AMR EL-KADI, AHMED NASHED, KAREEM EL GEBALY, MAHMOUD ABO DAOUD, NOHA EL SHARAWY, RANIA NAZMI, AND MOSTAFA MAZEN

architecture and internal design of the AUC-Abyss Web server

Dr. Amr El-Kadi is an associate professor of computer science at the American University in Cairo. He was a member of the IEEE-CS/ACM Joint Task Force on Software Engineering Ethics and Professional Practices (SEEPP). Dr. El-Kadi is a senior member of IEEE, ACM, and Eta Kappa Nu.

■ *elkadi@aucegypt.edu*

WITH THE ADVENT OF THE INTERNET, the need to deliver highly available scalable e-business systems has grown exponentially. The Web is transforming how companies transact business, communicate with their customers and business partners, and, ultimately, compete. Information technology departments are being asked to deliver and maintain systems that transact with customers around the clock, share data across the Internet, and generate large amounts of revenues. The penalty for downtime or slow response times in this environment is immense.

The term "Web service" describes specific functionality, value delivered via Internet protocols, for the purpose of providing a mechanism for another service or application to use [22]. Web services enable the specialization and reuse of traditional Web applications by exposing components of applications as Web services and enabling businesses to invoke these components. Web services will fundamentally transform Web-based applications by enabling them to participate more broadly as an integrated component of an e-business solution.

The industry is attempting to take advantage of World Wide Web Consortium (W3C: see http://www.w3c.org) and Internet Engineering Task Force (IETF: see http://www.ietf.org) standards, such as Extensible Markup Language (XML), HTTP, and Domain Name System (DNS) protocols to create specifications that define a way to publish and discover information about Web services. An example is the Universal Description, Discovery, and Integration (UDDI) specification.

Developing a scalable Web service requires developing an infrastructure to address a few fundamental challenges related to offering a service:

- Unpredictable loads, unreliable communications, and unreliable access
- Hardware scaling (i.e., the ability to arbitrarily throw hardware as scalability challenges)
- Integration (i.e., the ability to interoperate with other systems and services)

Before the Web, most communications between applications in the client-server world were synchronous. The client sent a message and then waited for the server to respond. In most synchronous situations, there was a predictable load, a simple response over a reliable communication infrastructure with reliable access (i.e., high service availability). For some situa-

tions now, a tightly coupled or synchronous service is acceptable. However, by virtue of being on the Web, the service can be exposed to unpredictable loads from unknown users over an unreliable communication infrastructure with unreliable access (i.e., can't predict the availability of other systems or Web services). On the Web, synchronous applications are often too fragile and inefficient to handle this level of uncertainty, and a loosely coupled, or messaging-based infrastructure, architecture is required.

A scenario arises where a single machine, no matter how the service is architected, does not possess the processing power required to handle Web service requests. In this scenario, Web services are deployed on a distributed multiserver architecture. As developers apply more hardware to solve critical scaling challenges, they potentially increase the complexity by introducing new factors into the architecture. Key factors such as load distribution, state management, and caching must be taken into account.

Web services are tautologically provided by Web servers, of which the Apache Web server is known to be the most widely used. The October 2003 Netcraft Web Server Survey reported that more than 64% of the Web sites on the Internet are using Apache, thus making it more widely used than all other Web servers combined. Since the 1.0 release (December 1, 1995), Apache has had a modular architecture (a feature unchanged until today [1]). Other notable Web servers include IIS (now IIS 6.0 for Windows Server 2003 [15,16]) with less than 40% of the market share, Zeus [17,18], and Flash [2,19,20]. While these Web servers outperform Apache in some aspects, they have some restrictions, such as being tied to a specific operating system or having high cost.

Three primary techniques enable the Web to handle high traffic loads: replication (mirroring), distributed caching, and improving server performance. Replication is simply duplication of Web information (either as a whole or partially) on multiple machines that either form a cluster [5] or are loosely coupled. Since any one of the machines can serve requests independently, the load of each individual server is reduced. Distributed caching includes client-side caching [6], proxy caching [7,8,9,10], or dedicated cache servers [11,12,13]. These approaches transparently cache documents closer to the clients, thereby reducing the network traffic as well as the overhead on the Web server. The effectiveness of Web caching is sometimes deemed obsolete when Web owners use cache-busting, that is, marking Web objects with a no-cache header, a technique used whenever Web owners are interested in collecting hit counts to track object popularity and usage patterns. Finally, improving server performance includes using more powerful hardware (e.g., hardware with SMP [Symmetric Multiprocessing] capability), better Web server software techniques (e.g., preforking process pools [14]), and high-bandwidth network connections [4].

Web servers, being crucial software systems, should normally benefit from advances in software engineering techniques and technology. Yet lots of software developers feel that such techniques will restrict the creativity of the developers as well as affect the performance of their products, so they elect not to use any well-defined process. We wanted to experiment with new modeling languages (such as UML), new iterative and incremental development methodologies (such as the Unified Process), new software architectures for distributed systems (such as peer-to-peer architectures), new testing techniques, and new performance evaluation methods. Open source software provides great opportunities for researchers not to start from scratch and for reuse, yet that is only possible for minor changes; making major changes to Apache was impossible, as we only have the code and neither models nor detailed designs.

Wanting to build a new Web server to experiment with all of the new technologies, we have reverse-engineered Apache and started to develop our own server. It seemed logical to us to concentrate on stand-alone servers (not clustered or

distributed servers) as a starting point, and yet ensure that their architecture would be extensible to support real businesses' ability to implement serious Web services. The beginning was a Web server called Artemis that reused many of Apache's modules and was written in C++. Artemis had good performance—close to that of the Apache version when it was developed but with a cleaner design. The goals of the AUC-Abyss project were to get rid of the many restrictions imposed by reusing Apache's modules (as they were not object-oriented by nature) and to be able to experiment freely with all of the new software development technologies.

We had as our primary objective producing a well-engineered stand-alone Web server that could outperform Apache 2.0 in comparable environments and at least support static files, fully support HTTP 1.1 requests, provide a good server-side caching mechanism, provide an efficient logging mechanism, and be both reliable and extensible.

In the following discussion we first provide some background on how Web servers function in general, while highlighting Apache's internal architecture. We then provide a summary of techniques available to handle incoming requests in parallel, detailing the model used by AUC-Abyss. The architecture of our Web server is then given and further details are revealed using static and dynamic artifacts. Before concluding, we compare the performance of AUC-Abyss against Apache.

Background

A centralized Web server, in the simplest form, could be perceived as a passive process.[1] Clients open TCP connections with the Web server and send their requested content using the HTTP protocol [1]. Since several clients may be issuing their requests in parallel, such requests are queued on the server's port. The server de-queues requests, finds the requested file, and (if found) sends an HTTP response header followed by the requested data (see Figure 1). This simple sequence is followed for satisfying static content (content that is accessible to a Web server in disk-file form). For clarity, we will not consider dynamic content (content that is generated dynamically by executing auxiliary applications) in the following discussion.

1. A passive process, as opposed to an active one, is a process that does not initiate computation. Instead, it remains dormant when there are no requests to serve and is activated by the operating system as soon as a request reaches its ports.

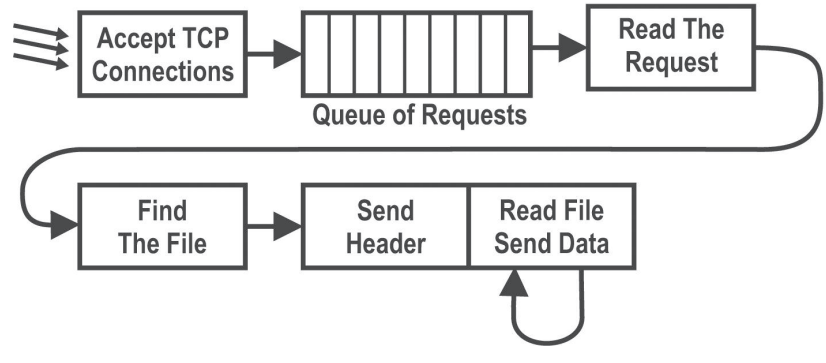

FIGURE 1: HANDLING OF HTTP REQUESTS

The open source model has stimulated the development of Apache functions by many volunteer programmers (and even recently by IBM), resulting in a fairly rapid pace of functional enhancements. Apache's modularity permits its users to pick and choose modules to fit their requirements. It is claimed that it can serve a large number of concurrent clients, limited only by the underlying hardware and operating system. The hybrid threading/multiprocess model increases its scalability. The Apache Portable Runtime layer (APR) means it can run at its

best on multiple platforms, which now include everything from common UNIX variants, the Microsoft Windows family, and NetWare, to OS/2.

The server can be configured easily (statically or dynamically) either by editing text files or by using one of the many available GUIs. Its modularity allows many features that are necessary within special application domains to be implemented as add-on modules and plugged into the server. To support that, a well-documented API is available for module developers. Its modularity and the existence of many free add-on modules make it easy to build a powerful Web server without having to extend the server code. Using many of the available server-based scripting languages, Web-based applications can be developed easily. When using scripting languages or add-on modules, Apache can even work with other server applications such as databases or application servers. Therefore, Apache can be used in common multi-tier scenarios. Additionally, Apache is completely HTTP 1.1 compliant in both of the current versions, and it also supports the HTTP compression enhancement tool, thus saving bandwidth, a feature heavily used by Google in running Apache as its Web server.

Since our target was to perform better than the Apache Web server, it was logical to attempt to understand the issues that affect its performance [4]. The single biggest hardware issue affecting Web server performance is RAM. A Web server should never have to swap, since swapping increases the latency of each request beyond a point that users consider "fast enough." This causes users to hit "stop" and "reload," further increasing the load. Too many clients attempting to connect to an Apache Server at one time can spawn child threads to the point where the need for memory swapping leads to a severe performance problem.

Concerning the issue of process creation (thread spawning), Apache's available threads are not always sufficient to accept all incoming requests, so constant per-second spawning is required. In addition, Apache's parent and children communicate with each other through something called the scoreboard. Ideally, this should be implemented via shared memory, which is the case for those operating systems that the designers had insight into. The remaining implementation of the Apache Web server defaults to using an on-disk file, which is both slow and unreliable (and less featured).

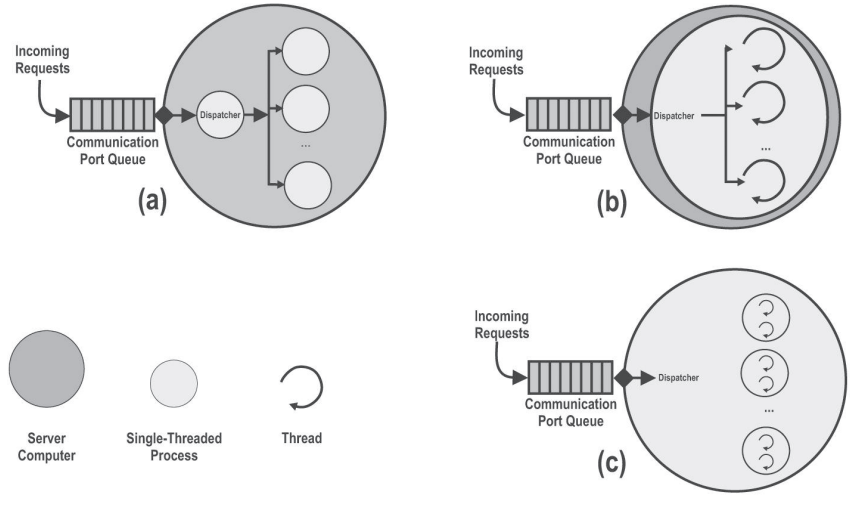

FIGURE 2: THREE MAIN PROCESSING MODELS

Parallel Handling of Requests

Before we detail the AUC-Abyss Web server architecture, it is important to discuss how clients' requests are actually handled. Web servers parallelize the han-

dling of clients' requests to exploit interleaving processing with I/O requests, thus reducing overall response time.

Two main issues greatly affect the performance of a Web server: the processing model, and the pool size behavior [3]. The processing model describes the parallelism adopted by the Web server in terms of processes and/or threads, while the pool size behavior specifies how the number of processes (or threads) varies over time in response to workload.

Three main options for a processing model are used by Web servers: the process-based model, the thread-based model, and a hybrid model (see Figure 2). In the process-based model (see Figure 2a), a dispatcher process receives requests from the queue and sends them to single-threaded processes for handling. In the thread-based model (see Figure 2b), a single multi-threaded process receives requests from the queue and assigns each to one of its own threads (lightweight processes). The hybrid mode has a dispatcher (which is a single-threaded process) that receives requests from the queue and sends them to multi-threaded processes (see Figure 2c). Each of these models has advantages and drawbacks, summarized in Table 1.

	Pros	Cons
Process-Based	Stability of the system. If a process goes down, the only effect would be the failure of the client being served by that process, without any other effect on the system.	High cost for creation and destruction of processes. Memory requirements are much less because threads share the same address space. Huge context-switching overhead.
Thread-Based	Memory requirements are much less because threads share the same address space. Spawning threads within the same process is much more efficient than spawning new processes. Much efficient inter-thread communication through the use of the shared address space.	Not as stable as the process-based model; one malfunctioning thread will take down the whole server.
Hybrid	It combines the pros of both models; if a thread crashes, it would take down the process that created it and its sibling threads. This means that some of the clients will be disconnected but not all of them.	

TABLE 1: PROS AND CONS OF EACH OF THE THREE MAIN PROCESSING MODELS

Any of the three processing models has one of two options for coping with varying workloads by controlling how the number of processes (or threads) varies over time (i.e., pool size behavior). In the first approach, a static pool is used in which a fixed number of processes (and/or threads) are created at startup. As a request arrives, it is more likely that this request will find a process already spawned ready to serve it, so no time is wasted on spawning or killing processes or threads. However, when the load on the Web server is low, many processes (or threads) will remain idle (wasting a lot of cycles and forming more switching overhead, especially if they use polling and do not block). Also, if there are p processes (or threads) already created and a request arrives finding other p requests being processed, the request will wait in a queue. As the workload for the Web server increases, the queue will get longer, increasing the response

time. In the second approach, a dynamic pool is used in which the creation and destruction of processes (and/or threads) varies dynamically according to the workload. This means that when the load is increased, more requests will be processed concurrently and the queue will be reduced. Yet at the same time, the dynamic creation and destruction of processes (and/or threads) does introduce an overhead for the server machine.

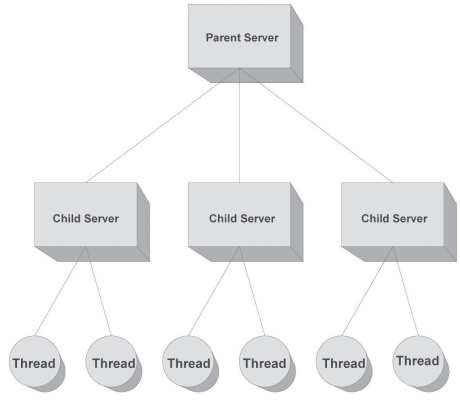

FIGURE 3: AUC-ABYSS PROCESS MODEL

We have decided to use the hybrid processing model for AUC-Abyss. In this model, processes as well as threads do not need to interact at all, since each thread serves a different request independently of the others. As our Web server starts, a single-threaded process root loads a configuration file to set up and configure its consequent operations. It starts to allocate and initialize pre-configured memory in RAM for its use. Once this phase is concluded, the root process forks another process and kills itself. That new process is the parent server (see Figure 3), which, in turn, is responsible for forking more child servers, depending on the workload and the condition of the currently running child servers. (See Figure 4a for the use-case diagram to serve an HTTP request.) Each child server spawns a number of threads by which the requests are actually handled (see Figure 4b for the use-case diagram of child servers). The fact that each thread handles a request independently improves the robustness of the server by reducing the likelihood of events that may cause a systemwide failure. However, the acceptance of new requests is synchronized through a mutual execution mechanism in order to make sure that each request is served only once. Furthermore, the child servers communicate with the parent server through the scoreboard, a file in which each child saves its current state.

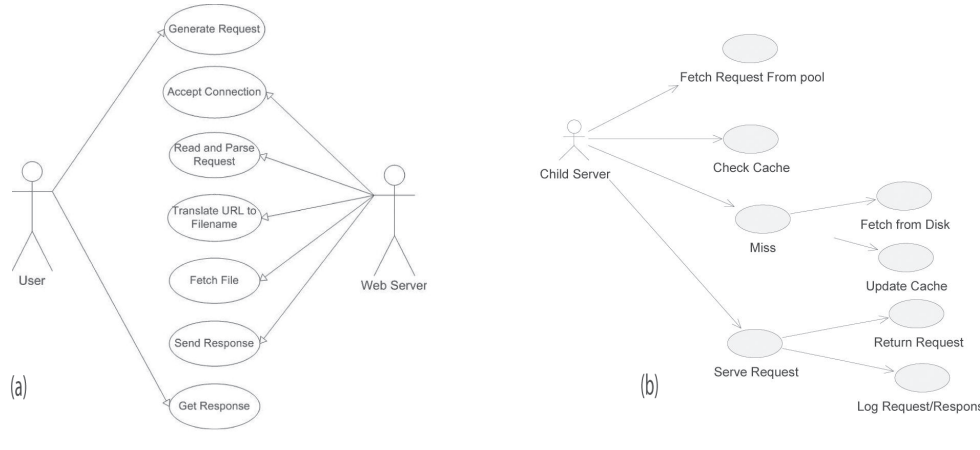

FIGURE 4: MAIN USE-CASE DIAGRAMS

Server Architecture

AUC-Abyss is based on a hybrid architecture in which many threads, spawned by child server processes, serve requests in parallel; this might dramatically decrease Web server performance, since file I/O operations are the most expensive operations a thread can perform as it competes with other threads (at least to access the log file). Thus, our primary concern was to find a way to reduce the cost of repeated I/O operations that occur after each request-response cycle. Our initial solution was to create a single global memory base buffer in which we could temporarily store the logging information and to call a periodic dumping function which would copy all this information to the log file on the hard disk. This was thought to improve performance, since it reduced the overhead of opening and closing a file stream for each single served request by buffering a large number of entries together and writing them in one chunk.

Our second concern was to maintain the performance of the Web server: the level at which a Web server is able to perform under a certain workload should remain constant, even while the server is flushing logging information. This raised a problem: once the flushing operation is underway, the memory buffer locks and cannot be accessed, the threads therefore are all forcibly put to sleep (since they cannot log after serving), and for a few seconds the server grinds to a complete halt. This was unacceptable. We came up with a twofold solution to this problem. First, and most important, it was decided that the flushing operation could not be performed by the threads themselves, since this reduces performance dramatically. Either the parent server process would perform this operation or a new twin thread (also known as a shadow thread) would be spawned to perform this operation, then die. Second, we decided to implement a mirror buffer, which performs exactly like the base buffer but is used as a backup. When the base buffer is being flushed, logging is automatically shifted to the mirror buffer and vice versa; thus request handling will never stop. Concerning the dumping of the logs from the memory, once a buffer is filled, the logging mechanism is responsible for spawning a twin thread for transferring this information onto the disk concurrently with the normal operation of the Web server, and therefore the performance of the Web server is not affected (except for using up some extra clock cycles). This twin thread is considered a twin to the threads in the process that made the final entry into the logging buffer. The parent of this twin thread is random and is not specified a priori.

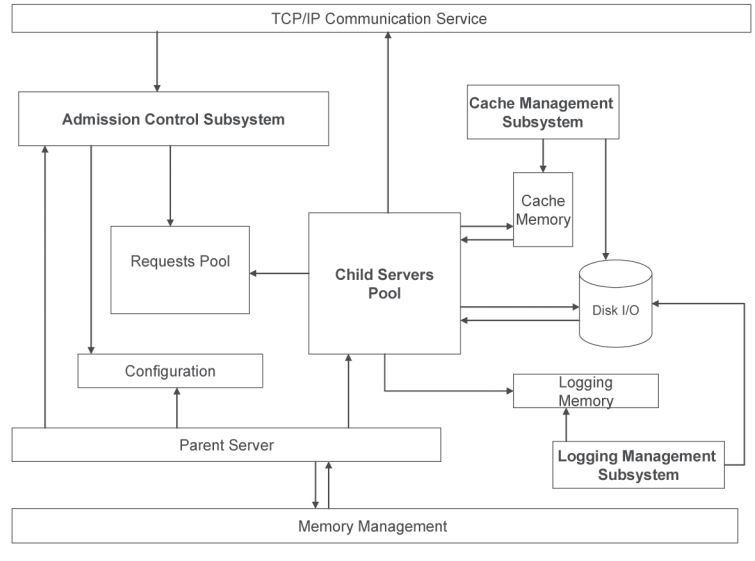

FIGURE 5: AUC-ABYSS ARCHITECTURE

Figure 5 shows the architecture of the AUC-Abyss Web server. Memory management is the most important component; it interacts with or is used by all the different entities in order to utilize memory effectively, avoiding leakage and reducing system calls. The configuration layer is responsible for the different types of configuration we handle in the system and is saved in a text file that is read when the server starts to boot. It deals with almost all the other entities that exist in our Web server.

The parent server controls the whole Web server. Since AUC-Abyss is a preforking Web server, the parent server is responsible for creating the child servers (processes) which spawn multiple threads that become responsible for handling the request-response cycle. The parent server communicates with the memory management, the configuration, and the child servers.

Once the child servers are forked, they handle all the request-response cycles of the system. The request enters through the TCP/IP interface and the admission control and is kept in a queue in the request pool. Child servers take requests and handle them; they look for them in the caching subsystem and, when done, log the operation through the logging subsystem. Finally, the TCP/IP layer provides the basic networking capabilities that the Web server needs to process its different activities. It interacts with the parent server and the request-response layer.

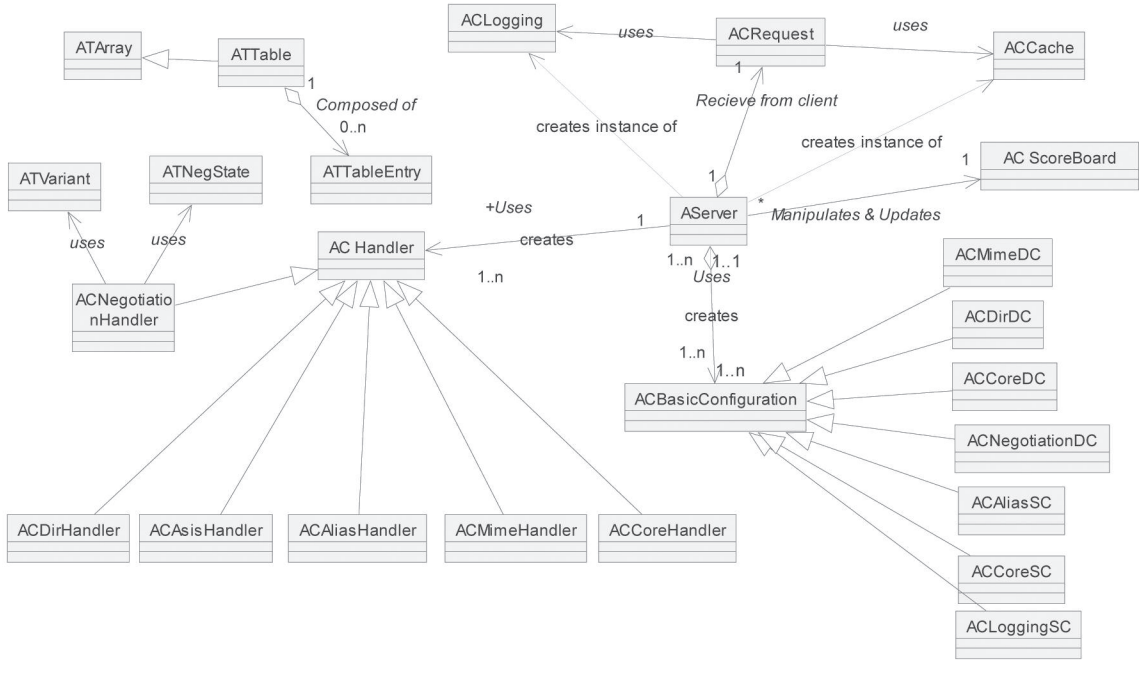

FIGURE 6: GENERAL CLASS DIAGRAM

Logical View

Our general class diagram more or less maps the system architecture, using several components (see Figure 6). The configuration subsystem consists of an abstract virtual class (ACBasicConfiguration); all other classes implement this basic class. This fosters extensibility by future addition of subconfiguration components as long as such components inherit from the abstract class and implement all its virtual functions. Two main ideas drive the existence of an abstract class. First of all, reusability suggests that all common operations and attributes be grouped into one class. Secondly, such a class facilitates adding new directives if the administrator desires. For example, an admin can create a new class in which each directive is saved with a function pointer to execute

when this directive is met in the configuration file. This class can be dynamically linked to the server and provide the extra functionality needed.

The ACCache class communicates with ACRequest during the service of the request; the logging class deals with the ACRequest class as well. The rest of the classes (AServer, ACRequest, and ACHandler) handle the request-response cycle. ACScoreboard is responsible for keeping track of the spawned processes and threads (child servers).

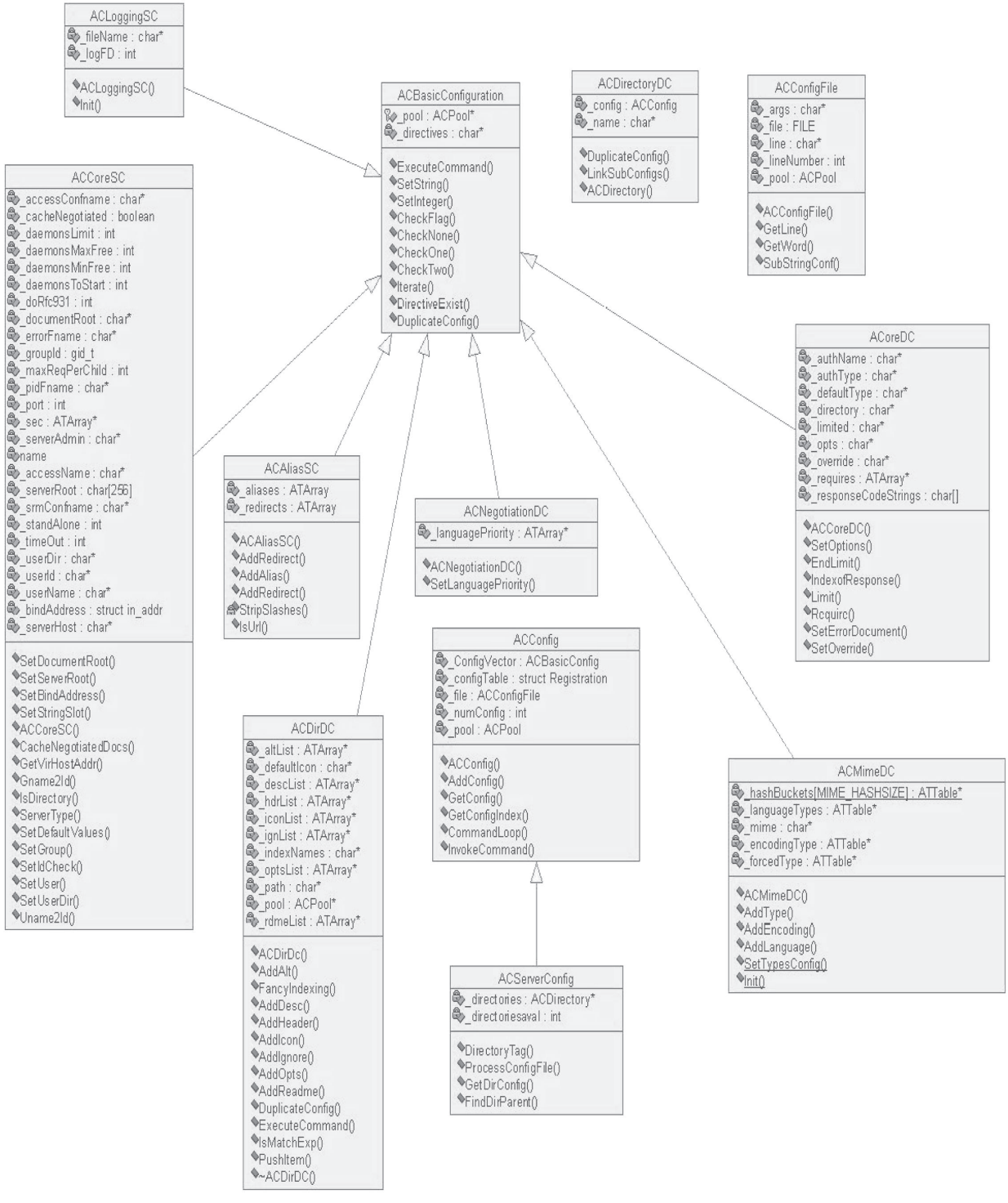

FIGURE 7: CONFIGURATION SUBSYSTEM CLASS DIAGRAM

The server cannot function properly without a configuration. There are two types of server configuration: per server, or per directory. The configuration is saved in a text file which is read when the server starts up. The configuration layer is responsible for the configuration of the server (see Figure 7). This subsystem deals with almost all the other entities that exist in the Web server. The naming schema is uniform to differentiate between various subconfigurations: those belonging to per-server configuration end with SC, whereas those belonging to per-directory configuration end with DC. The server configuration can either be a preloaded or an extra per-server configuration that can be added later on. The per-directory configuration can be either default or special.

For per-server configuration, the parent server, after creating the pool of memory, interacts with the configuration object to initialize the server configuration needed to process the various servers' activities. For example, the port number and document root are set by this object, among other variables needed by the server to start processing different requests. A preloaded per-server configuration is the basic configuration; the extra per-server configuration gives the user the ability to customize some of the configurations after the default configuration has been loaded.

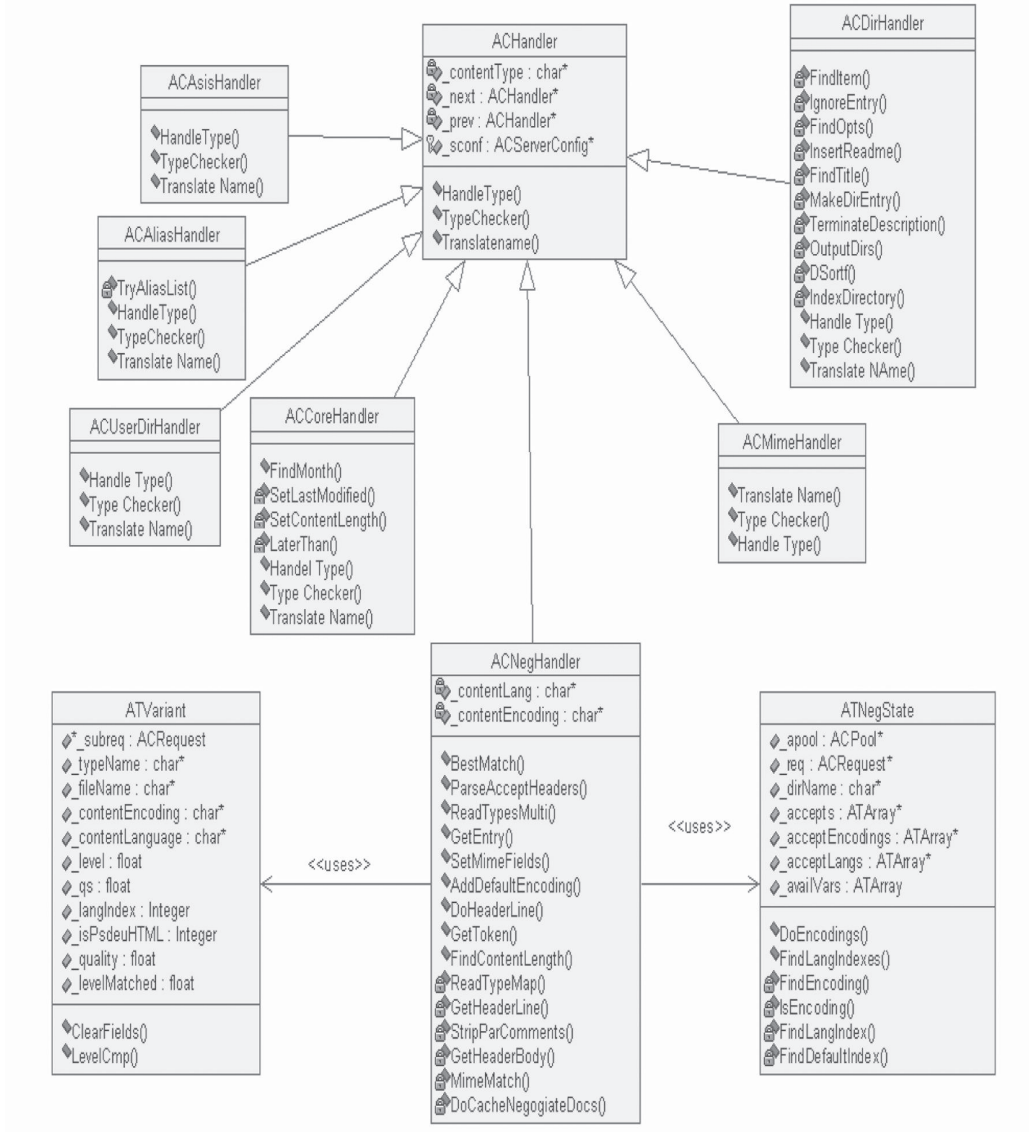

FIGURE 8: HANDLERS CLASS DIAGRAM

Per-directory configuration is responsible for initializing different directories existing in the system, both default and special configurations. Once again, the parent server interacts with the directory objects (directory, MIME, core, and negotiation) to initialize the basic configuration of each. To set up the default per-directory configuration, the parent server initializes first the server configuration and then the necessary directory configuration. This configuration is to be used when no special values are specified. For example, /usr/local/etc can use the default configuration /usr/local directly. the user can save special directory configurations, after the default initialization. These values will of course override the default values.

AUC-Abyss uses different handlers for different types of requests. The ACHandler base class contains the basic methods needed by all handlers to respond to requests. Each type of handler is a class inheriting from the base class. There are two other classes that are not handlers per se in the diagram: ATVariant encapsulates information about a particular variant, and ATNegState deals with the state of negotiation. AUC-Abyss deals with handlers as a linked list containing an instance of each. Once the server receives a request, it will loop over all these handlers and check the content type to know which one will be used in handling this specific request. This design is useful for reusability purposes.

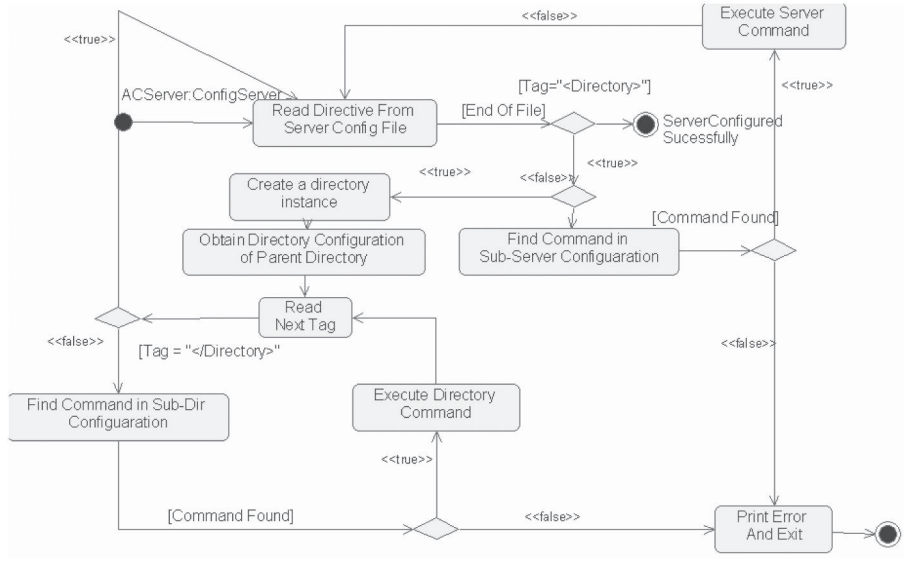

FIGURE 9: CONFIGURATION STATE DIAGRAM

Dynamic Behavior

As the root server process is responsible for the configuration of the server, it starts by executing `configServer()`. This function opens the configuration file and reads the directives (see Figure 9). Reaching end of file means the configuration was completed successfully. Each directive is read and then searched for in the command list. If there is a match, the function associated with the directive is executed and the process is repeated for the next directive. If it is not found, an error message is printed and the system is exited.

One important directive is "directory," which specifies that the user wants to create a special directory that should have a specific per-directory configuration. The server creates a directory instance and then initializes it with the default configuration by checking its parents' configuration until we have a complete directory configuration. The server then proceeds by reading another directive from the file, but this time the search is made in the table of subdirectory configurations. If a match is found, the associated function is executed and the

process is repeated until the closing tag for the directory is reached. This indicates that the special directory configuration has ended, and the server goes back to configuring the server.

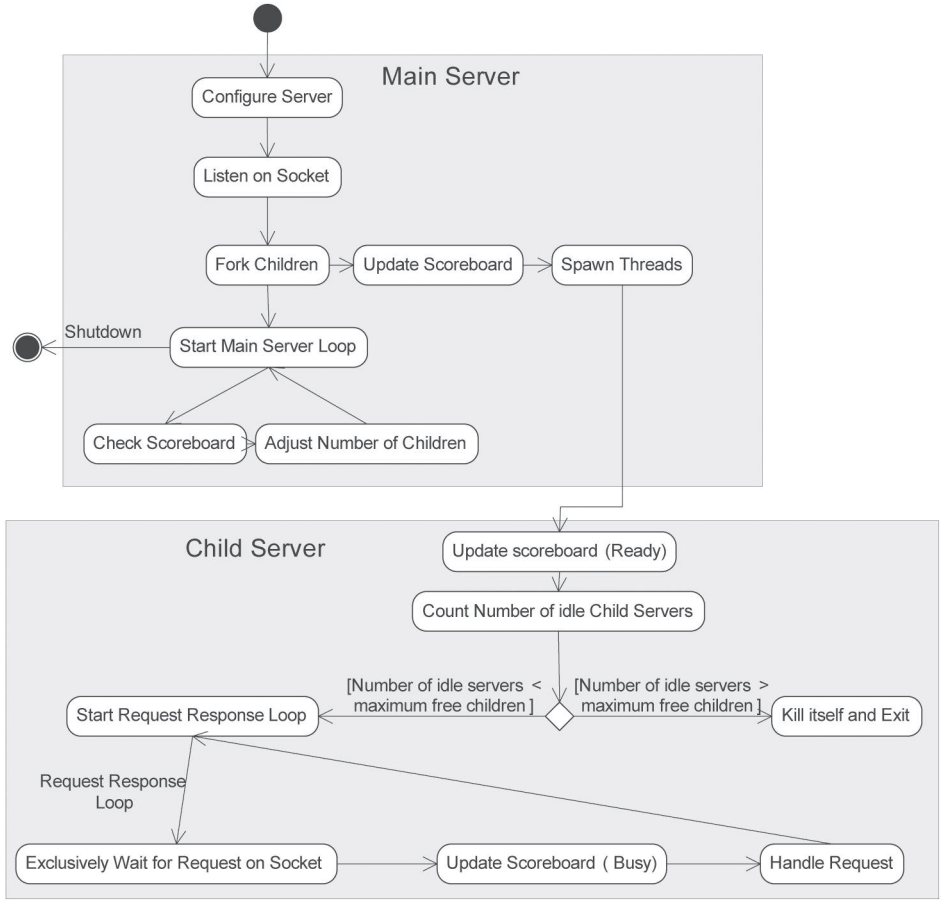

FIGURE 10: MAIN SERVER AND CHILD SERVER STATE DIAGRAM

Figure 10 shows the overall state diagram of the Web server. It starts the execution of the root process that is responsible for the entire configuration. It then opens a socket and keeps on listening for the incoming requests. Then it forks a number of processes, each of which in turn spawns a number of threads and updates the memory with the new status. Each of these threads represents a child server class responsible for handling a request-response cycle. When a thread is spawned, it updates its status in the scoreboard to "Ready" and counts the number of idle children. If that number is more than the maximum number allowed, the thread is killed and exits the connection; otherwise it starts the request-response loop. At the beginning of this loop, the thread (or child) is solely responsible for waiting on incoming requests on the socket; when a request arrives it is accepted and then handled. It then goes back to the start of the request-response loop. Meanwhile, the main server is in another infinite loop, maintaining the child statuses and the scoreboard. Note the hierarchy of the system: the main server represents the parent responsible for a number of child servers, whereas each child server is responsible only for handling a request.

Request-Response Sequence

The request-response cycle (shown in Figure 11) begins with an instance of the AServer class (see Figure 12). This object is the main core controller of the Web server and begins by creating instances of ACCache andACLogging, followed by the ACHandler classes. The AServer is solely responsible for handling the request-response cycle. It creates an ACRequest object and then loops infinitely, waiting for a connection from a client through the WaitForConnection() function. Once it is notified of a pending request connection, it proceeds to open the input/output connections through the ACRequest object initialized earlier through the sockets (OpenInputConnection() and OpenOutputConnection() functions). It is then concerned with processing the request by reading the request, parsing the URL, and obtaining the content type of the request through three functions: SendBasicHeader(), SendHttpHeader(), and SendFile(). Once this is done, the ACChildServer asks the ACServer controller to get the handler suitable for handling this type of request. Thus, the handler type is returned through a response message. The child server then proceeds to ask the ACHandler object to handle this specific type of request.

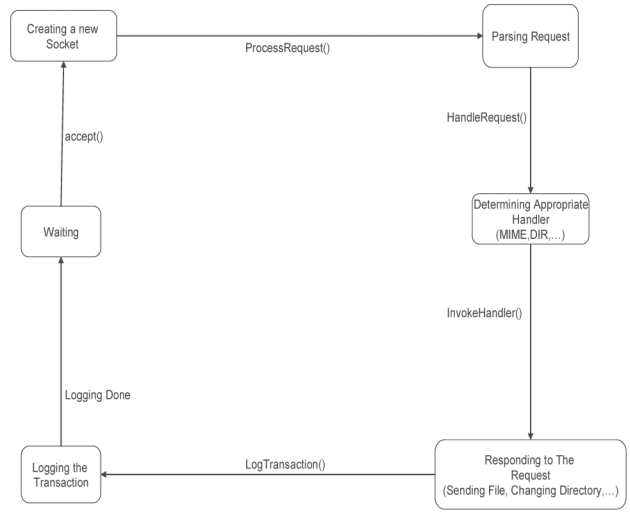

FIGURE 11: TYPICAL REQUEST-RESPONSE CYCLE

In responding, the ACHandler will first look up this request in the cache memory by using the lookup() function in the ACCache instance that was initialized earlier by the ACServer. A response is then sent back to the handler from the ACCache containing a pointer to the requested data in memory if it is found there. If not, a pointer to its location on the hard disk is returned. The child server uses this pointer to perform the response part of the cycle. This is achieved by sending the Basic Header, the HTTP Header, and the file itself through the functions previously mentioned in the ACRequest object back to the client through the OpenOutputConnection() of the socket. After the response is delivered, the child server is responsible for storing the transaction in the log buffer. This is accomplished by using the insert() function in the ACLogging object.

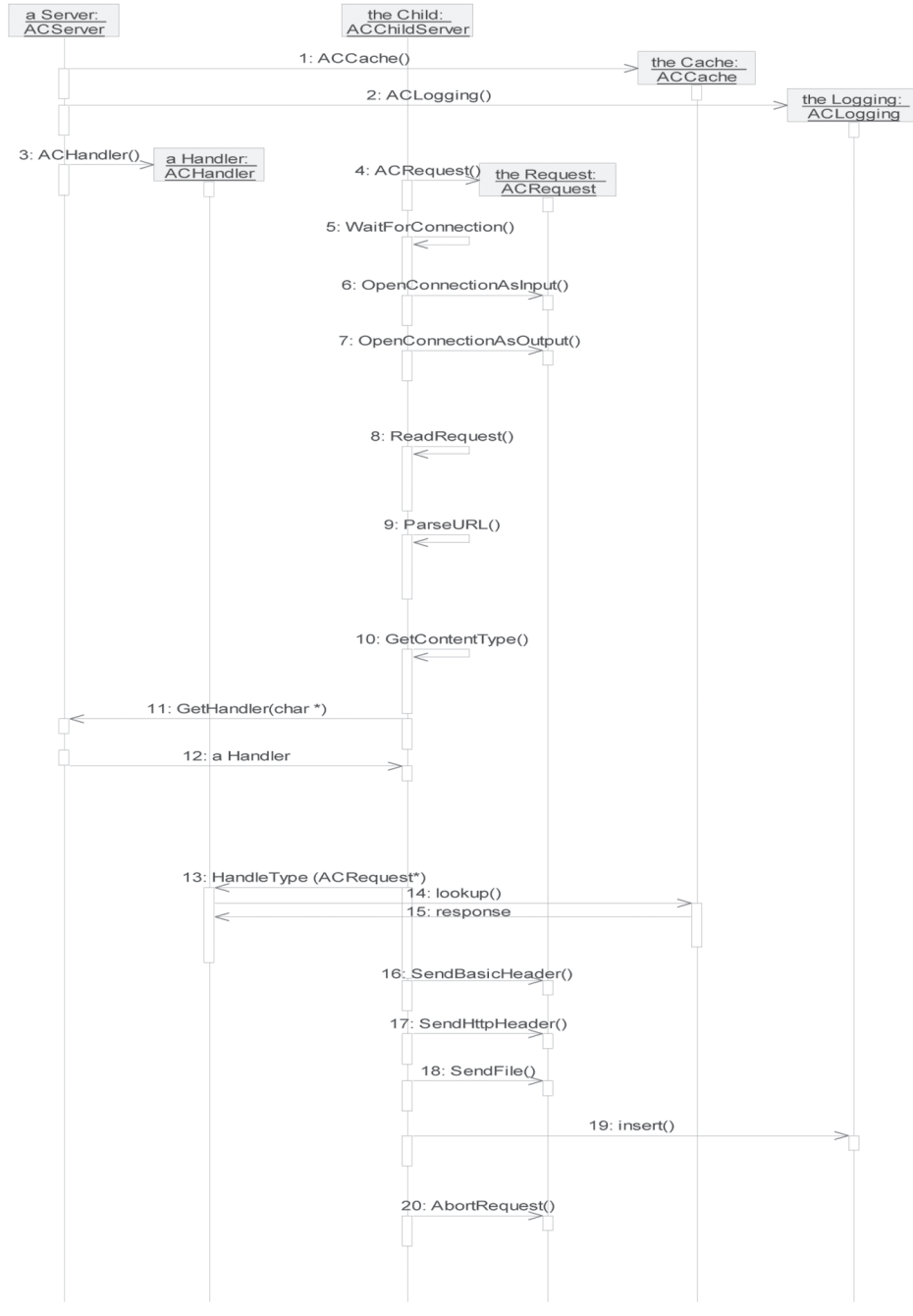

FIGURE 12: REQUEST-RESPONSE SEQUENCE DIAGRAM

ALICE 15 Kbytes	Test 1	Test 2	Test 3	Average	% Difference
AUC-Abyss	2	2	2	2	21.70%
Apache	2	3.3	2	2.4333333	-21.70%
The number of connections was 50,000 with a rate of 100 connections per second. The Net I/O of the network was 12.5 Mbps.					
BECKY 256 Kbytes	Test 1	Test 2	Test 3	Average	% Difference
AUC-Abyss	26	28.2	30.6	28.266667	2.90%
Apache	27.9	31.5	28.1	29.166667	-2.90%
The number of connections was 20,000 with a rate of 44 connections per second.					
The Net I/O of the network was 92.3 Mbps.					
CANDY 512 Kbytes	Test 1	Test 2	Test 3	Average	% Difference
AUC-Abyss	50.8	49	48.8	49.533333	1.55%
Apache	52.8	49.2	48.9	50.3	-1.55%
The number of connections was 10,000 with a rate of 22 connections per second. The Net I/O of the network was 92.3 Mbps.					
DOROTHY 1 Megabyte	Test 1	Test 2	Test 3	Average	% Difference
AUC-Abyss	95.8	97	97.2	96.666667	0.30%
Apache	97.2	97	96.7	96.966667	-0.30%
The number of connections was 5000 with a rate of 10 connections per second. The Net I/O of the network was 88.3 Mbps.					
EDITH 2 Megabyte	Test 1	Test 2	Test 3	Average	% Difference
AUC-Abyss	189.7	190.7	189.6	190	0.14%
Apache	189.6	190.6	190.6	190.26667	-0.14%
The number of connections was 2500 with a rate of 5 connections per second. The Net I/O of the network was 88.3 Mbps.					

TABLE 2: REQUEST-RESPONSE TIME TEST RESULTS

Performance Evaluation

The first set of performance evaluation benchmarks was concerned with testing the request-response time of both Apache 2.0 and AUC-Abyss. We used httperf (a standard benchmarking tool for evaluating Web servers), taking three samples for every file size and repeating the experiment for five different file sizes (15KB, 256KB, 512KB, 1MB, and 2MB), each test running for 10 minutes. By looking at the results (see Table 2), we notice that in small file sizes we outperformed Apache by a fairly obvious margin, but as file sizes grew in size, the performance of Abyss converges with that of Apache. This is due to network saturation as the network I/O reaches its maximum. It's also important to note that a file of 15k is the most common file size requested on the Internet in general, and in that case AUC-Abyss outperformed Apache by an average of 22%.

We then moved on to the width tests, focusing on evaluating the server's concurrency performance when numerous small files were requested. Concurrency is of vital importance here, especially for Web sites visited by millions of people (e.g., google.com or hotmail.com). These tests were carried out on both AUC-Abyss and Apache. Once again, each test was repeated three times and performed using four small file sizes (20B, 1KB, 2KB, and 4KB). All tests were at a constant rate of 2500 connections per second, which worked out to 150,000 total connections.

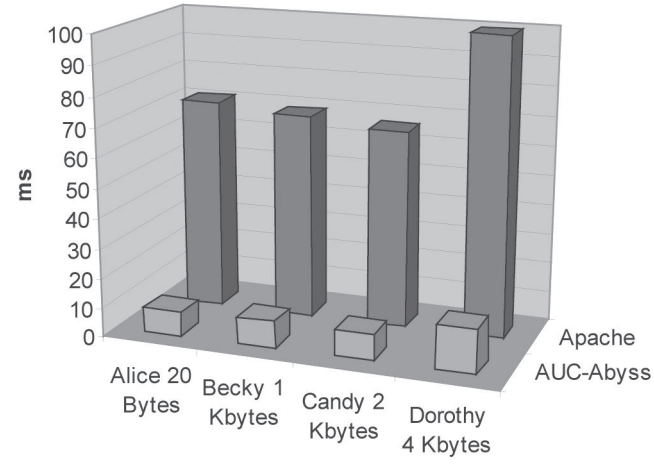

FIGURE 13: AVERAGE RESPONSE TIME

The results indicate that Abyss handled concurrency dramatically better than Apache (see Figure 13). It is important to note that Apache was inconsistent in its maximum number of concurrent users, which is unacceptable in an enterprise situation.

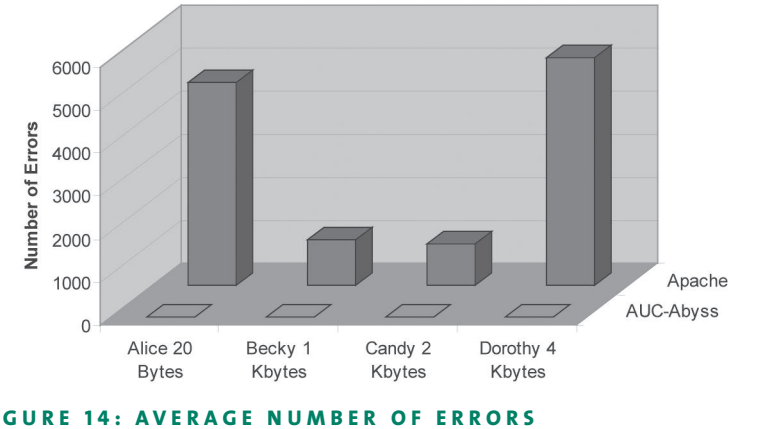

FIGURE 14: AVERAGE NUMBER OF ERRORS

The error performance of AUC-Abyss was outstanding, as it produced the least number of errors (zero errors, in fact) in comparison with Apache (see Figure 14). As a result of the number of errors, we can conclude that the Apache experienced denial of service whereas AUC-Abyss was still up and running.

Conclusion

We can safely state that our project met its goals. We have designed and implemented an extensible Web server using state-of-the-art software engineering technology that is on a par with the most widely used Web servers. But this is just the starting point. Much more work needs to be done to make AUC-Abyss as powerful as other Web servers. As yet, it does not provide sophisticated

admission control, it only supports static files, and it is a stand-alone, non-portable server.

Most Web server architectures reject excess requests without discriminating between different resource bottlenecks, or they use only one indicator for overload, often CPU utilization. Hence, they cannot take the potential resource consumption of requests into account, but have to reduce the acceptance rate of all requests when one resource is over-utilized [21]. Both high CPU utilization and dropped packets on the networking interface can lead to long delays and low throughput. Other resources that could be controlled are disk I/O bandwidth and memory. The admission control mechanism adaptively determines the client request acceptance rate to meet the Web servers' performance requirements, while the load balancing or client request distribution mechanism determines the fraction of requests to be assigned to each Web server (in the case of a distributed-based Web server architecture).

Adding admission control over and above basic load balancing reduces workload, increases server performance (faster response to users' requests), and maximizes the usefulness of server arrays. It is observed that admission control ensures that throughput is maintained at the highest possible level by controlling traffic to the Web servers when the servers' resources are approaching exhaustion. By controlling traffic before resources are exhausted, the chances of server breakdown are minimized, and hence system sanity and graceful degradation, in the worst case, are guaranteed. In addition, if admission control allows a user access to a Web server, the user will receive continuing priority access to server resources, thereby ensuring that the service a user perceives is maintained at an acceptable level.

To conclude, admission control plays a crucial role in ensuring that the servers meet users' quality-of-service requirements while maximizing site availability and preventing server congestion/failure during heavy traffic. Our next step for AUC-Abyss is to add admission control. The fundamental question here is, Is admission control really necessary in Web server systems? In response, we note that there are two ways to increase overall user utility, namely, increasing server (farm or cluster) capacity or implementing intelligent traffic management mechanisms. Our experiments show that we can utilize resource-based admission control to avoid over-utilization of critical Web server resources. We may also provide service differentiation using token buckets with logical partitions. The importance of having an admission control subsystem is accentuated with the support of both static and dynamic requests, mainly because the first is network intensive whereas the latter is CPU intensive. In a simple scenario the CPU could be causing a bottleneck when serving a high load of requests based on dynamic scripts, while the network and bandwidth are capable of serving static requests.

Right now the server only supports static HTML files. However, to be able to compete with Apache and other Web servers, AUC-Abyss needs to support dynamic scripts and CGI, which are commonly used nowadays. Therefore, one of the first possible future enhancements would be an add-on to support dynamic behavior. In addition, our Web server was built with C++ on Linux. Developing it in a standard programming language would make its portability easier, yet an operating system's dependency has to be architecturally addressed, and there are a lot of approaches that we can learn here from the development of portable operating systems to make AUC-Abyss portable across platforms. Other future plans for our server include adapting its architecture to provide distributed-based Web services and support for virtual servers [23].

The authors plan to release the Abyss server in the spring of 2005 under the GPL for research purposes only.

REFERENCES

[1] See http://apache.rcbowen.com/ApacheServer.html.

[2] V. Pai, P. Druschel, and W. Zwaenepoel, "Flash: An Efficient and Portable Web Server," *Proceedings of the 1999 USENIX Annual Technical Conference (Monterey, CA),* June 1999, pp. 199–212.

[3] Daniel A. Menascé, "Web Server Software Architectures," *IEEE Internet Computing*, vol. 7, 2003, pp. 78–81.

[4] See http://www.ele.uri.edu/Research/hpcl/Apache/journal_CA.pdf.

[5] E.D. Katz, M. Butler, and R. McGrath, "A Scalable Web Server: The NCSA Prototype," *Computer Networks and ISDN Systems*, vol. 27, no. 2, November 1994, pp. 155–164.

[6] A. Bestavros, R.L. Carter, M.E. Crovella, C.R. Cunha, A. Heddaya, and S.A. Mirdad, "Application-Level Document Caching in the Internet," *Proceedings of the 2nd International Workshop on Services in Distributed and Networked Environments (SDNE),* 1995.

[7] A. Luotonen and K. Altis, "World-Wide Web Proxies," *Proceedings of the First International Conference on the World-Wide Web,* 1994.

[8] M. Abrams, C.R. Standridge, G. Abdulla, S. Williams, and E.A. Fox, "Caching Proxies: Limitations and Potentials," *Proceedings of the 4th International Conference on the World-Wide Web (Boston, MA),* December 1995.

[9] C. Maltzahn, K.J. Richardson, and D. Grunwald, "Performance Issues of Enterprise Level Web Proxies," *Proceedings of the 1997 SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* June 1997.

[10] P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms," *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS),* December 1997.

[11] J. Gwertzman and M. Seltzer, "The Case for Geographical Pushcaching," *Proceedings of the 1995 Workshop on Hot Operating Systems,* 1995.

[12] S. Glassman, "A Caching Relay for the World Wide Web," *Proceedings of the First International Conference on the World-Wide Web,* 1994.

[13] A. Chankhunthod, P.B. Danzig, C. Neerdaels, M.F. Schwartz, and K.J. Worrell, "A Hierarchical Internet Object Cache," *Proceedings of the 1996 USENIX Annual Technical Conference (San Diego, CA),* January 1996.

[14] A. Cockcroft, "Watching Your Web Server," http://www.sun.com/sunworldonline/swol-03-1996/swol-03-perf.html, March 1996.

[15] See http://www.microsoft.com/windowsserver2003/evaluation/overview/technologies/iis.mspx, April 2003.

[16] Brian Livingstone. "Intel Blows Bandwidth," http://itmanagement.earthWeb.com/columns/executive_tech/article.php/3068161, April 2003.

[17] See http://linuxtoday.com/news_story.php3?ltsn=2001-01-29-005-06-PR-HE-SV.

[18] See http://www.zeus.com/products/zws/features.html, Web Server Feature Comparison.

[19] See http://www.cs.princeton.edu/~vivek/flash/.

[20] See http://www.csse.monash.edu.au/~impp/Docs/Thesis%20Final.pdf.

[21] Thiemo Voigt and Per Gunningberg, "Adaptive Resource-Based Web Server Admission Control," *Proceedings of the 7th International Symposium on Computers and Communications,* 2002.

[22] Adam Bosworth, "Developing Web Services," *Proceedings of the 17th-International Conference on Data Engineering,* 2001.

[23] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Philip S. Yu, "The State of the Art in Locally Distributed Web-Server Systems," *ACM Computing Surveys*, vol. 34, no. 2, June 2002.

Annual Technical Conference

May 30–June 3
Boston, MA

SAVE THE DATE!

2006 USENIX Annual Technical Conference

May 30–June 3, Boston, MA

Please join us at the 2006 USENIX Annual Technical Conference in Boston. USENIX has always been the place to present groundbreaking research and cutting-edge practices in a wide variety of technologies and environments and 2006 is no exception. Join the community of programmers, developers, and systems professionals in sharing solutions and fresh ideas.