is difficult. Hence, the design of a servlet engine written in these languages has to solve those challenges.

Security is necessary to minimize the chance that servlets can be exploited by an intruder. Our architecture reuses the security mechanisms that have been built into the operating system. It allows sandboxes, and thus servlets, within the same servlet environment to be executed with different user privileges. The advantage of this approach is that system administrators can use standard access privilege mechanisms and do not have to get familiar with a new security management system which can lead to misunderstandings. Additionally, our approach requires only a single security mechanism to be checked for possible vulnerabilities.

A disadvantage of C and C++, however, is that such programs are open to buffer overflow attacks if they have not been implemented carefully. The threat and impact of such attacks can be minimized by using the C++ Standard Template Library which has been designed in order to minimize such programming mistakes and by using operating system mechanisms such as a non-executable user stack area. No programming language or servlet environment, however, can guarantee that a program or servlet cannot be exploited by an intruder. The final responsibility is always up to the developer.

Ease of use is another important aspect for a servlet engine. Even though there are servlet engines for Java that provide good performance, these servlet engines are of little use to C or C++ programmers that want to use existing code for their web applications. Using C and C++ in combination with such servlet engines requires the use of the Java Native Interface (JNI) [16, 25], the conversion between C and Java data types, and to deal with low-level details of both languages. Although systems such as SWIG [4] can be used to help with this issue they do not solve the problem of having to deal with two different languages. Hence, a C++ servlet environment is more convenient to use for developers that have to deal with C or C++ code.

A good servlet engine must not only be easy to use but has to provide good *performance* as well. This gets apparent by the fact that the two most popular web servers are implemented in C and C++ respectively. A convenient servlet environment that requires load balancing among multiple servers to be able to handle the incoming requests loses much of its original appeal. This is one of the reasons why we have developed the CSE. The CSE was designed towards optimal performance.

Parallelism is, on a server machine, a prerequisite for performance and scalability. On a network, the upper bound of the access time to a service can be high and thus forbids handling requests in serial. The CSE introduces parallelism by providing several services on a single machine, by partitioning a service into several parts with simple interfaces between them, and by replicating those parts. Additionally, compared to using interpreted languages that use a global interpreter lock for thread locking [28, Section 8.1], the approach to use C++ has the advantage of having a fine grained thread locking mechanism as provided by linux-threads or pthreads [15].

3 Design and Implementation

Figure 2 shows the architecture of our C/C++ Servlet Environment. It consists of an Apache module, a servlet server and several sandbox processes. The CSE uses the Apache server to handle incoming HTTP requests, the servlet server is used for the management of the sandbox processes and the sandboxes are responsible for the execution of the servlets. We use different sandbox processes because this protects a servlet in one sandbox from an unstable one in another sandbox. Additionally, this approach allows administrators to execute different sandboxes with different user privileges. Although we have used C++ for the implementation, our architecture can be easily reused for a servlet environment implemented in plain C.

3.1 The Apache Module

We use the Apache web server [1] to handle HTTP requests. This approach has several advantages over implementing a new web server for our servlet engine such as provided by the Tomcat Servlet Engine [2].

- Apache uses a modular design and can be extended easily.
- It is implemented in C facilitating data exchange with our C++ servlet engine. C structures are compatible to C++ structures.
- Its add-on modules provide features that we would never have implemented as part of CSE.
- It has a 60% market share [20] and has proven to be a reliable web server.
- It is free software.

Another advantage of the modular design we have chosen is that only the web server's *Servlet Module* has to be re-implemented if a different web server has to be used for a given web site.

The Apache Module registers the URLs that are implemented by the servlet engine's servlets and the C++ Server Pages. Requests to other URLs such as static web pages or images are handled by Apache itself without consulting our module. Servlet requests are forwarded to the servlet server which assigns them to one of the sandboxes. One risk of this approach is that the servlet server might become a bottleneck. Hence, in future versions of CSE, we plan to extend the Apache module such that the requests are sent to the appropriate sandbox directly. Going a step further and executing the servlets by the Apache module directly, however, is not possible since that would compromise the web server's stability.

3.2 The Servlet Server

The *Servlet Server* is responsible for the management of the sandboxes. To do that it processes requests forwarded by the Apache module and determines, based on the servlet registry, the sandbox that is responsible for the servlet's execution. The *Servlet Registry* maintains a mapping of the servlets and the sandboxes they should be executed in. This mapping defined within the server's configuration file.

If a C++ Server Page (CSP) is used, the *CSP Manager* converts the CSP document into a servlet and compiles it. Several *Compiler Threads* can be started by the server in order to compile CSPs when they are first requested or when they have changed. They are synchronized so that no more than a single thread for a single servlet can run at a time. The compiled servlet is put into a cache directory, along with the servlet source code, a configuration file containing the destination sandbox, and an error output file which also serves as a timestamp of the last compilation attempt. A CSP is only recompiled if

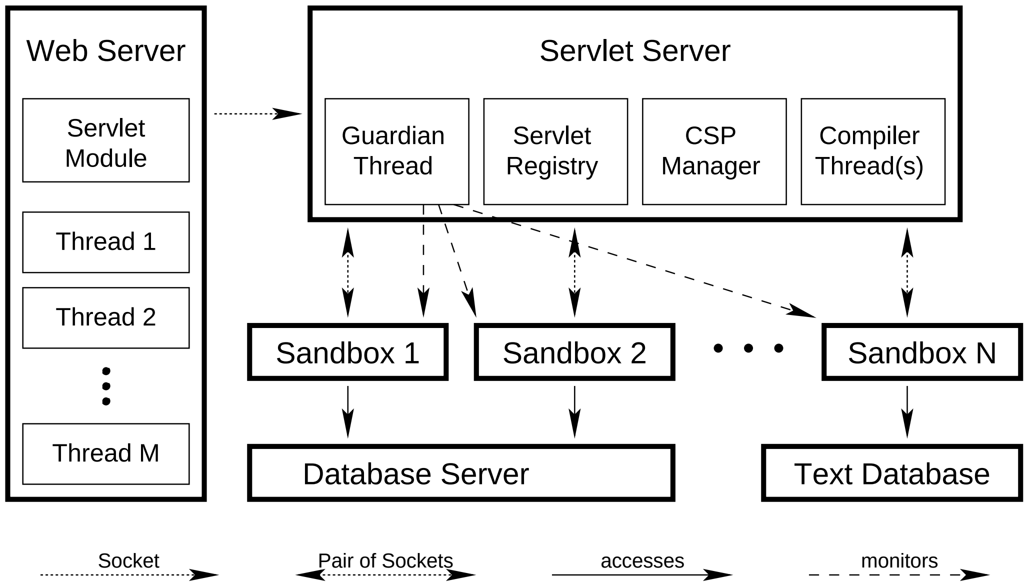

Figure 2: CSE Architecture

its timestamp is newer than the timestamp of its error output file. We do not use the CSP's object file since it is only created if the compilation is successful. If the sandbox does not exist yet, it will be created after compilation. Then, the destination sandbox is configured to host the new servlet. If for some reasons, however, CSP compilation during run-time is undesired or impossible, it is possible to compile them before starting the server.

A *Guardian Thread* watches over the state of the sandboxes and restarts them if an unstable web application crashes. Instead of using a separate thread that sleeps most of the time, it should also be possible on UNIX systems to catch the child signal instead. From the performance point of view, however, this poses no difference.

In order to minimize downtimes of the servlet server, its configuration can be changed dynamically. The configuration file tells the servlet registry which servlets should be loaded into which sandboxes. When the application server processes a request, it first checks the timestamp of the configuration file. If there was a change (or there was no request after startup yet), the configuration file is read, and the internal configuration data is updated. Part of the configuration data, such as the location of the shared object files, is passed on to the sandboxes where it is needed. This update process is also invoked when a sandbox has to be restarted by the guardian thread.

3.3 The Sandboxes

Several separate tasks, the *Sandboxes*, handle the actual execution of the servlets. Each sandbox may encapsulate one or more web applications consisting of one or more servlets. The purpose of using several sandboxes is to take care of the session management and to provide a barrier for unstable servlets.

The ability to execute several servlets within the same sandbox increases the scalability of our servlet environment. This approach allows developers to place servlets that frequently need to interact with each other into the same sandbox and hence enables a more efficient communication between the servlets and reduces the number of context switches required.

The sandbox processes read requests from an input socket and execute them. The servlets themselves are loaded as dynamically shared objects. C++ Server Pages are translated into servlets which in turn are compiled into shared objects. Shared objects can be loaded into a running program on demand by a system call. The disadvantage, however, is that the server's original programmer can never be sure about their stability.

If a servlet crashes while processing a request, the servlet server's guardian thread notices the crash and restarts and reinitializes the failed sandbox. If a request comes in while the destination sandbox is down, it is buffered and executed as soon as the sandbox is up and running again.

This architecture has already proved its usefulness: Our original implementation contained an error that would crash every sandbox after a little more than 1000 requests. We have not discovered this error until we have started running the benchmarks because the crashed sandboxes were always restarted, and (except for a slight performance loss) no problem was visible.

Additionally, since each sandbox is executed within its own process, system administrators may choose to run different sandboxes with different user privileges. This architecture provides a finer grained access control than Java servlet environments which execute all servlets within the same process and thus with the same user privileges.

3.4 Persistent Storage

The ability to provide persistent storage is of major importance for most web applications. A voting application, for example, has to manage the votes cast by its users. The persistence mechanism of the CSE can be used internally by the sandbox's session management if the servlet's `sessionType` attribute is set to `DatabaseSession`. Alternatively, it may be used directly by the application developer.

The C++ Servlet Environment has been designed so that the persistence mechanism can be replaced easily. The persistence mechanism is encapsulated by a traits class [19, 24] with which its classes are parameterized. A traits class can be compared to a set of callback functions with the difference that they are known during compile time and thus leave more room for optimization. This approach allows application developers to choose a persistence level that fits their needs best, e.g., a database such as MySQL [30], or a file-based database such as PSTL, a persistent implementation of the Standard Template Library [10].

We also provide a general-purpose database interface which is a set of template classes that offer general database handling functions, along with data-types used within a database. They do not contain functions to access specific databases but are able to use concepts like SQL statements and cursors. The template classes provided are:

`DB` is used to create, maintain, and close a database connection. How this is done depends on the concrete database driver. Also, this class allows for the execution of SQL statements that do not return any data such as `INSERT` statements.

`DBQuery` helps formulating SQL queries. The query is written to the object as if it would be written to a stream.

`DBResponse` executes an SQL query that returns some data. Again, the implementation depends on the concrete database driver. The class has many features similar to those of a C++ STL container [24].

`DBIterator` implements a C++ STL input iterator [24] that iterates over the elements of a result set and supports database drivers with and without cursors.

`DBRow` is created when the iterator is accessed. Its index operator ("`[]`") is used to retrieve an actual data item. It uses either an integer (the column id) or a string (the column designation) as the argument.

`DBObject` is the class that contains data items. It has the subclasses `DBString`, `DBInt`, `DBLong`, `DBFloat`, `DBDouble`, `DBBlob`, and `DBDateTime`. They store C++ strings, 32- and 64-bit integers, 32- and 64-bit floating point numbers, binary content, and dates.

In order to access a given database server, a concrete database driver (a traits class) for that database must be written. If the database has a C++ interface, this task is usually trivial because most work has already been done at the general-purpose database interface. A concrete database driver for the successful database MySQL is already available.

4 Application Development

The C++ Servlet Environment provides two approaches for writing web applications similar to those available in Java Servlet Environments: Servlets that use only C/C++ and C++ Server Pages that use C/C++ code embedded in HTML code.

4.1 Servlets

A Servlet is implemented as a C++ class inheriting from the `Servlet` class as shown in Figure 3. Subsequently this class is compiled into a shared object. Servlets can access the parameters and cookies that have been passed as part of the web request and output an HTML page onto their output stream.

The most important methods and attributes of the `Servlet` class are:

`service():` This method is called for each client request that has to be processed. The default implementation calls either `doGet()` for `GET` or `doPost()` for `POST` requests. If necessary, this method may be overridden. The `service()` method's parameters are a `ServletRequest` object providing details about the current request, a `ServletResponse` object handling the servlet's response, and a `Session` object containing session information if requested.

`doGet(), doPost():` These methods are similar to the `service()` method but handle only `GET` or `POST` requests.

`getSession():` Requests a Session object from the session manager. The application server automatically executes this method if the servlet's `sessionType` is set. The method is implemented within the servlet base class.

`sessionType:` This attribute indicates whether the sandbox should take care of the servlet's session management and the type of session management

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

class HelloServlet : public Servlet {
protected:
  int counter;

public:
  HelloServlet() : counter(0) { session=false; }
  virtual ~HelloServlet() { }
  virtual void service(const ServletRequest& rq, ServletResponse& re,
                       Session* session=NULL) {
    re << "<html><head><title>Hello World</title></head>" << endl
       << "<body><h1>Hello World</h1>" << endl
       << "<p>Servlet request count: " << ++counter << "</p>" << endl
       << "</body></html>" << endl;
  }
};

Servlet* factory() {
  return new HelloServlet();
}
```

Figure 3: C++ Servlet Example

requested. Valid options are `NULL` for no session management, `MemorySession` for transient session management, and `DatabaseSession` for persistent session management surviving sandbox or server restarts.

`threadSafe`: Indicates whether the servlet is able to process multiple requests simultaneously. This functionality, however, is not yet implemented.

The `ServletRequest` class encapsulates information about the current request. It provides access to the HTTP request parameters using the `getParameters()` method. Such parameters may be passed as part of the URL in case of a `GET` request and as part of the request body in case of a `POST` request. The request parameters are decoded and returned as a `map` using the parameter names as keys.

Cookies sent with the HTTP request can be obtained with the `getCookies()` method. They are automatically transformed into `Cookie` objects. The return type is a vector of these objects. Unless a cookie has been changed there is no need to include it in the response sent back to the client.

Among other information, the `ServletRequest` class also provides the URL of the request (`getURL()`) and whether the request used the `GET` or `POST` method (`getMethod()`).

The `ServletResponse` object contains a stream to which the output of the servlet is written. For example,

```
re << rq.getURL();
```

writes the servlet's URL to the output. Other interesting attributes of this class are `contentType` which specifies the MIME type of the response and `cookies` which is an initially empty `vector` containing the cookies to be sent to the client.

Session data is provided through the `Session` class. Each session has a name and an ID. Together with the Sandbox name, they uniquely identify the session. The name distinguishes between different types of sessions within a single web application. The session IDs are assigned in a random order, which makes guessing them almost impossible and thus enhances data security.

The `setParameter()` and `getParameter()` methods allow a servlet to store and retrieve session data. Depending on the kind of session, these data are stored within the memory or within a database.

The session identifier must be passed on between subsequent requests to the web server. This can be done using cookies. The function `getCookie()` provides a cookie (as defined in [12]) that contains the session information. Cookies, however, do not work with all web clients. If the servlet programmer cannot rely on them, the methods `asLink()` and `asForm()` transform the session designation into a string suitable for links (in URL-encoded format) or for forms (as a hidden field).

4.2 C++ Server Pages

C++ Server Pages (CSPs) are stored in the web server's document root directory along with static HTML pages and can be identified by their *.csp*-extension. When they

```
<%#vector%>
<%!vector<string> strings;%>

<% // 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); %>

<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: <%=strings.size()%></th></tr>
    <% for (vector<string>::iterator i=strings.begin(); i!=strings.end(); ++i) { %>
    <tr>
      <td><%=*i%></td>
      <td><a href="TableServlet.csp?remove=<%=i - strings.begin()%>">remove</a></td>
    </tr>
    <% } %>
  </table></p>
</body></html>
```

Figure 4: `TableServlet.csp`

are requested for the first time these pages are converted into servlets and subsequently into shared objects.

CSPs are HTML documents enriched with special tags containing, among other things, C++ code. A sample CSP is shown in Figure 4. The tags used to identify C++ declarations and code are described below. Except for the first two tags, they have been designed similar to the JavaServer Pages (JSP) specification [27] to increase readability for people familiar with JSPs.

- <**`%$sandbox%`**> declares the sandbox the CSP belongs to. CSPs without this tag are executed within the default sandbox.
- <**`%#include%`**> denotes a header file to be included. As in a normal C++ program, header files provide declarations for functions and objects defined by a library. The library itself can be loaded from the configuration file. Different libraries may be used for different sandboxes.
- <**`%!definition%`**> declares an attribute or a method.
- <**`%@initialization%`**> indicates code to be placed into the constructor of the servlet's class. Here, attributes (those declared within this class and those inherited from the `Servlet` class) are initialized.
- <**`%code%`**> executes C++ code. It is written into the servlet's `service()` function at the tag's location.
- <**`%=expression%`**> evaluates an expression and inserts the result into the servlet's response. It may be of any type that can be written to an output stream.
- <**`%--comment--%`**> declares a CSP comment. Unlike an HTML comment, its contents are not sent to the web browser.

The example shown in Figure 4 first includes the `vector` header file and declares a `vector` containing strings. The second block checks for the parameters passed to the CSP and based on these parameters removes or adds a new string to the `vector`. The remaining part of the servlet is used to display a form to add a new string and a table with the strings currently stored in the `vector`. Additionally, the strings are supplied with links to allow them to be removed.

After the example servlet has been deployed, our CSE translates it into a C++ servlet as shown in Figure 5. Include directives of the C++ Server Page are converted into include pre-processor macros at the beginning of the file (line 1), definitions are converted into attribute and member function definitions (line 8). Code and expression directives are used to form the `service()` mem-

```
#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);
};

// ... helper functions ...

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></p>
</body></html>
";
}
```

Figure 5: `TableServlet.cc` Generated from `TableServlet.csp`

ber function and HTML code is converted into statements sending it unmodified to the web client (lines 19–57).

5 Related Work

Since we have started with the implementation of our C++ Servlet Environment other developers have also recognized the need for a servlet environment for C++.

The commercial vendor Rogue Wave has developed *Bobcat* [21], a C++ servlet engine that has an API similar to that of the Java Servlet Specification [26]. Unfortunately, its evaluation license 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.

Ape Software, an Indian company, has developed *Servlet++* [3] under a BSD-like free license, which also supports a basic form of C++ Server Pages. Unlike CSE, it does not contain a stand-alone server for the execution of the servlets but is implemented completely within an Apache module. Hence, an unstable servlet can compromise the stability of the Apache server itself. The Servlet++ module supports C++ servlets with an interface similar to that of the Java Servlet class. Servlets are loaded as shared objects. Unlike CSE, servlets and C++ Server Pages have to be compiled manually. Also, Servlet++ currently lacks session management, a fundamental necessity for every servlet environment, and hence we left it out of the comparison in section 6.

C Server Pages [9] is a servlet engine that has been designed with goals somewhat similar to those of the C++ Servlet Environment, but does not use a free license (commercial use is non-free). However, this system uses no sandboxes to encapsulate its servlets, so that it is less secure. There is currently no support for dynamic configuration, which means that each servlet has to be compiled manually (using a tool to transform C Server Pages into servlets and a C++ compiler), and the server then has to be restarted. C Server Pages is implemented as a CGI script, but the author has also built an Apache module which is, unfortunately, not yet available for download.

Micronovae [17] is developing a C++ Server Pages engine which works together with Microsoft's Internet Information Server (IIS). Like the previous system, it does not use the concept of sandboxes. Additionally, it only provides support for C++ Server Pages but not for servlets. Unfortunately, their download (beta version) includes no source code, so that our information about this system is solely based on our experiences with it.

The *Weblet Application Server* [29] seems to be a servlet engine for C++ developed by *Webletworks*. Unfortunately, we were unable to contact the web server of the application server's vendor for several months now and were also unable to obtain a copy or other information about the system through other web sites.

Besides C++ servlet engines, there are numerous such engines for Java and scripting languages. One such servlet engine is the *Apache Tomcat* [2] servlet engine, a subproject of the Apache Jakarta Project. It has complete support of the JavaServer Pages and Java Servlet specifications and is included in Sun's reference implementation. Although Tomcat contains its own web server it can also cooperate with other web servers like the Apache HTTP Server. JavaServer Pages may be compiled by the built-in Java compiler or by alternative Java compilers such as Jikes.

Jetty [18] is a Java Servlet engine developed by the Australian company *Mort Bay*. It can cooperate with the Apache HTTP Server, but Mort Bay suggests to use the included web server. Jetty is available under a free license and offers both Java Servlets and JavaServer Pages. Like Tomcat, Jetty is configured using a set of XML files and may use alternative Java compilers for the compilation of JavaServer Pages.

Swill [13] is a lightweight programming library that provides a simple web server for C and C++. Unlike our servlet environment, its goal is not to provide a full fledged web 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.

Zope [14] is a servlet environment that allows developers to use Python for servlet development. It includes a web server and a web administration front-end. Initial performance results have shown that Zope cannot compete with the top servlet engines. We assume that this is due to the performance penalty incurred by the Python interpreter. Zope is probably the right environment for Python programmers maintaining small web sites.

6 Evaluation

For the evaluation of the C++ Servlet Environment's performance we have implemented a benchmark suite that tests various aspects of the different servlet engines. 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.

6.1 Benchmarks

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 functions 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 [16] 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.

6.2 Benchmark Results

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 1. 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.

As shown by the static page access 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 6, 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 3.1. 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 bulletin board system test all systems were relatively close together, which indicates that bulk transfers

Engine	CSE	Tomcat	Jetty	CSP	Micronovae
Static Page Access	35.79s	54.79s	48.20s	35.79s	35.03s
Bulletin Board System	3.96s	2.34s	3.66s	4.74s	2.40s
Dynamic Page Access	20.34s	18.83s	21.73s	24.62s	25.77s
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 1: Evaluation Results

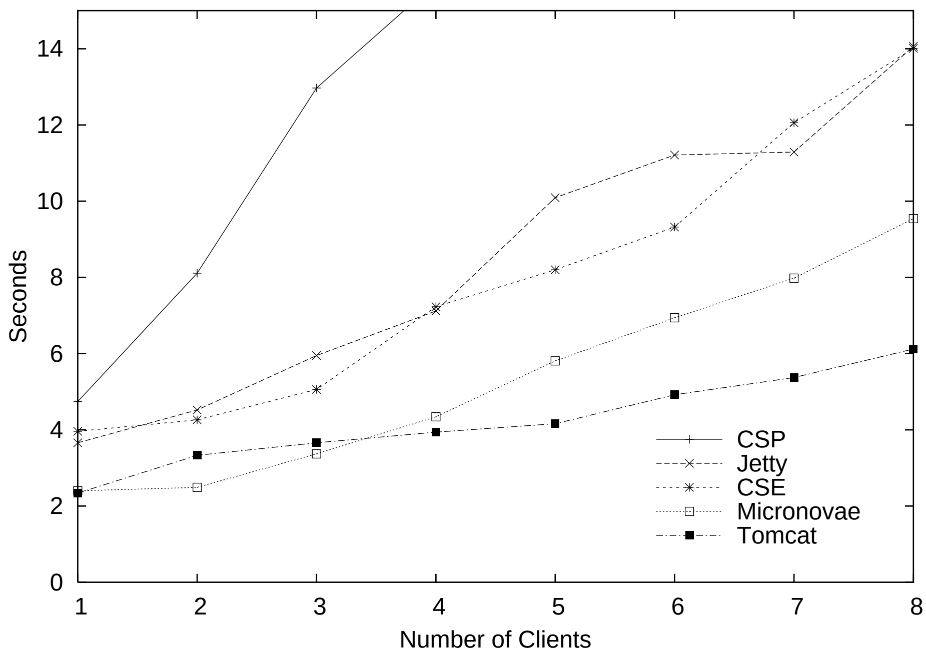

Figure 6: Parallel Page Access Benchmark Times

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.

7 Future Work

The current implementation of the CSE has some room for optimization. This gets apparent by looking at the architecture presented in Section 3. Requests to the individual servlets are delegated to the sandboxes by the servlet server. This is a potential bottleneck and could be solved by letting the Apache module themselves delegate the requests to the individual sandboxes. Additionally, our current implementation does not yet allow the simultaneous execution of a servlet's `service()` method. This stems from the fact that we do not yet honor a servlet's `threadSafe` attribute, which results in a loss of performance.

During our tests we identified that our servlet engine is not yet capable of handling binary content correctly. We assume that this bug is located within the Apache module that passes the result of the sandboxes back to the clients. Our current assumption is that we do not handle the NULL character correctly since it indicates the end of a C string.

In future versions we also plan to implement a test environment for servlets and C++ Server Pages that supports testing outside the CSE. To test the servlet, the environment would be linked to the servlet and could be debugged like a stand-alone C++ application.

In future versions of the CSE, we also plan submit it to the BOOST web site [6] which provides a collection of free peer-reviewed portable C++ source libraries. The focus of BOOST is on libraries that work well with the C++ Standard Library and are suitable for eventual standardization.

8 Conclusions

The contribution of this paper is an architecture that enables the implementation of high-performance web applications using C or C++ while providing the stability known from Java servlet engines.

We have also implemented a C++ Servlet Environment using this architecture. Although our implementation has not yet been optimized and provides enough room for further performance improvements, it offers similar performance as the top servlet engines available today.

Our C++ Servlet Environment uses an API which is based on that provided by Java Servlets and JavaServer Pages. This design choice has the advantage that developers familiar with this technology will immediately be able to write C++ Servlets and C++ Server Pages.

Providing a servlet environment for C++ is important since it allows developers to reuse existing C++ code without having to use the Java Native Interface [16, 25] and hence without having to deal with different languages and the conversion of different type systems. As we have explained in Section 5, this need has recently been identified by other researchers as well. Although similar, these products are slightly incompatible to each other. Hence, we think that the standardization of a C++ Servlet Environment will be important for the future of this technology.

Availability

The C/C++ Servlet Engine is freely available under the GNU General Public License. A more detailed description of the design and implementation of the CSE can be found in [23]. The CSE as well as its documentation is available for download from the CSE homepage at `http://www.infosys.tuwien.ac.at/CSE/`.

Acknowledgements

We would like to thank Dave Beazley, Andreas Grünbacher, and the many anonymous reviewers for their helpful comments. We also gratefully acknowledge the financial support provided by the USENIX Advanced Computing Systems Association and by the European Union as part of the EASYCOMP project (IST-1999-14191).

References

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

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

[3] Ape Software. The Servlet++ Homepage, 2002. `http://www.apesoft.net/servlet++/`.

[4] David M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the USENIX Fourth Annual Tcl/Tk Workshop*. USENIX, July 1996.

[5] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating user-perceived quality into web server design. Technical Report HPL-2000-3, Hewlett-Packard Laboratories, January 2000.

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

[7] 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. RFC2616.

[8] Pankaj K. Garg, Kave Eshgi, Thomas Gschwind, Boudewijn Haverkort, and Katinka Wolter. Enabling network caching of dynamic web objects. In *Proceedings of the 12th International Conference on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation*. Springer-Verlag, April 2002.

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

[10] 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.

[11] Arun Iyengar, Mark S. Squillante, and Li Zhang. Analysis and characterization of large-scale web server access patterns and performance. *The World Wide Web Journal*, 2:85–100, 1999.

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

[13] 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.

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

[15] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Pthreads*. Prentice-Hall, 1997.

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

[17] Micronovae. The Micronovae CSP Engine Homepage, 2002. `http://www.micronovae.com/CSP.html`.

[18] Mort Bay Consulting. Jetty—Java HTTP server and servlet container, 2002. `http://www.mortbay.org/jetty/`.

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

[20] Netcraft. Netcraft Web Server Survey, October 2002. `http://www.netcraft.com/survey/`.

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

[22] Peter Rossbach and Hendrik Schreiber. *Java Server and Servlets: Building Portable Web Applications*. Addison-Wesley, March 2000.

[23] Benjamin A. Schmit. A C++ Servlet Environment. Master's thesis, Technische Universität Wien, 2002. Draft version.

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

[25] Sun Microsystems. *Java Native Interface Specification*, May 1997. `http://java.sun.com/j2se/1.4/docs/guide/jni/`.

[26] Sun Microsystems. *Java Servlet Specification (Version 2.3)*, August 2001.

[27] Sun Microsystems. *The JavaServer Pages Specification (Version 1.2)*, August 2001.

[28] Guido van Rossum. *Python/C API Reference Manual*. Python Labs, October 2002.

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

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