Using Generative Design Patterns to Develop Network Server Applications

Zhuang Guo, Jonathan Schaeffer, Duane Szafron, Patrick Earl[x]

Abstract— **Design patterns are generic solutions to recurring software design problems. The Correct Object-Oriented Pattern-based Parallel Programming System (CO_2P_3S) uses design pattern templates to generate code for design patterns. CO_2P_3S has been used to generate small parallel and sequential applications. This research evaluates the utility and performance of CO_2P_3S on larger network server applications. The Network Server design pattern template is introduced, which significantly eases the complexities involved in network server application development. The Network Server is highly configurable and suitable for the construction of a large variety of network server applications, with a diverse range of functionality and performance requirements. In this paper we highlight a generated Web server with performance comparable to Apache.**

***Index Terms*—network servers, parallel programming, programming environments.**

I. Introduction

Network server applications are essential to networked computation, as they support sharing of critical system resources and provide services to multiple clients concurrently. Web servers, FTP servers, and e-mail servers are three examples of these applications. The continuing growth and usage of computer networks has put more demand on various new network server applications. However, their construction remains a difficult task, due partly to the limitations of conventional tools and techniques. In this research, we experimented with an innovative approach: using generative design patterns to construct network server applications.

Design patterns are proven solutions to recurring software design problems [16]. They capture experts' design knowledge and experiences. With design patterns, junior developers can learn from their seniors' best practices rather than reinventing the wheel. Although they provide many benefits, design patterns often exist in the form of documents and fail to address one important application development aspect: code reuse. The CO_2P_3S (Correct Object-Oriented Pattern-based Parallel Programming System) system overcomes this problem by using design pattern templates, a generative pattern form that generates correct object-oriented framework code for design patterns [19]. Each template consists of a set of options for adapting the generated code to the specific application context of the corresponding design pattern. To implement a design pattern, a programmer only needs to select a pattern template, customize it by setting template options, and provide a few application-specific hook methods.

Prior to the research reported in this paper, CO_2P_3S had been used to generate simple parallel/distributed applications from basic concurrent design patterns such as mesh, wavefront, pipeline, search-tree, etc. It had also been used to generate simple sequential applications from basic sequential design patterns such as decorator, observer, composite, etc. Experiments showed that CO_2P_3S can greatly ease the complexity of parallel application development, and that the generated parallel applications have comparable performance to their handcrafted counterparts [3]. Although these are all small applications, the success of CO_2P_3S has motivated current research efforts to apply it to the generation of larger and more complex applications. In this paper we describe a design pattern template called the Network Server (N-Server) that was created for the construction of network server applications.

The N-Server employs an event-driven concurrency model and uses the Reactor [22] pattern as the fundamental mechanism for event demultiplexing and dispatching. I/O operations must be non-blocking for event-driven concurrency. To meet this requirement, Java NIO [12] is employed for non-blocking socket I/O, and non-blocking file I/O operations are emulated using a pool of threads (because they are not yet available in the Java API). Many design pattern options have been created, some of which exist purely to address performance issues, while others deal with features demanded by many network server applications. These options include event logging, performance profiling, event scheduling, file caching, and overload control. The options make the N-Server highly customizable and suitable for the construction of a large variety of network server applications with dramatically different functionality and performance requirements, ranging from trivial applications (e.g., Time server) to those as sophisticated and performance-sensitive as Web servers.

To evaluate the N-Server, a Web server (COPS-HTTP) and an FTP server (COPS-FTP) were constructed. Only a small amount of code needs to be manually programmed for each server, while the rest of the code is either generated from the N-Server or reused from existing application libraries. More importantly, the generated code contains all of the complex concurrency control code and the manually written code is simpler server-specific sequential code. An experiment was conducted to compare the performance of COPS-HTTP against Apache, a widely used Web server, and we found that comparable levels of performance were achieved. Two

[x] Z. Guo, J. Schaeffer, D. Szafron and P. Earl are with the Department of Computing Science, University of Alberta, Canada. Email {zhuang, jonathan, duane}@cs.ualberta.ca. This work was funded by grants from the Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta's Informatics Circle of Research Excellence (iCORE).

additional experiments were conducted to demonstrate the effectiveness and the ease-of-use of the event scheduling and automatic overload control features of the N-Server.

The major contributions of this research include: (1) the introduction of the N-Server pattern template to support construction of network server applications, (2) evidence that the generative pattern template approach can be used to construct larger and more complex applications, and (3) a configurable Web server with a clean framework-based architecture that enables the rapid deployment of significantly different network server applications, with performance that scales well with multiple processors and is comparable to that of Apache. Section 2 of this paper describes relevant background issues. Section 3 gives a brief overview of related work. Section 4 describes the design of the N-Server. Section 5 presents the results of several performance experiments. Finally, Section 6 outlines our conclusions and future work.

II. Background

A network server application needs a concurrency strategy to handle multiple requests simultaneously and to scale well when run using multiple processors. Based on how concurrency is achieved, concurrency models used by server applications can be grouped into two broad categories: multiprogramming and event-driven. Multi-programming concurrency models use multiple operating system processes/threads to achieve concurrency. When the execution of one process/thread is paused when performing a blocking operation, the CPU can switch to another one, so that client requests are served simultaneously. Although multiprogramming concurrency models are straightforward to implement, they do not scale very well. When the number of processes/threads is large, the overheads associated with multiprogramming—including context switching and scheduling, cache misses, and lock contention—can cause serious performance degradation [28], [13]. In addition, it is hard to support different quality of service levels using multiprogramming concurrency models, because they have a limited capacity for resource management and request scheduling [27], [2]. This is essentially caused by the focus of most OS schedulers on supporting the concurrent execution of multiple time-sliced processes/threads. There are no scheduling mechanisms for the usage of other important system resources, such as disk I/O and network bandwidth.

To overcome the limitations of the multiprogramming concurrency models, event-driven models have become more popular. In these models, application behaviors are triggered by internal or external events. A small number of processes/threads (typically one per CPU) loop continuously to process events—this avoids the process/thread switching/scheduling overheads. On the other hand, since event-driven applications can manipulate events directly, they can more easily alter the order of event processing, and hence, provide better support for multiple quality of service levels. Event-driven concurrency models have many advantages over multiprogramming concurrency models and have become more pervasive. However, several challenges exist for their design and implementation. It is hard to implement an event-driven concurrency model correctly. The lack of OS support for non-blocking I/O mechanisms negates the performance advantage of event-driven concurrency models. Therefore, a poorly designed event-driven model can fail to achieve good performance. In general, we believe that employing event-driven concurrency models in a server design is an effective approach, if complexity and correctness issues can be solved.

Unlike other design pattern templates in CO_2P_3S that are based on a single design pattern, the N-Server pattern synthesizes the ideas of four concurrent and networked design patterns: Reactor [22], Proactor [10], Acceptor-Connector [21], and Asynchronous Completion Tokens [11]. Both the Reactor and Proactor address the problem of how an event-driven application demultiplexes and dispatches events. In the Reactor, different kinds of Event Handlers encapsulate application-specific logic for processing different kinds of events. Each Event Handler has a different implementation of a common hook method. Each Event Handler registers with the Event Dispatcher. The Event Dispatcher repeatedly polls for ready events and dispatches a registered Event Handler to process each one. Unlike the Reactor, which waits passively for events to occur and react, the Proactor associates a Completion Handler with an asynchronous operation. The Completion Handler encapsulates application-specific functionalities to handle a client request. Upon completion of the asynchronous operation, the associated Completion Handler is dispatched to process the result. The Acceptor-Connector helps to reduce the development complexity by separating two independent aspects of network communication applications: data communication and connection establishment. An event-driven application often issues asynchronous operations to acquire one or more services (e.g. disk I/O), and a service indicates its completion by sending back a response. The Asynchronous Completion Token provides an effective mechanism to associate service responses with actions to be performed. In the N-Server, the Acceptor-Connector is applied to automate connection establishment, and the ideas of the Proactor and Asynchronous Completion Tokens are used to emulate non-blocking operations.

III. Related Work

Many attempts have been made to improve the quality and the performance of network server applications. The Zeus Web server [30] and the Harvest Web cache [7] employ a single-process event-driven (SPED) architecture, which uses only a single process to handle multiple requests. This process performs asynchronous I/O operations and uses the UNIX *select* or System V *poll* to check for I/O operation completion. Pai, Druschel, and Zwaenepoel proposed the multi-process event-driven architecture (MPED) that enhances the SPED by using multiple *helper* processes to handle blocking I/O operations [20]. Both of these two architectures can be emulated using the N-Server. Hu and Schmidt constructed a high performance Web server using a collection of concurrent and networked design patterns as the basic building block [14]. In their work, design patterns are represented as an API in the form of a class library. However, to build a server, programmers still need to write code to assemble different design patterns together. In contrast, the N-Server requires less development effort since it generates a code framework that handles all of the complex interactions and the programmer need only write simple sequential "call-back" or "hook" methods. The advantages of using a framework instead of a function library are well documented in the software engineering literature [9]. Welsh, Culler, and Brewer proposed the staged-event-driven-architecture (SEDA) and implemented it using an object-oriented framework [28]. In SEDA, an application is modeled as a finite state machine and each FSM stage is embodied as a self-contained component, which consists of an event handler, an incoming event queue, and a pool of threads. Each thread in a stage concurrently performs the operation of pulling events off the incoming event queue and processing them using the event handler. It is fairly easy to program with SEDA, since only a few event handlers that encapsulate the application-specific functionalities need to be provided. SEDA's staged design has a modeling advantage. However, this design suffers from additional thread switching/scheduling overheads, which negate the performance advantage of event-driven concurrency models. This happens when there are more stages used than available processors, so that threads belonging to different stages contend for processors. Unfortunately, it is very likely for an application to be modeled using more stages than processors. Compared to SEDA, the N-Sever is more powerful. It provides more features that are desirable to many network server applications, including file caching and event scheduling. The generative design pattern approach is more configurable than a static framework, since application code underlying each feature can be included or excluded at code generation time, based on the corresponding option settings [6]. To achieve the same effect in a static framework, a large amount of indirection code would be needed to dynamically decide whether to execute the code for each feature (e.g., executing *if* or *case* statements to check which features are enabled, as opposed to using conditional compilation flags). Dynamic checks reduce application maintainability and add performance overheads. The benefits of implementing the N-Server as a template become more significant for features that crosscut [17] multiple application components. This occurs when a single feature requires checks throughout the code-base. Logging and event scheduling are examples of such crosscutting features.

Table 1 shows all of the options for the N-Server design pattern in the first column and the legal values for these options in the second column. The next two columns show the values for these options in the two N-Server applications described in the next section (COPS-FTP and COPS-HTTP). Table 2 shows each of the main classes that implement the N-Server design pattern as a row and each option supported by the pattern as a column. The entries in Table 2 indicate which classes have code that is affected by each option. Table 2 serves as a graphic illustration that a static framework that supports all the options is infeasible and supports our decision to generate a custom framework after option selection.

TABLE I
N-SERVER OPTIONS AND THEIR VALUES

Option Name	Legal Values	COPS-FTP	COPS-HTTP
O1: # of dispatcher threads	1 or 2..N	1	1
O2: Separate thread pool for event handling	Yes/No	Yes	Yes
O3: Encoding/Decoding required	Yes/No	Yes	Yes
O4: Completion events	Asynchronous /Synchronous	Synchronous	Asynchronous
O5: Event thread allocation	Dynamic/Static	Dynamic	Static
O6: File cache	Yes[1]/No	No	Yes: LRU
O7: Shutdown long idle	Yes/No	Yes	No
O8: Event scheduling	Yes/No	No	No, Yes[2], No
O9: Overload control	Yes/No	No	No, No, Yes[3]
O10: Mode	Production/ Debug	Production[4]	Production[4]
O11: Performance profiling	Yes/No	No[5]	No[5]
O12: Logging	Yes/No	No	No

O6[1]: cache replacement policies: LRU, LFU, LRU-MIN, LRU-Threshold, Hyper-G or Custom. O8[2]: Event scheduling was turned on only for the second of the three HTTP sever experiments. O9[3]: Overload control was turned on only for the third of the three HTTP sever experiments. O10[4]: Debugging was used during development. O11[5]: Profiling was used during tuning.

IV. Network Server Pattern Template

The N-Server is designed to significantly ease the complexity of network server application development, while satisfying a diverse range of performance and functionality requirements. It employs an event-driven concurrency model to achieve high performance, provides good support for resource management, and offers various levels of service quality. The Reactor pattern is used as the fundamental mechanism for event demultiplexing and dispatching. Thus, the basic structure of the N-Server is based on the Reactor pattern. However, the N-Server is not equivalent to the Reactor pattern. In one way, it specializes the Reactor pattern by limiting it to a network communication server. In another way, it extends the Reactor pattern by providing support for multiple event sources and multiple processors. In an event-driven network server application, events may arise from multiple sources, such as I/O ports, timers, or other application components. Different event sources have different characteristics, and therefore, they should be managed separately. Because it's not possible to anticipate and include all the event sources, there should be an effective mechanism for new event sources to be added. In view of these problems, an Event Source component that complies with the Decorator [16] pattern is added. Besides managing multiple event sources, it is also responsible for registering and deregistering Event Handlers and polling ready events.

One drawback of the Reactor pattern is that it does not scale up very well, because all events are processed by one thread contained in the Event Dispatcher. This prevents it from making use of multiple processors when they are available. A new participant, called the Event Processor, is added to solve this problem. An Event Processor contains an event queue and a pool of threads that operate collaboratively to process ready events. In this case, the Event Dispatcher is only responsible for querying the Event Source for ready events and then passing those ready events to the Event Processor for processing. The N-server pattern has an option to include an Event Processor or use a standard Reactor. Event-driven concurrency models require events to be non-blocking. However, non-blocking OS mechanisms are difficult to use so they are often not fully exploited for many types of events, such as File I/O, database access, and synchronization. In particular, there is no non-blocking File I/O in Java JDK1.3, although non-blocking socket I/O is supported in Java JDK1.4 and higher. Another usage of the Event Processor is to emulate the existence of non-blocking events.

Even with the addition of these two concepts (a thread pool and non-blocking events), a network server application generated from such an extended Reactor pattern template would require programmers to explicitly write code to deal with network communication, which is a difficult programming task. In addition, a great deal of user code would need to be written for frequent registration and deregistration of different Event Handlers and the queuing of user-defined Events. This situation arises due to the inadequacy of the Reactor pattern to exploit the similarities between network server applications. Other abstractions are necessary to reduce the manual coding effort.

Most network server applications are similar in the way they establish network communication with peers and in the way they handle requests. To establish network communication, they create a server socket listening to a certain network port for new connections. After a connection arrives, they accept it and iteratively carry out a five-step process to handle requests. These five steps are: Read Request, Decode Request, Handle Request, Encode Reply, and Send Reply (Fig. 1). The Read Request step reads the raw data of a request sent from the remote peer over the socket connection. The Decode Request step parses the request. The Handle Request step provides services by handling the request. The Encode Reply step encodes the result of the request in a form understood by the remote peer. The Send Reply step sends out the raw data of the result to the remote peer. Among the five steps, the Read Request and Send Reply are almost the same across different network server applications, while the other steps are application-dependent.

Therefore, to develop a network server application using the N-Server pattern, a programmer only has to write code corresponding to the three application-dependent steps, while the N-Server generates code for the other two common steps. This reduces coding effort and improves the chances for the correctness of the network server application. In addition, for simple server applications, there is no need to have an explicit Decode and Encode Request step. When using the N-Server pattern, a programmer has the freedom of generating this structural variation (Fig. 2) by setting a template option.

There exists a trade-off between generality and efficiency in the N-Server. Without the inclusion of the network server application specific code (Read Request, Send Reply, and establishing a connection), the N-Server would be a template that instantiates the Reactor design pattern. Thus, it would be more generic and could be used for many types of applications, such as event-driven simulations and graphical user interface frameworks. With the inclusion of the network specific code, the N-Server becomes more specialized and more efficient for the automatic generation of network server applications. The N-Server sacrifices generality in favor of improving efficiency. This choice is inevitable, because it is mandated by the design goal of the N-Server.

TABLE 2
AN ILLUSTRATION THAT THE N-SERVER OPTIONS CROSSCUT THE CODE. FOR OPTION NAMES, SEE TABLE 1. AN O INDICATES THE OPTION DETERMINES WHETHER THE CLASS EXISTS IN THE GENERATED FRAMEWORK. A + INDICATES THAT THE CODE GENERATED FOR THE CLASS DEPENDS ON THE OPTION VALUE.

Class / Option	O1	O2	O3	O4	O5	O6	O7	O8	O9	O10	O11	O12
Event				+				+				
Completion Event				O								
File Open Event				O		+						
File Read Event				O		+						
Handle	+											
File Handle				O		+						
Read Request Event Handler							+			+	+	+
Send Reply Event Handler							+			+	+	+
Decode Request Event Handler			O				+	+		+		+
Encode Reply Event Handler			O				+	+		+		+
Compute Request Event Handler			+	+			+	+		+		+
Event Processor					+			+	+	+		
Processor Controller					O							
Event Dispatcher		+		+					+	+	+	
Cache						O					+	
Reactor	+	+		+	+	+		+	+	+	+	+
Communicator Component			+				+	+			+	
Server Component			+				+			+		+
Client Component			+				+			+		+
Server Event Handler							+			+	+	
Connector Event Handler			+							+	+	+
Acceptor Event Handler			+						+	+	+	+
Container Component							+			+	+	+
Application Event Handler							+			+	+	
Client Configuration			+							+		
Server Configuration										+		
Server			+									

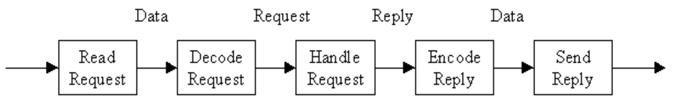

Fig. 1. The five basic steps for server request handling.

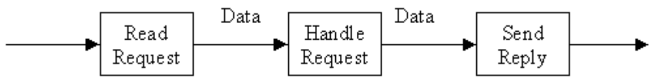

Fig. 2. Server request handling when no encoding/decoding is required.

Other functionalities that are often needed by many network server applications are also provided by the N-Server, including file caching, event scheduling, and automatic overload control. These are selected by setting pattern options.

Many network server applications need to access disk files to provide services. However, disk I/O is slower by several orders of magnitude than memory access. To boost application performance, network servers often implement a file cache that keeps the disk file in memory for fast access. To relieve users from the burden of implementing a file cache, the N-Server can be configured to generate code that automatically caches disk files in memory. The caching capability provided by the N-Server is transparent to the programmer. This means that programmers have no extra development effort, unless specialized cache replacement policies are required. Five cache replacement policies are provided by the N-Server: LRU, LFU, LRU-MIN [1], LRU-Threshold [1], and Hyper-G [29]. When necessary, a programmer can implement a different cache replacement policy by simply adding code to a hook method that is called automatically at the appropriate time. Whether to use file caching, and which cache replacement policy to use, are controlled by one template option.

Many network server applications can benefit from the capability of altering the order in which requests from different concurrent connections are serviced. At least two kinds of benefits can be gained by this capability. They are: (1) improving the overall quality of service for all and (2) providing differentiated levels of services. In an event-driven network server application, such a capability requires the support of Event Scheduling. The N-Server can be configured to support an efficient Event Scheduling mechanism to implement whatever event scheduling policy an application needs. In this mechanism, events of higher priority are processed first. However, each priority level is given a quota. When the quota is exhausted, events of lower priority are processed, so that starvation is avoided. When event scheduling is enabled, structural variations are introduced that crosscut several components in the generated code. These include the addition of a priority field and access methods in the Event class and Communicator Component class (whose instances represents network connections), and replacing a normal event queue in an Event Processor by a priority queue. This example illustrates a strength of the generative pattern template approach of CO_2P_3S. Application code is only generated when necessary, without adding extraneous code. Similar crosscutting changes occur for other template options.

Overload in a network server application can cause increased response times, decreased throughput, and even service rejection. Event-driven concurrent server applications are extremely vulnerable to overload conditions, since they usually do not limit the number of connections accepted. This is not the case in server applications based on a multiprogramming concurrency model [25], where connection limits are common. The N-Server provides two mechanisms to control overload. The first one is trivial: that is to limit the maximal number of simultaneous connections in the server. In the second approach, the N-Server is configured to generate code that queries the length of multiple queues. Each queue stores events of certain types. If there is a queue whose length exceeds its specified high watermark, then new connection requests are postponed until the length drops below a specified low watermark. The second mechanism is effective in handling overload situations that can be caused by multiple bottlenecks, such as CPU and disk [26].

There is no single “best way” to build a network server. The N-Server must be configured based on the type of traffic expected, quality of service, fault tolerance, etc. Other supported features include performance profiling, debugging, logging, and termination of long-idle connections. Important statistical information of the server application can be automatically gathered, if the N-Server is configured to enable performance profiling. This information includes: the number of connections accepted, the number of bytes read, the number of bytes sent, the file cache hit rate, etc. The N-Server can generate network server applications in two modes: debug and production. If the server is generated in debug mode, then all internal events that are triggered in the server are written into a file. The user can trace this file to get a snapshot of what happened during the time an error condition occurred. Such a debug mechanism is far from being perfect, but it is useful. The N-Server can also be configured to generate applications with a logging capability. Long-idle connections may consume unnecessary resources and degrade the performance of network server applications. The N-Server generates code that is able to automatically terminate these connections.

V. Evaluation

Two moderate-sized network server applications were developed to evaluate the N-Server: an FTP server (COPS-FTP) and a high-performance Web server (COPS-HTTP). The N-Server has wider use beyond the two applications described in this paper. For example, the pattern can be used to generate a mail server, time server, or any other network-based server.

A. COPS-FTP

COPS-FTP was developed to 1) demonstrate the flexibility and expressiveness of the N-Server to generate network server applications with complex architectures and 2) to show how the N-Server can make extensive use of existing code by adapting it to a new server architecture. The nature of FTP has made an FTP server relatively harder to implement using the N-Server than a Web server. In FTP, a client establishes a control connection to a server. This control connection carries commands that tell the server which service to provide.

However, unlike HTTP, FTP uses a separate data transfer connection for file transfer [8]. A server can establish this data transfer connection either passively or actively. For example, COPS-FTP can passively open a data transfer connection to store a file transferred from a client. COPS-FTP is a full-featured FTP server. Rather than building it from scratch, we modified the Apache FTPServer, a Java-based multithreaded FTP server that is part of the Apache Avalon project [4].

Table 3 summarizes the code distribution of COPS-FTP. A large fraction of code, 8,141 lines of non-comment source code (NCSS), is reused from Apache FTPServer. This code supports many functions, including a GUI, a database for LDAP access, and user activity monitoring. A total of 1,897 lines of NCSS are added, replacing 1,186 lines. Only 711 lines of extra code have to be programmed, to transform the Apache FTPServer to an event-driven FTP server, even though an event-based server has a much more complex architecture.

B. *COPS-HTTP*

COPS-HTTP demonstrates that the N-Server is capable of constructing high performance applications. COPS-HTTP is not yet a full-featured Web server; it only handles static Web page requests. The same pattern can be used to generate a server for dynamic content, except that more application-dependent code would be required to support the additional protocols.

The N-Server uses an Event Dispatcher to dispatch ready Reactive Events to an Event Processor for processing. Another Event Processor is used to emulate non-blocking disk I/O. COPS-HTTP is generated with the caching capability and enforced LRU cache replacement policy specified.

The code base of COPS-HTTP is composed of two parts: automatically generated code and handcrafted code. The handcrafted code can be further divided into code that constitutes an HTTP protocol library and code that implements the HTTP server-specific logic. Table 4 shows the code distribution. In total, there are 3,931 lines of non-comment source statements (NCSS) in COPS-HTTP, of which 2,697 lines of NCSS are automatically generated by the N-Server. If an existing HTTP protocol library were used for the development of COPS-HTTP, only 785 lines of NCSS would need to be programmed, which accounts for 20% of the total code of COPS-HTTP.

An experiment was conducted to measure COPS-HTTP performance against Apache (version 1.3.27) under workloads of the form “get me a file”. Apache is the most widely used Web server on the Internet. A survey shows that more than 60% of Web sites run Apache [5]. Apache implements the process-per-connection concurrency model and uses a bounded worker process pool of 150 processes to serve simultaneous client connections. COPS-HTTP is written in Java with most of the code generated using CO_2P_3S; Apache is handcrafted C.

The hardware environment for the experiment is two web servers connected to 16 clients. Both web servers run on a Sun Enterprise 420R server (4 450-MHz Sparc9 processors, 4 GB of RAM, SunOS 5.9). The clients are Sun Ultra 10 machines (440-MHz UltraSparc processor, 256 MB of RAM, and SunOS 5.8). A switched Gigabit Ethernet connects the clients and servers. The maximal packet size of the Ethernet Switch is 1500 bytes. This complies with the SpecWeb99 benchmark rules; however, the actual network bandwidth is limited to something slightly higher than 100 MBits/sec. Although network bandwidth is clearly the bottleneck resource, this network configuration is quite close to the real-world situation that a high-performance Web server often faces.

The client workload generators repeatedly perform the following actions: establish a connection to the Web server, issue 5 HTTP requests (to simulate HTTP 1.1 persistent connections), and then terminate the connection. To simulate the wide-area transfer delay, there is a 20 milliseconds pause after receiving each page, before the next page is requested. The file size and access frequency distribution follows the SpecWeb99 benchmark [23]. A file set of size 204.8 MB is created using the SpecWeb99 suite, with an average file size of 16 KB. The file cache of COPS-HTTP is limited to 20 MB, and the file system has a memory buffer of size 80 MB. Both Web servers were warmed up before the experiment. The number of Web clients simulated in the experiment is up to 1024. Each measurement ran for 5 minutes.

TABLE 3
THE CODE DISTRIBUTION OF COPS-FTP. THE NCSS COLUMN CONTAINS THE NUMBER OF LINES OF CODE THAT WERE NOT COMMENT STATEMENTS.

	Classes	Methods	NCSS
Reused code	124	945	8,141
Removed code	18	199	1,186
Added code	23	150	1,897
Generated code	84	480	2,937

TABLE 4
THE CODE DISTRIBUTION OF COPS-HTTP. THE NCSS COLUMN CONTAINS THE NUMBER OF LINES OF CODE THAT WERE NOT COMMENT STATEMENTS.

	Classes	Methods	NCSS
Generated code	79	474	2,697
HTTP protocol code	10	50	449
Other application code	16	89	785
Total code	105	613	3,931

Fig. 3 shows the throughput of COPS-HTTP and Apache (note the logarithmic horizontal scale). Apache achieves slightly better throughput than COPS-HTTP under light workloads (less than 32 Web clients). However, under heavier workloads (32 clients to 256), the throughput of COPS-HTTP is higher than that of Apache. This result confirms that the advantage of event-driven concurrency models lies in its superior performance and scalability to handle a large number of simultaneous requests. With more than 256 Web clients, both applications get saturated because the network becomes the performance bottleneck. The throughput of Apache is better than COPS-HTTP at 1024 clients, but Apache exhibits extreme service unfairness.

Fairness is an important performance metric that is concerned with the equal allocation of resources. Fig. 4 shows the service fairness metric that is based on the Jain fairness index [15] of the number of responses received by each Web client. This metric is computed from:

$$f(x)=\left(\sum x_i\right)^2 / N\sum x_i^2 \,,$$

where x_i is the number of responses received by Web client i, and N is the total number of Web clients. If all the Web clients receive an equal number of responses, the fairness index is 1. If k Web clients receive equal number of responses, and the other Web clients receive no responses at all, the fairness index is k/N. Under heavy loads, the fairness index of COPS-HTTP remains high, while Apache's fairness index drops significantly. With 1024 Web clients, the fairness index of Apache is a mere 0.51. The extreme unfairness of Apache is caused by the exponential backoff scheme of the TCP protocol. Apache only handles 150 simultaneous connections at any time. For a lucky Web client, its connection is accepted, and a single process handles all its requests quickly. Unfortunately, some Web clients are very unlucky, in that their TCP SYN packets for establishing connections are dropped by Apache. In this case, they may wait for a significant amount of time before doing a retransmit. The maximal retransmission timeout under Solaris is 1 minute. Therefore, Apache achieves higher throughput than COPS-HTTP under very heavy workloads (1024 clients), at the expense of fairness.

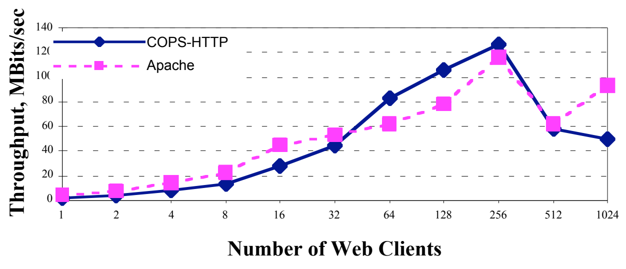

Fig. 3. Throughput for the COPS-HTTP/Apache Web server experiment.

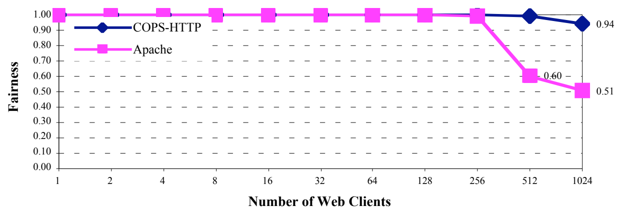

Fig. 4. Service fairness for the COPS-HTTP/Apache web server experiment.

The second experiment demonstrates the utility of the N-server's event-scheduling mechanism to support multiple levels of service on the same Web server. This facility is absent from Apache, but can be activated in the N-server by selecting a single pattern option. As a simple example of this feature, we consider a scenario where an Internet Service Provider (ISP) hosts two types of Web content: a corporate portal and personal homepages. Since the corporation pays a higher service charge, Web accesses to the corporate portal are prioritized by allocating more system resources (for example, network bandwidth, disks, processors, memory, etc). In this experiment, two 933 MHz Pentium III systems with 256 MB of RAM and Linux 2.4.9 are used as client machines to generate workloads. A dual-processor 600 MHz Pentium III machine with 512 MB of RAM and Linux 2.4.9, interconnected with the two client machines over a 100 MBits/sec Ethernet, is used to host COPS-HTTP. The IP address is used to determine whether a request sent from each client machine is considered as an access to the corporate portal or as an access a to personal homepage. Only 13 lines of code are added to COPS-HTTP to implement this scheduling policy. In addition, the file caching capability is disabled to make the workload heavier.

Fig. 5 shows the throughput of requests for the two types of content under various priority level settings. A priority level setting is specified as a ratio x/y, where x gives the quota allocated for homepages and y for the corporate portal. The rightmost column shows the maximal throughput for corporate portal requests, when no homepage request is generated. There is a small gap between the ratio of priority levels and the actual throughput ratio of requests for the two types of Web contents. However, such a gap is quite acceptable, because the COPS-HTTP variant exerts no control over the management and scheduling of many operating system resources, such as the order in which the network socket buffers are drained.

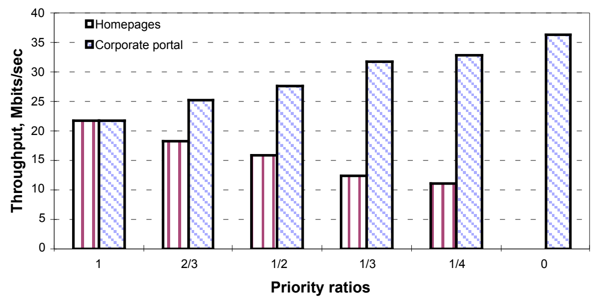

Fig. 5. Service throughput for differentiated service levels supported by COPS-HTTP.

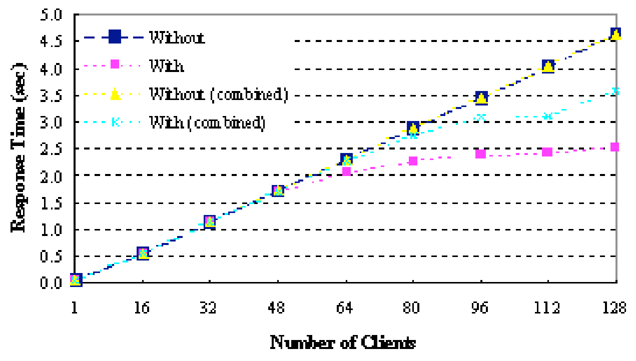

Fig. 6. Response time with and without automatic overload control, for the COPS-HTTP server. Combined times include connection establishment times.

The third experiment demonstrates the automatic overload control mechanism of the N-Server. In this experiment, CPUs are considered as the bottleneck resource. To make the workload more CPU-intensive, each thread is forced to sleep for 50 milliseconds when decoding an HTTP request. The high watermark and low watermark for the Reactive Event

Processor queue length are set to 20 and 5 respectively. The number of Web clients in this experiment varies from 1 to 128. Fig. 6 shows that COPS-HTTP with the automatic overload control capability has a significantly lower average response time. Notably, this is achieved without degrading the server throughput. The combined response time also counts the time a Web client waits to establish a connection when calculating the response time. The response time alone better describes what Web clients with established connections experience when a Web server gets overloaded. When overloaded, a server should concentrate on preventing the QoS of established connections from dropping rather than taking on more tasks. In this sense, the response time is more meaningful than the combined response time. However, the average combined response time is also presented, because it describes the experiences of all the Web clients (including those with and without established connections) as a whole.

VI. Conclusion

In this research, we applied the generative design patterns of the CO_2P_3S system to the construction of larger and more realistic applications that belong to a difficult problem domain (network server applications), and demonstrated the effectiveness of this approach. CO_2P_3S enables the rapid deployment of custom network server applications with significantly smaller programmer effort than required by manual creation of server applications. The hard parts of the application—such as the concurrency control—are automatically generated, leaving the programmer to only supply the sequential application-specific code.

The most interesting extension of this work is to support the generation of distributed N-servers that will serve from a network of workstations. As was the case with previous CO_2P_3S patterns, the distributed version of this pattern would require the programmer to write identical hook methods, for an application whether the application was generated for a shared memory machine or a network of workstations [24].

References

[1] M. Abrams, C. Standridge, G. Abdulla, S. Williams, and E. Fox. Caching Proxies: Limitation and Potentials. Virginia Polytechnic Institute technical report TR-95-12, 1995.

[2] J. Almedia, M. Dabu, A. Manikntty, and P. Cao. Providing Differentiated Levels of Service in Web Content Hosting. University of Wisconsin technical report CS-TR-1998-1364, 1998.

[3] J. Anvik. Asserting the Utility of CO_2P_3S Using the Cowichan Problems. Master's Thesis, Department of Computing Science, University of Alberta, Fall 2002.

[4] Apache Avalon Project. http://avalon.apache.org.

[5] August 2003 Web Server Survey. http://www.netcraft.com/survey.

[6] S. Bromling. Meta-programming with Parallel Design Patterns. M.Sc. thesis, Department of Computing Science, University of Alberta, 2002.

[7] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A Hierarchical Internet Object Cache. In USENIX Annual Technical Conference, 1996, pp. 153-163.

[8] D. Comer and D. Stevens. TCP/IP Volume I: Client-Server Programming and Application. Prentice Hall, 1997, pp. 419-426.

[9] M. Fayad, D. Schmidt, and R. Johnson, editors. Building Application Frameworks, John Wiley & Sons, 1999.

[10] T. Harrison, I. Pyrarli, D. Schmidt, and T. Jordan. Proactor – An Object Behavioral Pattern for Dispatching Asynchronous Event Handlers. Washington University Technical Report #WUCS-97-34, 1997.

[11] T. Harrison and D. Schmidt. Asynchronous Completion Tokens: An Object Behavioral Pattern for Efficient Asynchronous Event Handling. In Pattern Languages of Program Design, Addison-Wesley, 1997.

[12] R. Hitchens. Java^{TM} NIO. O'Reilly, 2002.

[13] J. Hu, I. Pyarali, and D. Schmidt. High Performance Web Servers on Windows NT: Design and Performance. USENIX Windows NT Workshop, USENIX, 1997, pp. 149-149.

[14] J. Hu and D. Schmidt. JAWS: A Framework for High Performance Web Servers. In Domain-Specific Application Frameworks: Frameworks Experience by Industry, Wiley & Sons, 1999, pp. 339-376.

[15] R. Jain, D. Chiu, and W. Hawe. A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer Systems. Technical Report TR-301, DEC Research, Sep. 1994.

[16] R. Johnson, E. Gamma, R. Helm, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994, pp. 1-2, 175-184.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, J. Irwin. Aspect-Oriented Programming: Object-Oriented Technology: ECOOP'97 Workshop Reader, Vol. 1241, Springer, 1997, pp. 220-242.

[18] J. R. Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance. SIGPLAN Notices, Vol. 36, 8, ACM Press, 2001, pp. 182-187.

[19] S. MacDonald, D. Szafron, J. Schaeffer, J. Anvik, S. Bromling, K. Tan. Generative Design Patterns. Automated Software Engineering (ASE), 2002, pp. 23-34.

[20] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In USENIX Annual Technical Conference (USENIX-99), 1999, pp. 199-212.

[21] D.C. Schmidt. Acceptor-Connector: An Object Creational Pattern for Connecting and Initializing Communication Services. In Pattern Languages of Program Design, Addison-Wesley, 1997.

[22] D.C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In Pattern Languages of Program Design, Addison-Wesley, 1995, pp. 529-545.

[23] The Standard Performance Evaluation Corporation. SpecWeb96/SpecWeb99. http://www.spec.org/osg.

[24] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, and S. MacDonald, Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment, Principles and Practice of Parallel Programming (PPoPP), June 2003, pp. 203-215.

[25] T. Voigt. Overload Behavior and Protection of Event-Driven Web Servers. Springer-Verlag LNCS, Vol. 2376, 2002, pp. 147-157.

[26] T. Voigt and P. Gunningburg. Handling Multiple Bottlenecks in Web Servers Using Adaptive Inbound Controls. Springer-Verlag LNCS, Vol. 2334, 2002, pp. 50-68.

[27] M. Welsh and D. Culler. Virtualization Considered Harmful: OS Design Directions for Well-Conditioned Services. Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII), Schloss Elmau, Germany, IEEE Computer Society Press, 2001, pp. 139-146.

[28] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. SIGOPS Operating Systems Review, Vol. 35, No. 5, ACM Press, October 2001, pp. 230-243.

[29] S. Williams, M. Abrams, C. Standridge, G. Abdulla, and E. Fox. Removal Policies in Network Caches for World Wide Web Documents. SIGCOMM Computer Communication Review, Vol. 26,4, ACM Press, 1996, pp. 293-305.

[30] Zeus Inc. The Zeus WWW Server. http://www.zeus.co.uk.