

THE UNIVERSITY OF CALGARY

Automatic Generation of Network Servers

by

Kelly Wilson

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

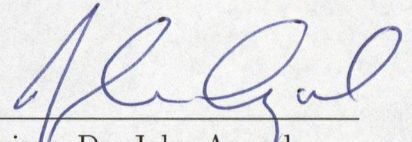
CALGARY, ALBERTA

September, 2005

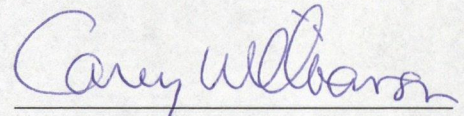
© Kelly Wilson 2005

THE UNIVERSITY OF CALGARY
FACULTY OF GRADUATE STUDIES

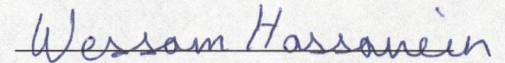
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Automatic Generation of Network Servers" submitted by Kelly Wilson in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE.



Supervisor, Dr. John Aycock
Department of Computer Science



Dr. Carey Williamson
Department of Computer Science



Dr. Wessam Hassanein
Department of Electrical and
Computer Engineering

15 Sept 2005

Date

Abstract

NEST, NETwork Server Tool, is a tool for automatically generating code infrastructure for TCP-based network servers. It uses a specification language to describe client-server interaction and state transitions within the server. This language has some similarities to the compiler tools Lex and Yacc, and its design makes network server specification straightforward. One of the main features of NEST is that it can generate three different types of server from the same basic specification: process-based, threaded, and event-driven. In addition, use of a server generation tool can improve programming productivity, abstract away unnecessary details, and help eliminate certain classes of error. We have tested the performance of our NEST generated servers against many other open-source servers, with favorable results for all three server models.

Acknowledgments

I would like to thank my supervisor, Dr. John Aycock, for his knowledge, insights and patience. A special thanks to my wife Joanna and daughter Kylie for supporting me in this endeavor. My twin brother, Kevin, for helping me where he could. My parents and brothers, family members, as well as my friends, who explained that there are some things you just can't get from book learning. I would also like to thank the members of the Programming Languages Lab at the University of Calgary for answering my questions and motivating me throughout this process.

Table of Contents

Approval Page	ii
Abstract	iii
Acknowledgments	iv
Table of Contents	v
1 Introduction	1
1.1 Client-Server	3
1.2 Compilers and Tools	10
1.3 Motivation	11
2 Related Work	15
2.1 Languages	15
2.1.1 MSPL	15
2.1.2 MAWL	17
2.1.3 Morpheus	18
2.2 Libraries	19
2.2.1 Serveez	19
2.2.2 BEA Tuxedo	21
2.2.3 Twisted	21
2.2.4 SEDA	22
2.2.5 ACE	23
2.2.6 Capriccio	24
2.3 Protocol Specification Languages	25
2.3.1 Promela++	26
2.3.2 HIPPCO	26
2.4 Miscellaneous	27
2.4.1 FistGen	27
2.4.2 RPCGen	28
2.5 Summary	29
3 NEST and its Specification Language	30
3.1 Introduction	30
3.2 Specification	30
3.3 The Specification Language In Depth	32

3.3.1	Declaration Section	33
3.3.2	Rules Section	36
3.3.3	C code	42
3.4	Some Slightly Larger Examples	42
3.4.1	Simple HTTP Protocol	42
3.4.2	Simple SMTP Protocol	46
3.5	Summary	47
4	Implementation Details	48
4.1	Process-based Server	50
4.2	Threaded Server	56
4.3	Event-driven Server	59
4.3.1	Non-blocking I/O	61
4.3.2	File-globals	64
4.4	Common code	69
4.5	Summary	70
5	Evaluation	71
5.1	Development Effort	71
5.2	Performance	72
5.2.1	httperf	74
5.2.2	Performance Graphs	74
5.3	Error Rates	80
5.4	Summary	82
6	Conclusions and Future Work	83
6.1	Conclusions	83
6.2	Future Work	84
A	The NEST grammar	85
	References	87

List of Tables

5.1	Process-based Server Error Rates	81
5.2	Event-driven Server Error Rates	81
5.3	Threaded Server Error Rates	82

List of Figures

1.1	OSI Model	4
1.2	Process-based Server	6
1.3	Threaded Server	8
1.4	Event-Driven Server	9
3.1	Server Generation	31
3.2	Stand-alone vs. Inetd Server	32
3.3	Simple NEST specification	34
3.4	Sample of simple authentication and HTTP-like commands	43
3.5	Regular Expression Expansion	44
3.6	Sample code for an SMTP-like protocol	45
4.1	Protecting Memory Pages	68
5.1	Benchmark results for process-based servers	76
5.2	Benchmark results for event-driven servers	77
5.3	Benchmark results for threaded servers	79

Chapter 1

Introduction

The Internet has revolutionized the way that people communicate with each other in everyday life as well as dramatically increasing the interaction of computers and devices over a network.

When most people think of the Internet, they envision clicking their mouse to jump from one “web” page to another. Some people will also envision a less interactive experience, such as typing in an electronic mail (email) message to a friend and sending it over the Internet. When these people interact with any portion of the Internet, they may be unaware of the vast resources and complex technology that make these seemingly simple interactions possible in today’s digital world.

Sending email and transferring files between computers were two of the earliest uses of the Internet. Surfing the “web”, or the World Wide Web (WWW) as it is properly called, is a more recent phenomenon.

These days, anyone wishing to view web pages will typically create a connection from their computer, or host as it is called, to another host on the Internet using a program called a web browser. These web browsers translate the content they receive into a visually appealing format for the user. The user types in a URL or search criteria into a web browser and they are magically taken to some informative or trivial web page located “out there” somewhere. Netscape Navigator and Microsoft Internet Explorer are just two examples of popular web browsers.

When two hosts communicate over a network in the fashion described in the

previous paragraph, we describe their communication as a client-server interaction. One host acts like a “client”, requesting information from another host, which serves up the requested data. Above, the user’s web browser is the client. This client-server architecture is one of the mainstays of the Internet.

Throughout this thesis we will be concentrating on the automatic generation of network server programs, the “server” part of the client-server architecture. More precisely, we will concentrate on the generation of the programming language code that makes up a server program, or daemon as they are called. These daemons are mainly programs that run continuously, listening for an incoming connection from a client, and then interacting with that client to service any of its requests.

The proliferation of the client-server paradigm throughout the Internet has led to extensive research in this area over the past several years. While other architectures are also prevalent on the Internet, such as peer-to-peer, they were not a point of focus for us as they would expand this thesis beyond our research goals. Peer-to-peer networks do not even have a concrete definition as yet[1], which would make code generation for this type of architecture difficult.

While web servers are very popular on the Internet we will also expound upon other types of common servers throughout the rest of this chapter. In the following sub-section we will give a more detailed description of the client-server paradigm that our tool focuses on.

1.1 Client-Server

The simplest form of client-server model has one computer called a server¹, and one or more “client” computers connected to this server. Servers are usually high performance machines connected to the network via a high-bandwidth connection. The server machine is used to “serve” out some sort of data in response to each client machine that makes a request for the data. There are many ways to connect a client and server, as well as many different ways to forward data over a physical connection.

We have already mentioned that computers need one or more types of common language to communicate with each other effectively. These languages are called protocols. Protocols for communicating between clients and servers can be placed into two categories: connectionless or connection-oriented. Connectionless protocols, in general, send out a data request onto a network connection when it is ready. They do not set up an explicit connection with the target machine, nor do they wait for, or confirm, a response to their request. Some connectionless protocols include the User Datagram Protocol(UDP) and the Internet Protocol(IP). Connection-oriented protocols, on the other hand, require a logical connection to be established between two machines prior to data being transferred. Some connection-oriented protocols include Transmission Control Protocol(TCP) and Asynchronous Transfer Mode(ATM).

The connection-oriented protocols we have mentioned usually have each end of

¹ One potential cause of confusion in the client-server world is the use of the word “server” to describe the physical machine that serves out data, as well as using “server” to describe a program that runs on these machines to “service” client requests. We will continue to use both meanings for “server”; but context will clarify whether we are talking about the machine or the “server” daemon.

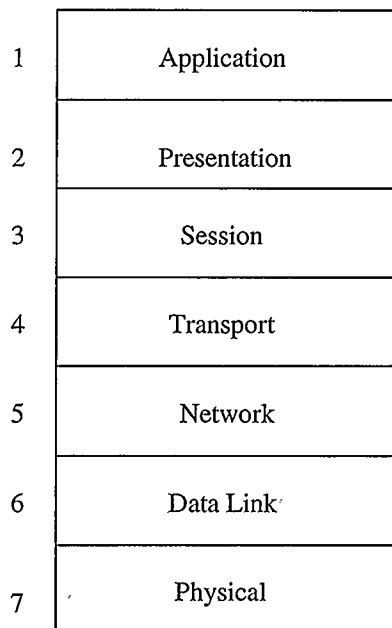


Figure 1.1: OSI Model

the connection maintaining state information about their communications, while connectionless protocols do not. One could think of a connectionless protocol as a pager; we send a message but we don't necessarily know if it was received by the person we intended. The connection-oriented relationship, in contrast, is analogous to a phone conversation with a certain person you want to talk to. With this "connection" one can confirm each statement prior to continuing the conversation, if desired.

We will be concentrating on the TCP/IP connection-oriented protocol in this thesis, and some higher, Application[42] level protocols that have been layered on top of TCP/IP. The OSI model in Figure 1.1 shows the different protocol levels. One of the higher level communications protocols is the HyperText Transfer Protocol (HTTP), which is used as a protocol for easily transferring small units of text and

graphics on the Internet. HTTP is the main protocol used by Web browsers to communicate with servers. Other Application level protocols include the File Transfer Protocol (FTP) for transferring files and the Simple Mail Transfer Protocol (SMTP) for sending mail.

Internet servers using FTP are referred to as FTP servers, while servers that communicate using SMTP are called mail servers, and HTTP servers are called Web servers. Many server types can run on the same hardware, but sometimes each type of server is hosted on a separate machine to minimize overloading. The server hardware and operating system may be optimized for certain types of transfers, as well. For instance, an FTP server tends to serve up large files so a very large cache may be used (or no cache at all). A cache is a list of in-memory files that may be queried by a server application so as to minimize slow disk accesses to send out a regularly-requested file. Web servers, on the other hand, may be optimized to serve out many small files quickly.

The implementation of many of these server programs operating at the Application level are varied and complex, but most can be placed into one of three general programming models. These are process-based (or forking), threaded and event-driven server models.

Forking A forking server is a server that uses the Unix “fork()” system call to spawn a new process for each incoming client request. The new process is a replica of the parent process. This process spawning can be seen in Figure 1.2 where one connection only spawns one process, but three connections force the server to spawn three separate, but identical, processes. There is usually

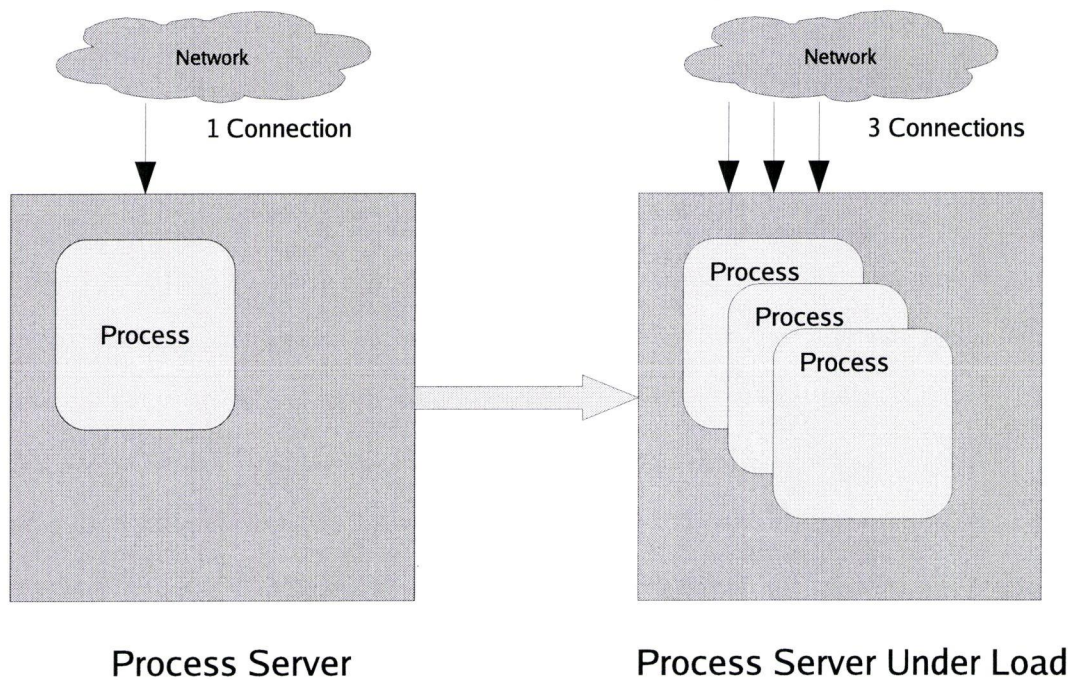


Figure 1.2: Process-based Server

some significant overhead involved in this process replication (depending on the operating system), and thus forking servers are generally considered to be slower than the other models. Pre-forking servers allocate some new replica processes upon startup, then use these replicas for each client. Re-use of these replicas helps to keep the process startup overhead down. We have not added these optimizations to our process-based server model.

Threaded A threaded server works on the same principle as a forking server but it services new client connections using a new thread, instead of a new process. A thread is a “light-weight” process in that the thread contains some of its own context information but it shares some information with the process that

created it as well. This information sharing is illustrated in Figure 1.3, where one can see a single process with the threads for each client connection being completely contained inside the process. The information that is shared by the threads contained inside the process include things like file handles, global variables and other resources. This shared information needs to be protected, via synchronization, so that multiple threads do not attempt to modify the shared information at the same time. This thread synchronization can be difficult to keep track of and may lead to hard-to-find bugs in server code.

A thread pool is usually used in high-performance servers and we have incorporated one into our threaded server model. The idea of a thread pool is similar to the pre-forking and process reuse in a process-based server.

Figure 1.3 is actually missing a work queue that is used by the threads inside the main process. We have not shown this queue in Figure 1.3, as it is an optimization that is not necessary for this type of server model to function properly, and omitting it makes the distinction between our threaded model and the event model more clear.

Event-driven Event-driven servers are different from the other models in that they are usually single process/threaded servers that handle incoming or outgoing events when needed. They do not spawn new processes or threads to handle new incoming client connections. For each client connection, the server needs to have a collection of information set aside that can be used to service subsequent events related to that client's needs. This collection of information is called an event structure, and it contains things like the client's socket handle, open

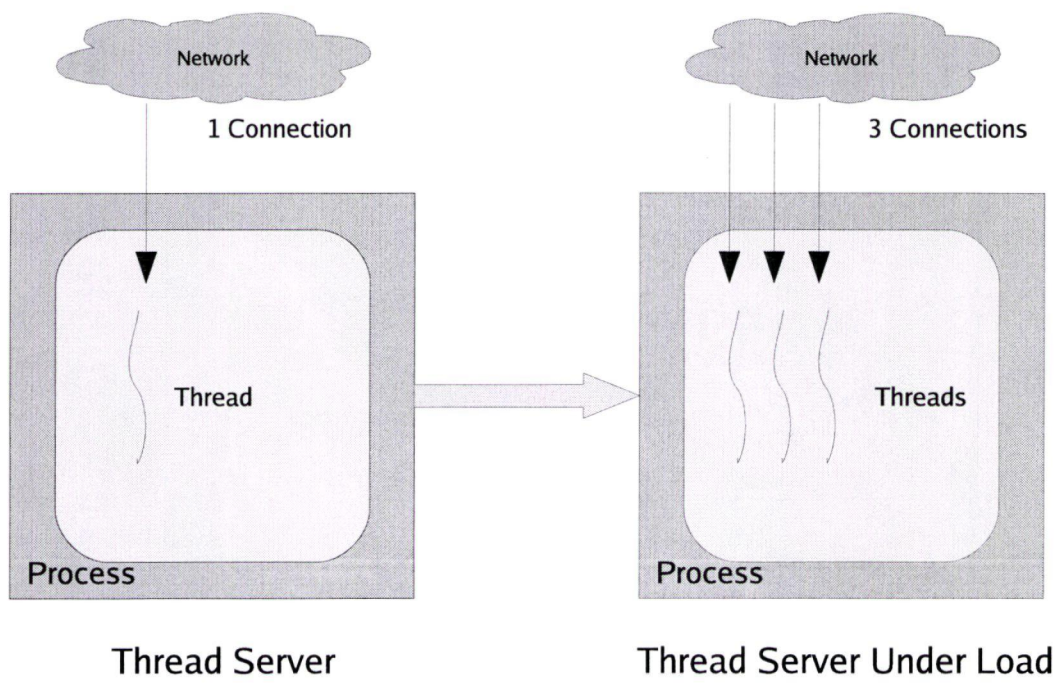


Figure 1.3: Threaded Server

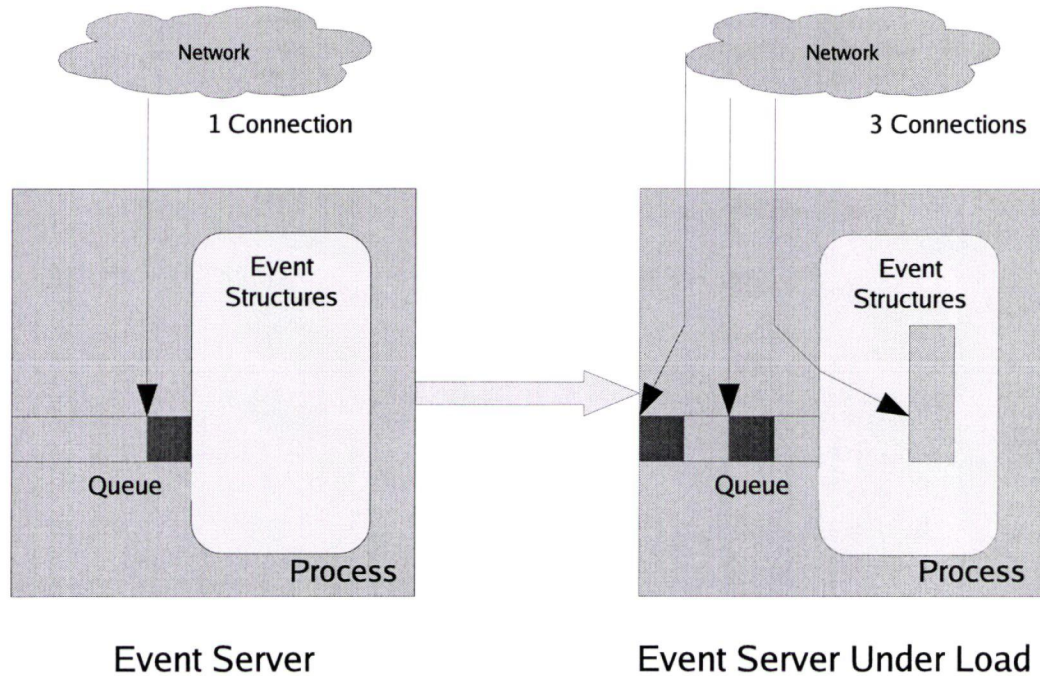


Figure 1.4: Event-Driven Server

file handles and allocated memory structures. As can be seen in Figure 1.4, there is a queue that is filled with pending events, until the server is ready to service the incoming request or action. When an operating system signals the main process or thread that an event has occurred, the main process/thread loads the appropriate event structure and services the event. If other events occur during this time, they are queued for future servicing. Pre-allocating event structures and using non-blocking I/O are techniques that are used to help event-driven servers perform better under heavy load.

There are advantages and disadvantages to these models, each with years of research and/or practical application to support them. NEST: Network Server Tool²

² Previous work on NEST has been published as a Technical Report[48] at the University of

has been developed with the aim of being able to switch between these three models quickly so that new Internet servers using any of these model types can be tested with a minimum amount of time and effort. With user-supplied code, complete server applications that run on the Linux operating system can be automatically generated using NEST. NEST uses compiler technology to automate server code generation for a given specification. We will give a brief overview of some compiler principles in the next section.

1.2 Compilers and Tools

Compilers are complex pieces of software that can be simply thought of as translators that take some kind of coded input and, according to a specific set of rules, output a translation of that input. Many programming languages are compiled from the input language to a low-level assembly language. This low-level assembly code is then “assembled” into machine language, the zeros and ones that the machine operates on.

There are certain tools that can be used by compiler developers when developing new languages to help ease the compiler-building process. Two popular tools in this development process include Lex[31] and Yacc (Yet Another Compiler Compiler)[24]. These tools, or their more recent incarnations, are usually used in combination.

Lex breaks up any input text into usable “chunks” called tokens. A programmer supplies Lex with a number of patterns to match, and Lex will forward the longest possible match of these patterns to Yacc for further processing.

Calgary, a poster presentation based on this technical report was given at IC 2005 in Las Vegas[49] and a full paper submission at the forthcoming APCC 2005 Conference in Perth, Australia[50]

Yacc matches the stream of input coming from Lex against a specific grammar. The function of this language grammar is similar to the English language grammar; it is used to check for proper syntactic use of a language. If a program contains any grammatical errors then a compiler builder can use Yacc to check for these errors and terminate the compilation process with an appropriate error message. Yacc also gives access to the underlying C language so that a compiler builder can take advantage of all of the C language facilities.

The use of compilers and automation tools for network server development has not been as prevalent as the use of libraries. Though some researchers have argued that language design and library design are equivalent[27], others argue that “most useful application specific languages perform domain-specific analysis at compile time”[28, page 5] to differentiate themselves from libraries, and thus justify their construction and use in a particular domain. We agree with the second assertion and have developed NEST with the belief that it will be useful when developing and testing servers.

1.3 Motivation

The programming of network servers is both tedious and error prone. NEST is a new tool for automatically generating most of the communications code infrastructure that TCP/IP-based servers require. This offers several advantages, such as:

- Easy switching between different server models.
- Decreased development time/improved programmer productivity.

- Easy prototyping of new protocols.
- Higher-level network protocol specification, abstracting away implementation details.
- Reduction in the number of bugs and security problems.

Some of these advantages are a direct result of using a tool, while others need more explanation.

Using our tool, a programmer can choose to generate code for event-driven, threaded or process-based servers. Switching from one type of server to another can be done by simply changing one line of code in the NEST specification. All underlying code changes between server types are transparent to the programmer, unless they desire to delve into the generated code for finer control of their server. There is no consensus as to which server model is best, and one traditional argument even states that threaded servers and event-driven (or message passing) servers are essentially a “duality” [29]. We have given programmers the ability to choose between different server types so that they can quickly test the different models if they so choose. Some previous work suggests there are places where event-driven servers are desirable [46]; other work suggests general difficulties with threaded code [35]. There is also evidence that process-based servers can perform adequately (e.g., Apache) and that programmers may prefer this model due to ease of programming.

Some common errors and poor programming practices in Internet applications can be avoided or minimized when using a tool such as NEST. One example is the ubiquitous buffer overflow [4]. A buffer overflow occurs in a program when insufficient bounds checking is performed on data to be placed in a buffer. These overflows may

occur, and thus be exploited, in statically allocated and dynamically allocated data. In some cases, one single byte of overflow can lead to an exploitable server[26]. Our tool offers some protection against buffer overflows because a portion of the buffer handling code is automated. Buffer overflows accounted for around 60 percent[47] of all vulnerabilities in 2003. With the prevalence of buffer overflows we feel that any tool or process that may help minimize their number would be worthwhile to consider.

Keeping buffer overflows and similar errors to a minimum via automatic code generation, should lead to applications with fewer security holes and less programming errors when compared to manually-written server code. On the other hand, the prevalence of tool-generated code could also introduce a single commonly exploitable error into hundreds of different servers. While this is possible, tool-generated code need only be security audited once, instead of having to inspect hundreds of different server implementations. There are also new techniques, such as address obfuscation[10], Position Independent Executables[18], failure-oblivious computing[39] and microrebooting[13], being explored that could be used to help mitigate the effects of these common errors if they occurred in our automatically-generated code. The trade-off between “time saved” versus “security problems” introduced by our NEST tool is an area of research we have yet to explore.

One final advantage to using an automated coding process for servers is that there will be fewer alignment problems. In other words, there will be fewer problems arising from a change in the number or order of function arguments in a server without a corresponding change in the client. Although we do not generate client software from a NEST specification, it would be easy to do so, and thus closely couple the client and

server so as to eliminate the alignment problems altogether. Many other networking suites mentioned in this thesis are actually libraries or frameworks that exhibit an alignment problem to one degree or another. In a specification language approach such as ours, these alignment problems can be fixed with a simple recompile, and users of NEST wouldn't have to update a lot of code by hand.

The following structure will be used for the rest of this thesis. Chapter 2 will discuss some related work. An in-depth presentation of the specification language, with examples, can be found in Chapter 3. Next, implementation details of NEST will be given in Chapter 4, followed by experimental results and evaluation in Chapter 5. Finally, some conclusions and future work are presented in Chapter 6.

Chapter 2

Related Work

There are several languages that can be used for rapid server implementation as well as several frameworks and libraries that can be used to develop servers. Protocol specification languages are also related to our work, though they tend to focus on lower level protocols than NEST. Some miscellaneous projects share several design decision commonalities with NEST, as well.

2.1 Languages

The languages described below share some attributes with NEST but differ greatly in other aspects. We outline the only other language we have found that is very similar to NEST, as well as one other language for a specific high-level protocol and one for lower-level protocol development.

2.1.1 MSPL

A specification language approach to server code generation has been described and implemented by Melvin Douglas[16]. The language, called My Simple Protocol Language (MSPL), can be used to easily describe an Internet protocol.

His compiler takes a specification and uses it to generate the communication and protocol modules for both the client and server applications. An MSPL user must also supply some extra code to be included as a user module to finish off the server

or client, as the case may be. One of the drawbacks of MSPL is that it gives no immediate access to the underlying implementation language, Java. What we mean by this is that the programmer cannot use Java in the actual protocol specification, rather they must modify the generated code after the fact, or add code in a separate user module outside the specification. Unlike our NEST tool, which is made for server code generation, MSPL is geared more towards protocol specification and testing.

MSPL seems to be somewhat limited in its flexibility and expressiveness. There are no performance statistics shown for any of the example protocol implementations described in the thesis. The author does admit that the MSPL language has sacrificed speed for the portability available through the use of the Java language. There is also no technical information on the underlying structure of the communications between the generated clients and servers. We can deduce some of the structure from the generated code given in the thesis. From this structure we conclude that the communications structure is rather naive, and not made for high concurrency or high server load. Speed is not necessarily compromised by the use of Java, since we know that some server implementations that use Java are still very fast[46], but it would seem that the naive infrastructure design is the cause of any MSPL server deficiencies.

Though MSPL has its limitations it does have one major benefit over our tool, which is the automatic generation of a *client* for any given protocol description. This leads to more rapid testing of the specified protocol and a tighter coupling between clients and servers. It may also lead to easier debugging of a new protocol.

2.1.2 MAWL

MAWL[28] is a domain-specific language for use in developing interactive World Wide Web (WWW) pages. Languages like MAWL are geared towards automation of many of the tasks that HTTP servers perform. In the case of MAWL, it automates state management, synchronization and shared memory using its compiler and an extension to the Hypertext Markup Language(HTML) called MHTML. Regular HTML is a stateless language that is heavily used on the World Wide Web to serve out different content to clients. HTML works very well when serving static content but it falls short when dynamic content is requested by a client or when stateful transactions are required.

MAWL gives access to the underlying host language, Standard ML, and only introduces new constructs when this general-purpose language cannot be used to solve a given problem. MAWL is similar to NEST in this regard and, like NEST, makes some of its design decisions based on the successful aspects of Yacc.

One advantage of using the MAWL compiler is that it performs static checking of its input to guard against common WWW programming errors. Error handling in MAWL form submissions is implicit, which is another advantage of this language. Users are returned to forms that have some type of incomplete or malformed data input. Static checking in NEST is minimal due to the fact that NEST was designed for maximum flexibility when developing networking applications.

New languages or language extensions have recently been developed to address some of the same issues that MAWL was originally developed for, such as state management. These languages or extensions include JavaServer Pages, Active Server

Pages and PHP.

2.1.3 Morpheus

Morpheus[2] is a theoretical programming language that can be used to describe communications protocols. We have no knowledge of a Morpheus compiler in existence, and thus we apply the designation of “theoretical” to the Morpheus language. The author of Morpheus does have some C language simulations of several protocols in his paper. This gives him the ability “to report some preliminary performance measurements based on hand optimizations of an implementation of the Morpheus protocol architecture in C”[2, page 5].

Morpheus can be used to implement new low-level protocols as well as higher-level protocols through composition of these lower-level protocols. Morpheus concentrates on defining many simple protocols that can be combined to make a more complex and functional protocol. One interesting feature of Morpheus is its optimization mechanisms that can reduce per-layer overhead when combining these simple protocols.

Morpheus also avoids message header alignment problems and byte-swapping overhead, to a certain degree. There are no message header alignment problems because a Morpheus compiler automatically pads any headers accordingly. Byte-swapping overhead is kept to a minimum by in-lining any code for assignments that may appear in the source program. Morpheus also has an internal byte ordering keyword that is used to avoid machine-dependent inconsistencies (i.e., big endian versus little endian) when representing data in a message header.

Some of the main contributions Morpheus makes are protocol abstractions and

optimization techniques. Protocol abstractions help support protocol development through the building-block approach, which is a layering and reuse technique that speeds development time. Morpheus also preserves modularity which makes the building-block approach to protocol generation easier to maintain. Though this building-block approach is not useful for making TCP/IP implementations, the author contends that it may be the next evolution of protocol design. Morpheus’s “procedure cloning” technique extends a compile-time inter-procedural optimization to run-time without a performance penalty. There is evidence that exceptional performance may be obtained using similar techniques[38].

Though Morpheus has some very good ideas for protocol generation, we believe that a working compiler would make some of its main contributions more accessible.

2.2 Libraries

Libraries are frequently used for server development, due in part to the fact that programmers are already familiar with the language used to develop and interface with the library. This means that development of a server can start right away rather than starting off with learning a new programming language. In the following subsections we will describe several libraries written in, and accessible from, diverse programming languages.

2.2.1 Serveez

Serveez[23] is a library written in C that provides much of the functionality necessary for quickly writing event-driven Internet servers. One of its main goals is portabil-

ity so it can be used on many different platforms, though other tools mentioned here suggest that portability is one of the main benefits of using other languages such as Python and Java. Despite using the C language, Serveez can be used on approximately 20 flavors of Unix and Windows.

Though this library is written in C, it can use another language called `GUILE`[20] to specify the structure of a server. `GUILE` is actually a Scheme language interpreter that can be used to easily write server implementations without having to modify the Serveez internals. Users can simply write a `GUILE` program that uses the appropriate library functions to build the server of their choice. We contend that this `GUILE` “interface” to the Serveez library doesn’t actually qualify Serveez as a specification language approach to server generation like `NEST`, since there is no code generation. It is also possible to write “embedded” servers with Serveez. These servers use dynamic library calls to define a server program and are “embedded” into a C program. In this case, “embedded” means that the Serveez code is incorporated into the server via shared library calls, which contain the server implementation.

There are some limitations in Serveez that arise due to the use of the `poll/select()` system calls for network I/O in Serveez. Most libraries mentioned here, including Serveez, use non-blocking asynchronous I/O for increased performance, but using `select()` can lead to lower performance in some situations. There is also an open file descriptor limit when using `select()` that peaks at around 1000 on an older Linux 2.2 kernel[23]. Serveez uses the `poll()` system function, if available, to get around this limit but faster and more scalable mechanisms like `epoll()` or `kqueue()` are not used in Serveez, as yet. Serveez is single-threaded, though the authors do mention that multi-threading can also be used as an alternative to their event-driven model, to

serve many simultaneous clients.

2.2.2 BEA Tuxedo

BEA's Tuxedo[8] is a commercial library that supports several different communication methods for software in a business environment. It is a very large and complex middle-ware system for use in large business transaction processing. It is continually being developed and several new versions have come out since starting our research on this project.

BEA describes Tuxedo as a middle-ware product that provides "distributed transaction processing and application messaging for applications that operate across multiple hardware platforms, databases, and operating systems." [8] The underlying technology of Tuxedo is a library with many functions available to programmers of servers and clients. The functions in the Tuxedo library can be accessed from Java, C, C++ and Cobol interfaces. The library supports event-driven and multi-threaded applications on several platforms but the internals of the program are not easily deduced due to the fact that Tuxedo is not open source. Tuxedo is geared towards transaction processing but it is also the basis for BEA's very fast, fault-tolerant WebLogic HTTP server. Due to the closed-source nature of this system we are not able to examine any of the internals that lead to its performance.

2.2.3 Twisted

Twisted[30] is an open-source asynchronous networking framework, written mostly in Python. Twisted uses an event-driven model for communication and, unlike many other tools mentioned here, it allows access to the underlying platform specific fea-

tures if a programmer wishes to exert more control over their system. The use of Python as the development language for Twisted allows for easy sub-classing, memory management and cuts down on some types of security problems like buffer overflows. This language choice does have a major drawback however: it is slow when compared to C or C++ frameworks. This is evidenced by the statement “High performance is not a major goal of the Twisted framework.”[30, page 2] We had gathered some preliminary test results involving the Twisted web server, but we have not continued with this testing due to its poor performance compared to the C or C++ web servers.

Twisted is a very comprehensive and large framework that may be used for easily implementing most conceivable networking servers. The framework comes with examples for many common Internet protocols, including HTTP, IMAP, and FTP. It also supports several low-level transport protocols, such as TCP, UDP, SSL/TLS and Unix sockets.

Twisted.spread[43] can be used to develop clients and servers in tandem. There is no compiler used to generate code automatically: instead, a collection of Python classes are used to easily construct clients and servers as well as the protocols for communications between the two. There is a low-level marshaller for a limited set of data types, a broker for copies of objects and a serializer of Python objects included in Twisted.spread to ease the development process.

2.2.4 SEDA

The Staged Event Driven Architecture (SEDA)[46] is a framework for developing event-driven network servers. Matt Welsh developed this architecture and provides

good empirical evidence to support the hypothesis that event-driven servers can handle large concurrent loads better than either process-based or threaded servers. The line between event-driven and threaded is a little blurred in SEDA due to the fact that Stages use dynamic internal thread pools, but also use externally-exposed event queues. The author *does* call SEDA an event-driven framework, however.

SEDA is an open-source project, with an implementation called Sandstorm that has been written in Java. This framework also makes use of non-blocking network and file I/O, which was not part of the Java language when SEDA was being constructed.

A major goal of SEDA is fairness to clients under heavy load, as well as high performance. This is accomplished through load shedding, with client notification, and self monitoring to manage resources optimally. If any given Stage is overloaded then previous Stages can shed some load dynamically, or reconfigure to off-load work to another stage when possible.

One drawback to this system has actually been touted as an advantage of using Java, namely memory management. Java uses automatic garbage collection, thus alleviating the burden of task cleanup from the programmer, however there is evidence[41] that general-purpose garbage collection techniques can be detrimental when used in high-performance servers.

2.2.5 ACE

The Adaptive Communications Environment[40] is a large framework for writing concurrent communications software. It is written in C++ to support object orientation and design patterns, which are recurring solutions to a standard problem. Patterns purportedly increase the flexibility, re-usability and modularity of the ACE

framework when writing communications software. There is also a recently developed Java version of ACE available to the public.

At the lowest level, the operating system (OS) adapter layer sits on top of specific application programming interfaces (API) to shield other layers from OS-specific dependencies. The adapter layer includes multi-threading, multi-processing, event demultiplexing, shared memory and filesystem accessing mechanisms¹.

There are several other layers like C++ Wrapper, which acts like the OS Adapter Layer, but is meant to be used from within C++ programs. There are also frameworks that enhance the C++ wrappers with dynamic service configuration, hierarchically layered stream components and Object Request Broker (ORB) components. Two main projects using the ACE framework are the TAO ORB and the JAWS web server.

JAWS is touted as a high performance adaptive web server that is very rich in features. It supports different concurrency models, I/O models and a filesystem caching mechanism for improving performance. JAWS is a framework of frameworks that is built by extending and combining other components in ACE.

2.2.6 Capriccio

Capriccio[45] is a very new and experimental development project from the University of California, Berkeley. This new user-level thread package or rather, this optimized version of the old GNU Pth threading package, is used to dispute claims that the event-driven model is the best for writing highly concurrent Internet servers

¹ACE also includes support for low latency, high performance, high bandwidth and quality of service requirements which makes it suitable for embedded and real-time servers [40].

or transaction processing applications.

The authors argue that threads provide “a more natural abstraction”[45] for writing high-concurrency applications or servers. They also show that small changes to compilers and threading libraries can lead to a more useful paradigm than the traditional event-driven one. They point out that another group has already tried to “just fix events”[3] which actually leads back to a threading model anyways.

An important part of the Capriccio package is the use of a tool to modify the output of a compiler. This tool performs some optimizations that make use of the new Capriccio threading model.

The Capriccio authors show some impressive results when comparing their new web server, Knot, to a previous implementation of the SEDA-based web server called Haboob. The new server can service up to 65,000 clients at around 700 Mbits/sec. Haboob’s maximum throughput is around 500 Mbits/sec, with a maximum of approximately 16,500 clients able to connect before the server runs out of memory.

We believe that new systems like Capriccio could be incorporated into NEST as yet another model type to choose from. This could lead to more rapid comparisons between new server models, or performance enhanced versions of older server models.

2.3 Protocol Specification Languages

Protocol Specification Languages are similar to NEST, but they are geared towards lower-level protocols and their verification or validation.

2.3.1 Promela++

Promela++[7] is an extension of the protocol validation language Promela. It can be compiled into efficient C code. It also allows automatic protocol verification against programmer-specified requirements via the Promela language. The C-like Promela++ language takes advantage of many programmers' previous knowledge of this popular language. Promela++ provides for layered specification of protocols, composition of these layers into protocol stacks via an event-based mechanism and encapsulation of states and message passing headers. The Promela++ execution model is based on Horus[44].

Though Promela++ seems to be useful for lower level protocols, it is not targeted at the application layer we are concerned with. It simply exposes a function call interface to an application and hides the protocol internals. Promela++ does not support an explicit memory allocation mechanism, which makes allocation somewhat cumbersome. It also has no support for timeouts in real time, which are required for a TCP implementation.

2.3.2 HIPPCO

HIPPCO is "A High Performance Protocol Code Optimizer"[14]. It builds on the HIPPARCH compiler which is built around a synchronous language called Esterel[9]. HIPPCO optimizes a control automaton just below the application layer, and uses an Application Level Framing (ALF) compiler as part of HIPPARCH. The Esterel language offers a library of pre-defined modules, a parser and a front end that compiles the Esterel code into an automaton. HIPPCO optimizes the automaton and produces efficient C code as its output. There are a great number of optimiza-

tions implemented by HIPPCO, including instruction count optimizations, input rescheduling and branch pruning.

HIPPCO outputs client and server APIs, which are macros that differ according to the input specification. The focus of HIPPCO is still on the lower level protocol specification and its optimization. [14] compares the performance of the BSD TCP implementation versus a HIPPCO-optimized implementation. HIPPCO's TCP performs significantly better, in terms of instruction processing by the CPU to process an incoming packet, but there is a penalty of increased code size when compared to hand-coded TCP implementations. The authors of HIPPCO state that they “envisage integration of the application into the communication automaton” [14], which would automate protocol and application production even more.

2.4 Miscellaneous

This section reviews some tools that are similar to NEST in design or function. The tools are used in different ways, and one is not even used for networking software development. Both of these tools have some aspects in common with NEST, and are therefore deemed to be relevant, though they were developed much earlier.

2.4.1 FistGen

FistGen[51] is not a networking related tool, rather it is a tool for automatically generating code for stacking filesystems. From a single description, the FiST language compiler can produce filesystem modules for multiple platforms. The generated code handles many kernel details, freeing developers to concentrate on other major issues

that relate to their filesystems.

The reason that we mention FistGen here is because of some striking similarities of design. Both FistGen and NEST are based on Yacc, and Lex to some degree, with some similar syntax and compilation mechanisms. Though we only recently discovered FistGen, and NEST has been completely independently-designed and implemented, we felt that the similarities in design between our projects in some way validated our implementation. FistGen is a well designed and usable system that has continued in its development for years, thus proving that it has merit as a good tool for generating stacking filesystems.

2.4.2 RPCGen

RPCGen[22] is a protocol compiler that is used with Sun Microsystems' Remote Procedure Call (RPC) protocol. RPC has been around for a long time[11] and is used in most operating systems today.

RPC sits at the presentation layer of the OSI Model in Figure 1.1. This means that it operates at a slightly different level than NEST, but it is still relevant. The RPCGen compiler produces stubs for both server and client code using RPCL, the Remote Procedure Call Language. This language is similar to C, which leverages many network programmers' prior knowledge of this language. The output code produced by RPCGen is also C code, like NEST. A server skeleton is produced that can be compiled and linked with when producing a full-blown server. There is also an eXternal Data Representation (XDR) filter routine for parameters and results being passed back and forth between client and server. The XDR routines convert the parameters and results into network format, and back out of network format.

Many of the servers written using RPC are not secure; this is a common source of problems [19]. This doesn't mean that RPC is not popular, quite the opposite. The Network File System (NFS) uses RPC, and is a very popular remote filesystem used on most operating systems. A language that produces a larger amount of code, to safely wrap the procedure calls, may be useful in mitigating some of the more common RPC server coding errors.

2.5 Summary

In this chapter we have reviewed many different languages and libraries that are similar to NEST, or that have some bearing on our research. There are more libraries available for server development than languages. Hopefully in the future there will be more, and more useful, server languages that appear to help programmers avoid some server implementation pitfalls. Work in the server performance arena is continuing with a fervor and we believe that incorporating this work into a code generation tool would be useful.

Chapter 3

NEST and its Specification Language

3.1 Introduction

A good portion of the research effort for this thesis was spent on the specification language for NEST. Over the course of the language development we ran into several interesting problems that were not expected. We will cover the specification language in its final form, and explain some of the problems we ran into during its development.

3.2 Specification

The NEtwork Server Tool has a general structure like that found in Figure 3.1. A user supplies a NEST specification to our compiler, which translates the input into appropriate C code for any of our three server model types. The automatically generated code is then compiled by a regular C compiler; in this case we use the gcc compiler that is supplied with most versions of the Linux operating system. There is also a secondary file containing extra C code to be compiled into the final server. Once everything is compiled we link the generated code and extra code, as well as any library code that a user would like to use, to produce the final server executable.

The extra code file is filled with many functions that do not change across server model types. This file also contains some functions that may require conditional compilation techniques to provide the appropriate functionality depending on the

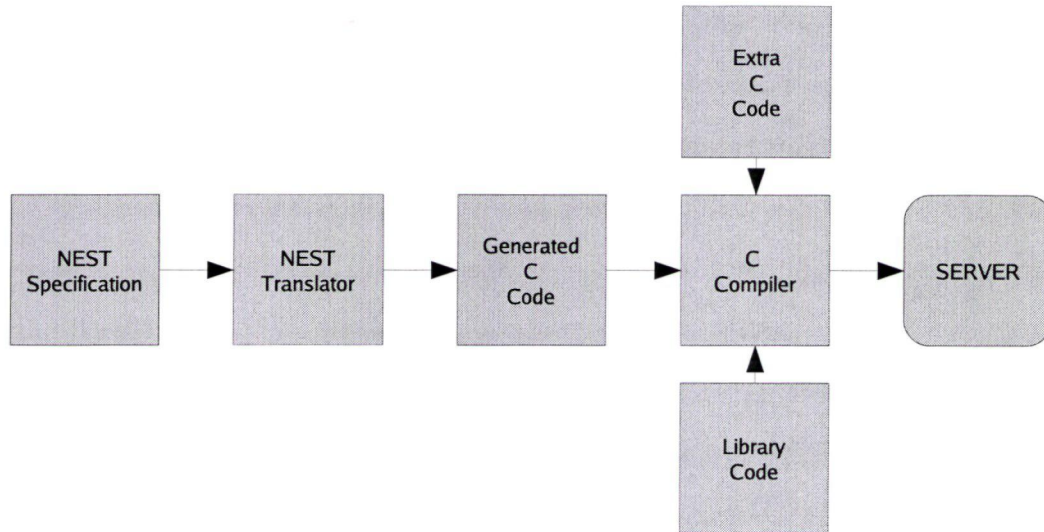


Figure 3.1: Server Generation

model type chosen. Users of NEST may add code to this file if they prefer.

In the NEST specification file we have given the user the ability to choose between two different types of connection models: a stand-alone server or an `inetd` server. A stand-alone server is one that runs continuously, listening for new incoming connections. An `inetd` server is one that is run on demand by the Internet super-server `inetd`. Switching between these two connection models can be accomplished by changing one line of code in the NEST specification.

In Figure 3.2 we illustrate the difference between the two server connection models. The stand-alone server has more resource usage, but a faster response time due to the continuously running server daemons. The stand-alone model should be used for high volume servers. The `inetd` connection model uses less resources, on average, and should be used for lower volume server machines.

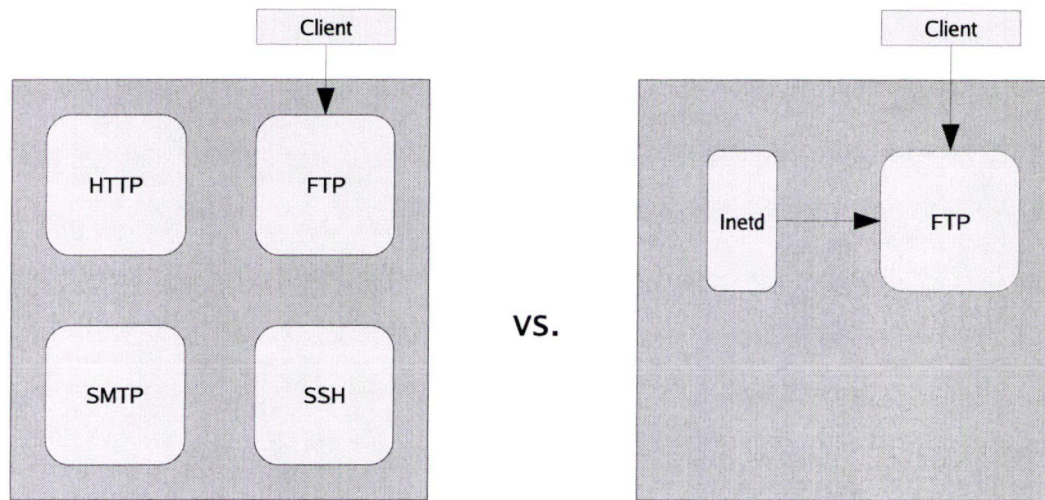


Figure 3.2: Stand-alone vs. Inetd Server

3.3 The Specification Language In Depth

The specification language design was inspired by the design of Lex and Yacc[32]. This decision was made due to the fact that Lex and Yacc are popular compiler tools that have stood the test of time. It is true that newer tools have come into being that are improvements on Lex and Yacc, but the basic design is still solid. There are other advanced projects that use similar syntax and design, like FiST, which lends support to our contention that this is a good design choice. See Appendix A for the Yacc grammar we used to develop NEST.

In our specification language, as in Lex and Yacc, there are three sections, separated by %%:

declarations

%%

rules

%%

C code

The last section is simply C code that is copied directly to the output file. Figure 3.3 gives a short sample specification, for a simple server that echoes user input in encrypted form.

3.3.1 Declaration Section

The initial part of the declaration section is arbitrary C code, encapsulated between %{ and %} delimiters. The user may include header files or define C preprocessor macros in this part of the specification. The C code contained in this section is copied verbatim into the output file of the NEST tool. This is the place where users would place “global” variables. These variables are accessible from within the entire specification file and, more specifically, from within the action code sections of the specification. One could also access these variables from the “extra code” file using C’s extern keyword which is part of the C programming language. While global variables are generally regarded as hazardous and can make code harder to maintain[33, page 360], this functionality is available in NEST because it is available in the C programming language.

```
%{  
    #include "rot13.h"  
}%  
  
%option threadServer 100  
  
ANY [^\n]*  
NL  \r?\n  
  
%%  
  
char buffer[256];  
  
START: {  
}  
  
DEFAULT: ANY NL {  
    strcpy(buffer, $1, sizeof(buffer));  
    rot13(buffer);  
    $reply(buffer);  
}  
  
%%
```

Figure 3.3: Simple NEST specification

The remainder of the declaration section is used for defining NEST options, states and macro substitutions. The commands `%option` and `%state` are used to inform the translator that a single server option or that one or more states will follow.

Some of the options in the declaration section in a NEST specification can be overridden at run-time by using a special file whose default name is `config.nest`. This file mechanism is supported to give network administrators the ability to tune a server's properties without having to recompile the server. An example of some options that can be changed include:

- `serverName` - set the server's name
- `serverPort` - port to run the server on
- `inputBuffer` - default size of the server's input buffer
- `outputBuffer` - default size of the server's output buffer

The `standAlone` option can't be modified at run-time as different code is generated depending on whether the user wants a stand-alone server or a server that is run by `inetd`.

The format of the configuration file is the same as the syntax used for options in our NEST specification file, without the `%option` qualifier. Each option name is followed by a space and an appropriate value.

The `%state` command is used to declare server states used in the specification. This means that the user has effectively added new keywords to the NEST tool so that it can later check for proper usage of these state names.

As in Lex, macro substitutions defined in the declarations section may be used to increase the readability of large or complex regular expressions and boolean state expressions. There are no practical limitations to the complexity of these substitutions in the NEST compiler. The syntax of substitutions is simply one collection of characters followed by a space and then the collection of characters to be substituted, ending with a carriage return. This feature helped keep specifications readable when large and complex regular expressions were written for our test servers.

Our tool does not support the declaration of variables, in this section, as there is no meaningful way to use these variables in the output code. Any variables declared here cause our compiler tool to output a warning message about improper use, but the compilation does not stop. If we wanted to support variable declarations in this section we would need to implement a full C parser, which would be needed to correctly recognize the declarations when compared to any macro substitutions. For now, we recommend that users do not declare any variables in this section, just %options and %states, as the declared variables will not appear in the final generated code. We will discuss this further in the Implementation Details chapter.

3.3.2 Rules Section

Following the declarations section is the rules section of our tool which starts and ends with %%. The user begins this section by declaring any per-connection variables that may be needed. These variables have a different meaning depending which model NEST is generating. For the process-based and threaded models these variables are simply copied into the generated output code near the start of our monolithic function. (This monolithic function contains most of the logic for state transitions

in a specified server, plus all of the code contained in the action code sections of the supplied NEST specification.) Therefore, if a process-based or threaded model has per-connection variables declared then these variables will have “function scope” within our monolithic function. However, if an event-driven server is being generated then any per-connection variables need to be protected in a specific way as they will have a scope that encompasses the entire file. We explain the necessity for this protection mechanism in Section 4.3.2. Figure 3.3 contains a per-connection variable called “buffer” which is 256 bytes long.

In the rest of the rules section, qualifying expressions and patterns are specified which NEST will try to match against the server’s input. If an input pattern is matched, and its qualifying expression is true, then that pattern’s corresponding action code is executed.

Network servers often have some notion of state in their transactions with a client. For example, a client may initially be in an unprivileged state, and move to a privileged state upon authentication. NEST specifications therefore have extensive support for states. Each state can be thought of as either being true or false, depending on whether the state has been “seen” yet or not.

Each state has a name, and state names are used in conjunction with each other to construct boolean expressions. Patterns in the specification are qualified by these boolean expressions; recall that a pattern only matches if its qualifying boolean expression is true. Valid boolean operators in NEST include the logical and (`&&`), or (`||`), and not (`!`) operators. The boolean predicate `$LASTSEEN(STATENAME)` may also be applied to a state, returning true if and only if that `STATENAME` was the last state to be set to true.

NEST has two predefined states called START and END that are reserved for startup and cleanup code. The START state can be used to set up the server prior to accepting any initial incoming data. (Some startup code can also be placed at the beginning of the last C code section mentioned above, but a dedicated start state is useful as well.) The END state can be used to send back error information before dropping a connection if the user wishes, as this state will always be entered before a connection is closed, in a well-behaved server (i.e., one that doesn't crash!).

With the exception of START and END, which are special cases, all the patterns and qualifying expressions in the rules section take the form:

```
[qualifying-expression:] pattern {
    action-code
}
```

The patterns that follow the qualifying expressions are actually regular expressions, extended to permit extra whitespace for readability. A regular expression is a string (a collection of characters) that describes another set of strings according to certain syntax rules. What that means is that we can express patterns very easily using regular expressions. For example:

```
mv *.txt ./docs/
```

is a Linux command that uses a regular expression to easily move all files that end with the extension “txt” to a “docs” subdirectory.

There are many different types of regular expression syntax rules, depending on the programming language one is using. We have based our regular expression

syntax on the Perl Compatible Regular Expressions library, or libpcre, syntax. This means that we can simply pass through user defined regular expressions directly to libpcre. As the name implies, libpcre’s regular expression syntax rules are based on the PERL programming language

NEST also allows syntactic sugar, so that one expression may qualify a number of patterns:

```

qualifying-expression {
    [qualifying-expression:] pattern {
        action-code
    }
    [qualifying-expression:] pattern {
        action-code
    }
    ...
}

```

We call this expression a multistate since one qualifying expression can be used to qualify several subsequently defined expressions.

There is one state that is internally defined in NEST, the “DEFAULT” state. It can be used for extremely simple servers to immediately transition from the START state to a state where some input text will be matched. The DEFAULT state is automatically \$seen when leaving the start state so that any state whose qualifying-expression is DEFAULT will be entered when leaving the START state. This means the NEST programmer doesn’t have to add a state and mark it as seen when leaving

the START state, because we have basically done this for them. The DEFAULT state was also needed to maintain a consistent *qualifying-expression: pattern {action code}* sequence in a NEST specification, as can be seen in Figure 3.6.

The example in Figure 3.3 goes from the START state and immediately transitions to the DEFAULT state, where it matches any given input plus a new-line character. The reason for this seemingly strange mechanism is that we needed a way (i.e., the START state) in which the server could send outgoing data to the client before having to match any incoming data. Figure 3.4 uses the START state to send a “Username:” prompt to the client prior to matching any incoming data. Our tool automatically generates a network read for each closing brace that ends an action-code sequence, which means that the START state attempts to read from the network automatically.

The code NEST generates will attempt to match the server’s input against all patterns whose qualifying expression is true. If there is ambiguity as to which regular expression can be matched, then the longest match is chosen or, if two or more matching expressions are equally long, the pattern specified first in the specification is used. Users should remain aware of the fact that ambiguities in this matching mechanism are allowed and the possible consequences of this first-written-first-served matching. This design decision will be expounded upon in Chapter 4.

In the action code, we allow a mixture of C code and special NEST directives. The latter are preceded by a dollar sign, and include:

\$seen Notes that a state has “been seen”, setting the state to true. In our design, we have left this for the user to do explicitly, to allow fine-grained control of state transitions.

\$clear “Forgets” that a state has been seen by setting it to false. The wild card * refers to all states except the START state. \$clear has no meaning for the END state.

\$reply Sends a reply from the server to the client.

\$close Closes the network connection in a manner consistent with the chosen server model.

\$return Used for “returning” out of any action code in our tool. This construct must be used rather than a bare return statement to guarantee a properly functioning server for all models.

\$fileReply Can be used for binary data file transfers.

\$fileReceive Accept incoming file data.

This is not an exhaustive list of our built-in functions, and any number of other functions could easily be added to help automate server infrastructure development.

Inside the action code we also have access to the matched regular expressions of the pattern using a Yacc-like mechanism. In Figure 3.3, the specification contains the pattern ANY NL. Each of the matched strings in the regular expression may be accessed individually: \$1 in the action code corresponds to the input text that ANY matched, and \$2 corresponds to the text matched by NL (which is actually the regular expression \n, after macro substitution). We can use these references in the action code anywhere that a character string can be used. A user should make a copy of matched input if they want to use that matched input in other action code segments

of the NEST specification. This is an appropriate place to use the per-connection variables, mentioned previously, to keep a copy of the matched input.

3.3.3 C code

The code that is placed after the last `%%` in the NEST specification is copied to the NEST output file near the start of the `main()` function. This `main()` function is the entry point for all C programs and therefore any code placed at the end of a NEST specification file will be run at startup in a NEST server. See Chapter 4 for further details.

3.4 Some Slightly Larger Examples

To further illustrate the specification language that NEST uses, we have provided two slightly larger examples. The first performs a simple user authentication before allowing HTTP-like commands; the second is a simple SMTP-like protocol.

3.4.1 Simple HTTP Protocol

In the start state of Figure 3.4, we reply to any incoming connection attempt with a username prompt. We then indicate that we have seen the state `EXPECT_USER`. This means that `EXPECT_USER` is the only active state when the next incoming data is analyzed. The incoming data will be matched against the regular expression `[^\r\n]*` (everything except for line terminators) followed by a newline. When the data is read in and matched, we enter the `EXPECT_USER` state where the input matched by `[^\r\n]*` is copied to the `user[]` character array. Other similar steps

```

%{
    #include <string.h>
    #include "myExtraCode.h"
}%
%option processServer 150
%option serverName www.cpsc.ucalgary.ca
%state EXPECT_USER EXPECT_PASS VALID

WS      [[:blank:]]+
NL      \r?\n
ANY     [^\x\n]*

%%
char user[256];
START: {
    $reply("Username: \n");
    $seen(EXPECT_USER);
}
EXPECT_USER: ANY NL {
    strncpy(user, $1, sizeof(user));
    $reply("Password \n");
    $seen(EXPECT_PASS);
}
EXPECT_PASS: ANY NL {
    char *pass = $1;
    if (validateUserPass(user, pass))
        $seen(VALID);
    else
        $clear(*);
}
VALID {
    "GET" WS (http://)?{NOT_WS}* {WS}? (.*)? {
        char *fileName = $3;
        char *version = $5;
        confirmVersion(version);
        $fileReply(fileName);
    }
    "PUT" WS ANY NL {
        $fileReceive($3);
        $reply("200 OK");
    }
    ANY NL {
        $clear(*);
    }
}
END: {
    $reply("Incorrect input - connection terminating!");
    $close();
}
%%
// C code

```

Figure 3.4: Sample of simple authentication and HTTP-like commands

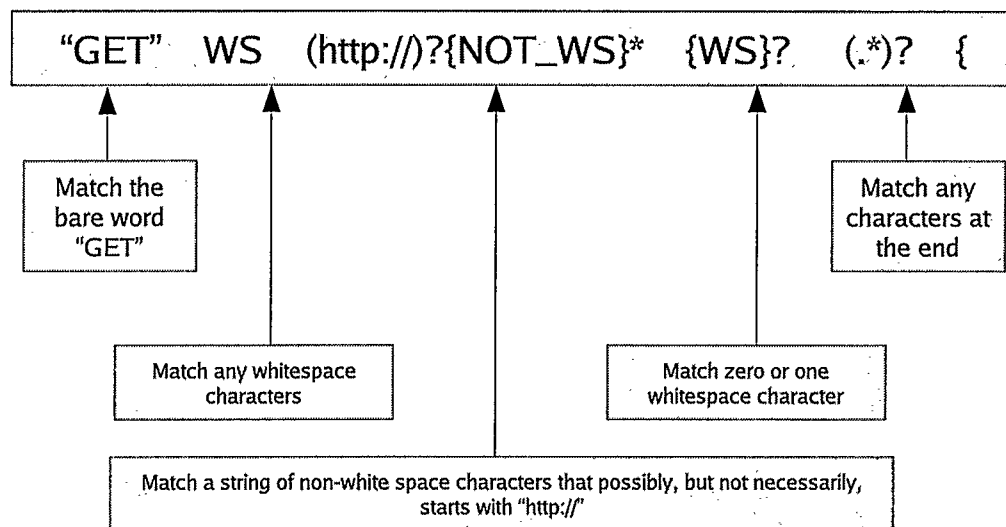


Figure 3.5: Regular Expression Expansion

will eventually authenticate the user and put us in the VALID state, or all states will be set to false via the `$clear(*)` statement.

Once in the VALID state, our HTTP-like GET and PUT commands are recognized. The pattern for the GET command looks complex, even with some substitutions for WS and NOT_WS. An interpretation of this pattern is shown in Figure 3.5. This concise, simple pattern matching mechanism was one of our research goals when we started developing NEST.

The last state of our Figure 3.4 example is the END state. This state will be entered on incorrect input, which will then cause the server to inform the client that something is wrong, and abruptly close the connection. Of course, in a larger and more well-behaved server a more appropriate error recovery mechanism would be used.

Looking at the declaration section, the `processServer 150` option indicates that


```

%{
    #include <string.h>
    #include "myExtraCode.h"
}%

%option threadServer 200
%option serverName smtp.cpsc.ucalgary.ca
%option inetd
%state GOT_HELO GOT_MAIL GOT_RCPT

WS    [[:blank:]]+
NL    \r?\n
ANY   [^\r\n]*
BOTH  GOT_MAIL && GOT_RCPT

%%

char mailFrom[256], rcptTo[256];

START: {
    $reply("220 mailServer Simple SMTP; %s", date());
}
DEFAULT: "HELO" WS ANY NL {
    $reply("HELO ", $3, "\n");
    $seen(GOT_HELO);
}
DEFAULT: "MAIL FROM:" WS ANY NL {
    strncpy(mailFrom, $3, sizeof(mailFrom));
    $seen(GOT_MAIL);
}
DEFAULT: "RCPT TO:" WS ANY NL {
    strncpy(rcptTo, $3, sizeof(rcptTo));
    $seen(GOT_RCPT);
}

GOT_HELO && BOTH {
    "DATA" (.\n)* NL "." NL {
        mail(mailFrom, rcptTo, $2);
    }
}

START || GOT_USER || GOT_HELO || GOT_RCPT: "QUIT" NL {
    cleanup();
    $close();
}

%%
// C code

```

Figure 3.6: Sample code for an SMTP-like protocol

a process-based server should be generated by NEST in this case, and that the maximum number of simultaneous connections (and thus processes) is 150. We also set the `serverName` option to a default value of `www.cpsc.ucalgary.ca`. Of course this default value can be over-ridden using the `config.nest` file.

3.4.2 Simple SMTP Protocol

The simple SMTP-like server in Figure 3.6 allows the `HELO`, `MAIL`, and `RCPT` commands to be issued in any order, but requires them all to have been issued prior to a `DATA` command. We feel that this explicit, and simple, state machine building mechanism is an attractive part of NEST. With our NEST tool we can construct relatively complex state machines in a readable format.

The `threadServer 200` option indicates a threaded server with a maximum of 200 active threads. We also choose the `inetd` option in this example to select the startup model for the resulting server.

This threaded server specification also shows macro substitution used as part of a qualifying expression. The substitution of `BOTH` is performed which means that we must match `GOT_HELO && GOT_RCPT && GOT_MAIL` all together prior to entering the final state. This is what we intended and the substitution of two state names with one can increase readability of the specification.

This example also illustrates how the input “QUIT” can always be recognized, because all states are being logically “OR”ed together at the end of the input file. When we are specifying a catch-all statement like this we must keep some things in mind. For instance, if we have two states that can match the input (e.g., “QUIT”) then only the first state will be entered, and this may not be the intended behaviour.

One must also be careful to write the correct qualifying expression, as catch-alls can grow to be long and complex. Use of the substitution mechanism in these cases is recommended.

3.5 Summary

In this chapter we have given an in-depth presentation of NEST and its specification language. We showed some simple examples that highlighted the major syntax conventions used in NEST. We also described some of the underlying design decisions that were made during development of our NEtwork Server Tool.

Chapter 4

Implementation Details

The NEST compiler tool is written using the very tools it is based on: Lex and Yacc. We primarily use C code in our compiler, though some C++ is used where it could be used effectively to cut down on development time. The implementation of NEST consists of approximately 12,000 lines of Lex/Yacc/C/C++ code. The output from NEST is exclusively C code and currently runs on the Linux operating system and under Cygwin/Windows, as well. Our tool should compile on other versions of Unix, with little modification, since we have used the autoconf/automake suite of tools for the build process.

We have saved a significant amount of work, and coding, by using some library code in our server. Two libraries are used: libevent[37] is used for event driven servers, and libpcre[21] is used to match regular expressions in all three server types.

Libevent The libevent library is a versatile collection of C functions that may be called to take care of all kinds of events asynchronously. The library can be used to intercept file descriptor events, timeouts and signals. The library also supports several different types of polling mechanisms. Our implementation of NEST running on Linux uses the epoll polling mechanism due to its good performance characteristics and the fact that it is quite robust and scalable under heavy load[37].

Libpcre This library has been used to great advantage in NEST. The use of regular

expressions makes input matching easier, and the regular expressions are easier to read when using our substitution mechanism. The fact that we use `libpcre` to compile the regular expressions upon server startup keeps the matching fast when the server is under load. This library is also used internally to parse the `config.nest` file.

We have also considered automatically generating a Lex specification to match incoming patterns, instead of using `libpcre`. This solution would have made the compilation process more difficult and we didn't see a significant savings in matching speed so we have left this solution as possible future work.

Output from the compiler contains only relevant code for the server model that has been chosen in the NEST specification. Extra boiler-plate code, for all three model types, is provided with our compiler. Any extraneous boiler-plate code is conditionally compiled away during the final server build.

Most of the compiler generated code is contained in a single monolithic function. There are several reasons for this design decision, including execution speed, simplicity of design and ease of code generation. When switching between the different server models we found it easier to contain most code in a single function. We also assume that most users would not be viewing or modifying the generated code, therefore readability in this large function was not a primary consideration for us.

A large `"switch{ }"` statement is used inside the monolithic function to jump directly to the proper action code section depending on which states have been `$seen` and which are `$clear`. The evaluation of the boolean qualifying expressions is accomplished at the end of this main `"switch{ }"` statement, once the appropriate action

code has been executed and any state changes have taken place. This evaluation is done sequentially and thus each qualifying expression is evaluated in turn. Since this is inefficient for large numbers of qualifying expressions, we have spent some time looking at Binary Decision Diagrams[5] and similar algorithms to speed this evaluation process. We haven't spent a great deal of time or effort on this optimization, however, as we haven't seen servers with prohibitively large numbers of qualifying expressions in practice.

In the next several sections we will give annotated code examples of NEST input and output, for each of the three model types.

4.1 Process-based Server

The process-based server code was the easiest to develop. We could basically have the compiler output proper networking code that would work with one client, then proceed to “fork()” off new processes to accept more than one client. We didn't have to worry about many complexities that can plague the other models. Very little optimization has been done for this model as we wanted a baseline server to compare our other models against. We will review some input code, and its corresponding output code for the simple rot13 server in Figure 3.3 over the next section. This example produces a process-based server and its corresponding output code follows. The portions of code in light gray are the NEST specification code input and the dark gray code inside the boxes is the corresponding C code output. There are also some comments to help explain what is being done. Some routine or repetitive portions of the output code have been omitted for brevity.

```
%{
#include "rot13.h"
%}
```

```
#include <NEST.h>
#include "rot13.h"
```

We simply include our compiler's header file, as well as any user-supplied code.

```
%option processServer 100
```

```
#define NEST_MAX_CLIENTS 100
```

Here we use a “#define” statement to set the maximum number of clients that the server will allow.

```
ANY [^\n]*
NL \r?\n
```

```
const char *NEST_func0Pattern0 = {"^[^\n]*"};
pcre *NEST_func0Pattern0PCRE;
const char *NEST_func0Pattern1 = {"\r?\n"};
pcre *NEST_func0Pattern1PCRE;
```

These substitutions are equated with new libpcre-compiled regular expressions later in the server's “main()” function. In its current form, our tool takes any substitutions that are not used and throws them out, thus conserving space in the executable. There is also less work done at startup, as the unused substitutions are not compiled via libpcre.

```
%%

int NEST_totalClients;
typedef enum {DEFAULT, START, END} NEST_States;
const char *NEST_errorPCRE;
int NEST_errorOffsetPCRE;

void NEST_MainLoop(void *NEST_ac) {
```

This is some setup code and the start of our monolithic function called “NEST_MainLoop”. All libpcr variables have “PCRE” appended for clarity during development.

```
char buffer[256];

NEST_sd = *((int *) NEST_ac);
bool NEST_START = true;
bool NEST_END = true;
int NEST_ovectorPCRE[NEST_OVECMAX];
char *NEST_subjectPCRE;
int NEST_lastSeen = -1;
bool NEST_DEFAULT = 0;

    //User Variables
    char buffer[256];
    //End User Variables

char *NEST_func0PattMatch0PCRE;
int NEST_func0Patt0LenPCRE;
char *NEST_func0PattMatch1PCRE;
int NEST_func0Patt1LenPCRE;
int NEST_func0NumPatterns = 2;
int NEST_func0ExecPCRE;
int NEST_switch = 0;
```

Here we can see that the user-defined variables are declared near the start of our monolithic function. These variables are just copied into the code verbatim so that the user can reference them in their action code; we don’t have to modify this code in any way. The NEST_lastSeen variable will be set to the last \$seen variable’s offset in the enumeration NEST_states. Upon initialization it is set to -1, as no state has been \$seen yet.


```

START {

NEST_MainLoopStart:
{
    NEST_lastSeen = DEFAULT;
    NEST_DEFAULT = 1;
    (void) signal (SIGALRM, NEST_read_timeout);

```

The NEST_MainLoopStart label is used with goto statements so that we can jump back into this START code without re-initializing any variables. Dijkstra famously cautioned against the use of goto statements [15] in programming code, but there is evidence that this option is very appropriate for state machine construction[25].

We set up our NEST_DEFAULT and NEST_lastSeen variables so that we can transition to the next state correctly after we “NEST_Evaluate” the current state of our machine.

```

} //The end of the START state

for (;;) {
    NEST_ok = NEST_dollarRead (NEST_sd, NEST_buffer);
    if (NEST_ok == 0) {
        close (NEST_sd);
        NEST_totalClients--;
        exit (0);
    }
    else if (NEST_ok < 0 && (errno == EINTR || errno == EAGAIN)) {
        continue;
    }
    else {
        break;
    }
    (void) alarm (NEST_READ_TIMEOUT);
}
goto NEST_Evaluate;
}
NEST_FuncStart:

```

Here we have included the code for reading from the incoming socket, which is

connected to a client. This socket reading code could be moved out of our monolithic function, thereby saving some code generation, but we would just ask the compiler to “inline” this code anyways, so we decided to explicitly generate it here instead. The `NEST_FuncStart` label is used when we don’t want to re-enter the `START` state but would rather re-evaluate our qualifying expressions.

```

DEFAULT: ANY NL {

    strcpy(buffer,$1,sizeof(buffer));
    rot13(buffer);
    $reply(buffer);

switch (NEST_switch) {
//"DEFAULT" state is matched here
case 0:
{
    strcpy(buffer,NEST_func0PattMatch0PCRE,sizeof(buffer));
    rot13(buffer);
    NEST_dollarWrite(NEST_sd, buffer);
    break;
}
}

```

This is the output code for our qualifying expression “DEFAULT”. This is guaranteed to evaluate to true upon exit from the `START` state, unless a user desires otherwise and uses the `$seen` keyword to explicitly move to a different state. Our first matched pattern (`$1`) is accessed via the variable `NEST_func0PattMatch0PCRE`. This character pointer holds the matched input corresponding to the “ANY” pattern. One problem is that we cannot check the user-supplied code for possible buffer overflows or other possible security concerns. Therefore, the NEST programmer still needs to be vigilant when writing their own action code.

```

%%

NEST_Evaluate:
    int NEST_longestMatchingFunc = -1;
    int NEST_matchedLength = -1;
    if (NEST_DEFAULT) {
        int NEST_totalLength = 0;
        NEST_func0ExecPCRE = pcre_exec (NEST_func0Pattern0PCRE,
            NULL, NEST_buffer+NEST_totalLength,
            (int)strlen(NEST_buffer+NEST_totalLength),
            0, 0, NEST_ovectorPCRE, OVECMAx);

        if (NEST_func0ExecPCRE < 0) {
            //Error
        }
        ...
    }

```

In this section of code we evaluate our state machine and match the patterns associated with each state. If the NEST_DEFAULT boolean expression is true, then we match the patterns associated with that state of the input specification. In a larger specification this boolean expression may be very large, using several state names plus the &&, ! and || operators.

There is some pattern matching code that has been omitted for brevity. Suffice it to say that each pattern following a valid qualifying expression is matched against the server input in sequence. We save the matched portions of input into the appropriate variables.

```

//This is the end of code for the input specification

    if (NEST_totalLength > NEST_matchedLength) {
        NEST_longestMatchingFunc = 0;
        NEST_matchedLength = NEST_totalLength;
    }
}
if (NEST_longestMatchingFunc != -1) {
    NEST_switch = NEST_longestMatchingFunc;
    goto NEST_FuncStart;
}
goto EXIT;

```

If the `NEST_longestMatchingFunc` variable ends up being equal to -1 at the end of the evaluation, then we fall through to the `EXIT` label. This `EXIT` label forces entry into the `END` state so that error processing may take place. If, however, there is a `NEST_longestMatchingFunc`, then the code jumps back to the `NEST_FuncStart` label to actually run the code associated with the valid qualifying expression/patterns.

This gives us a basic overview of the output that `NEST` produces for an extremely simple server. In the next two sections we will show some of the thread-specific and event-specific code that is produced by `NEST`.

4.2 Threaded Server

The monolithic function code for the threaded server model is similar to our process-based code. There are only a few changes including the “`pthread_cleanup_pop()`” function, which is used to make sure that each thread exits properly if the user exits from within their action code. This problem with returning properly from within any action code led to the late addition of our `NEST`-specific `$return` statement for each server model. This `$return` statement must be used within action code to have

NEST servers run properly. The “`pthread_cleanup_pop()`” mechanism is also used if there are errors or exceptions in the action code that are not caught properly by the NEST programmer when a thread illegally exits. In some cases this mechanism can keep the server from crashing if a thread behaves badly.

Much of the extra code needed for the threaded model is contained in a separate file. This code includes some optimization code, including a thread pool. A thread pool is a collection of preallocated threads that can be used without thread startup overhead during heavy server load. These threads can be used repeatedly because they are just grabbing work from a queue, running until that work is complete and then checking the queue for more work. The maximum size of the thread pool is adjusted inside the NEST specification by defining the maximum number of clients that can connect to the server at any one time. We found that there is a relatively low default threading limit of 256 under Linux-2.4.22 with the LinuxThreads implementation and the default 8MB thread stack size. Thus, our threaded implementation has a thread pool size limit of 256.

We believe that NEST could be modified relatively easily to run with the Native POSIX Threading Library (NPTL). This library can purportedly run upwards of 100,000 threads at any one time, which may be advantageous for use in some types of servers[17]. The fact that we should be able to easily modify our tool to automatically output new threading code is one of the large advantages of this tool (i.e., we modify the tool, and programmers just recompile their servers without having to modify every line of threading code themselves). There are some other good ideas for network server optimization, like stack-ripping[3] or Capriccio[45], that may be incorporated into NEST without too much trouble. These two modifications may be presented to

the user as new server model types, thus expanding our tool’s abilities and usefulness.

One thread-specific portion of code that is generated is the code to update the server’s total client count.

```
//There is no code needed in the input specification to generate
//the following thread specific code

void NEST_cleanClients(void *arg) {
    NEST_lock();
    NEST_totalClients--;
    NEST_unlock();
}
```

Here we are using a mutex locking mechanism to gain exclusive access to the NEST_totalClients variable (and other global variables). We decrement the variable, unlock the mutex (in a separate function) and return from this function. This function is also used by the “pthread_cleanup_push()” function so that it can be used to decrement the total number of connected clients by the “pthread_cleanup_pop()” function. Our tool cannot guarantee proper global variable locking for user-defined variables, unfortunately. It is still the NEST programmer’s responsibility to lock global variables properly when using the threaded model.

```
//This code is generated for each exit from an action code block
//for thread-specific code

bzero (&NEST_buffer, sizeof (NEST_buffer));
NEST_n = NEST_dollarRead (NEST_sd, NEST_buffer);
if (NEST_n == 0)
{
    close(NEST_sd);
    cleanClients(NEST_ac);
    return;
}
```

In this code snippet we are performing a read on a client socket and checking for

a remote connection closure. If the socket is closed, we clean up and return out of the thread so that the thread can return to the thread pool to wait for more work. If the connection has not been closed remotely, then we have some information in our buffer that can be used to match against any user-defined patterns.

There are many other differences in the boiler-plate code contained in our extra code file but the code excerpts contained here (and comments at the start of this section) are good examples of the differences in the automatically generated code.

4.3 Event-driven Server

The event-driven server code was the hardest to generate automatically. We use non-blocking network input/output as an optimization mechanism in our event-driven model. This choice was largely influenced by the design of Matt Welsh’s Staged Event Driven Architecture[46]. Though we don’t have “Stages” in our model, as in SEDA, we still thought that non-blocking I/O would be a viable and useful optimization for our tool. Another small optimization is the preallocation of event structures, and the reuse of these structures when clients disconnect from our servers, to avoid allocation overhead. The number of structures is the same as the user-defined maximum number of client connections allowed to the server. Much of the code for the event-driven model optimizations is contained in our extra code file and is not generated, as it doesn’t change across event-driven NEST server implementations.

The next code snippet shows some differences, and similarities, between the `NEST_MainLoop` function for an event-driven server and the previous code shown for a process-based server.

```

%%

NEST_MainLoop (int NEST_dummy, short NEST_event, void *NEST_arg) {
    NEST_sd = ((NEST_conn *) NEST_arg)->sfd;
    NEST_conn *NEST_c = (NEST_conn *) NEST_arg;
    NEST_c_addr = (NEST_conn *) NEST_arg;
    NEST_currentInMem = NEST_sd;
    NEST_c->NEST_START = true;
    NEST_c->NEST_END = true;
    #define OVECMAX 9
    int NEST_ovectorPCRE[OVECMAX];
    char *NEST_subjectPCRE;
    int NEST_lastSeen = -1;
    bool NEST_DEFAULT = 0;
    ...
}

```

The main differences in this event-driven code are the arguments passed to the `NEST_MainLoop` function and the event structure initialization code at the start of this function. The arguments passed into this function include a dummy variable called `NEST_dummy` that is needed by `libevent`, the `NEST_event` variable that is also needed by `libevent` and our event structure that is pointed to by `NEST_arg`. Some variables like `NEST_START` and `NEST_END` are still the same as in the process-based model.

In this section of code we also make two copies of the pointer to our event structure for use later in our code. We also set the `NEST_currentInMem` variable to the socket descriptor associated with this event structure. This gives us a unique identifier that lets us keep track of the event structure that is currently in memory. We use our `NEST_currentInMem` variable in later sections of code to decide whether an event structure needs to be swapped out, or not. After these initialization steps, our code starts to look similar to that generated for a process-based server.

4.3.1 Non-blocking I/O

Since we are using non-blocking network I/O for our event-driven server we need to have a way to jump out of our monolithic function, where code may block on network I/O or user input, and then return directly to that same spot once execution resumes. This led to the need for us to keep extra information in our event structures, and the need for some extra branching code to be generated inside the monolithic function. There is also a lot of event-driven code contained in its own separate file, not in the extra code file or the monolithic function. The code in the event-driven specific file is not compiled into a process-based or threaded server.

```
NEST_MainLoopStart:
  switch (NEST_conn->started)
  {
    case 1:
      goto NEST_FuncStart_Switch1;
    case 2:
      goto NEST_FuncStart_Switch2;
    ...
  }

  ...
NEST_FuncStart_Switch1:
NEST_c->started = 1;

NEST_dollarRead...

NEST_FuncStart_Switch2:
NEST_conn->started = 2;
goto NEST_Evaluate;
```

This code shows a new “switch{ }” statement that facilitates the branching to the appropriate spots in our NEST_MainLoop function. One important thing to notice here is that we have one label before our network read and one after. This means that we can block on the read, jump out of this code to service a different event, and

then branch back into this code after we receive an event telling us that the network read has finished. We have to branch back into this code *after* the network read statement, and this is one solution we came up with to solve this non-blocking I/O specific problem in our generated code.

Next we will present an example of some code for the event-driven write operation.

```
//There is no code needed in the input specification to generate
//the following event-specific code
```

```
NEST_c->started = 3;
NEST_FuncStart_Switch3:
NEST_outResult = NEST_fileReply (NEST_sd, NEST_tmp);
NEST_c->write_bytes = NEST_c->write_size;
NEST_c->started = 4;
NEST_FuncStart_Switch4:
    if (NEST_outResult == -1) {
        int NEST_inResult;
        for(;;) {
            NEST_inResult=write(NEST_c->sfd,
                NEST_c->write_curr,NEST_c->write_bytes);
            if (NEST_inResult > 0) {
                NEST_c->write_curr += NEST_inResult;
                NEST_c->write_bytes -= NEST_inResult;
                if (NEST_c->write_bytes > 0) {
                    break;
                }
            }
            else {
                NEST_totalClients--;
                NEST_c->op = NEST_closing;
                NEST_close(NEST_c);
                return;
            }
        }
    }
    else if (NEST_inResult == 0) {
        NEST_totalClients--;
        NEST_c->op = NEST_closing;
        NEST_close(NEST_c);
        return;
    }
    else if (NEST_inResult == -1
        && (errno == EAGAIN || errno == EWOULDBLOCK)) {
        if (!update_event (NEST_c, EV_WRITE | EV_PERSIST)) {
            NEST_totalClients--;
            NEST_c->op = NEST_closing;
            return;
        }
        break;
    }
}
...
}
```

This code shows our “fileReply()” function, which is a precursor function to the socket write code we display here. In our “fileReply()” function we set some variables in the appropriate event structure and copy the file we are going to write to the socket

into a buffer. We then start the write operation to the socket inside the “fileReply()” function. If this write operation is going to block then we allow our event structure to be swapped out, only to continue the write operation when this event structure gets to the head of the event wait queue.

Once our write operation starts again, at `NEST_FuncStart_Switch4` in this case, our code goes into an infinite “for{ }” loop. Inside this infinite loop we attempt to write the remainder of the file to the specified socket. Several actions can get us out of the infinite loop, including a client-side connection closure, an incomplete write operation or a possible blocking operation. If this write operation is going to block again then we simply “update_event()”, which places our event at the end of the event queue, and break out of the infinite loop. We may also break out of this loop if we are indeed finished the write operation. After we break out of the infinite loop we evaluate the status of our write operation and either continue on with the program if the write is complete, or eventually re-enter our write code to finish the file write.

4.3.2 File-globals

A major dilemma we faced when automatically generating code for our event-driven servers was NEST programmers using file-global variables. Any variables declared after the first “%%” sign in the NEST specification, and before any qualifying expressions, are file-global and thus may be accessed from within any other part of the remainder of the specification file, including the important action code sections.

The reason we had to come up with a solution to file-global variables being accessed in the action code is illustrated with the following example:

- Client #1 accesses a file-global integer called “X” and sets it to seven ($X = 7$),

then client #1 blocks on I/O.

- The event structure for client #1 will be swapped out, making room for client #2.
- Next, client #2 accesses the file-global variable “X” and changes it to the value three ($X = 3$).
- Then client #2 gets swapped out, without any other changes to “X”, and client #1 is swapped back in because its I/O operation is complete.
- Client #1 prints out “X” only to find that it is now equal to three and not the expected value of seven.

Since a NEST programmer can access “X” from anywhere in the action code we would need to parse the action code and replace any instance of “X” with the client’s own event structure specific variable (e.g., `NEST_c->X = 3`). We didn’t want to write a complete C parser to accommodate this, so we came up with another solution.

The next code excerpt will illustrate part of our solution to this file-global access problem.

```
%%
char buffer[256];

char NEST_StartBuff[PAGESIZE];
char NEST_bss1;
char NEST_dataSeg1 = '1';

char buffer[256];

char NEST_bss2;
char NEST_dataSeg2 = '1';
char NEST_EndBuff[PAGESIZE];
```

Here we surround the file-global variable “char buffer[256]” with two known marker values, in this case NEST_bss1 and NEST_bss2 for non-initialized variables. The NEST_dataSeg variables are used for initialized variables, as Linux puts these two variable types in different memory segments in the compiled executable server. We do some distance calculations using these variables to determine how large our event structure specific, file-global data holding area should be. Then we simply copy the 256 raw bytes between the markers that represent “buffer” to our designated event structure data holding area, along with the rest of the event structure. A copy happens in the other direction when we swap the event structure back into memory. (The NEST_StartBuff and NEST_EndBuff variables will be explained below.)

We have two versions of our file-global variable swapping solution. One is an eager swapping solution and the other is a lazy solution. The eager solution will swap out an event structure every time there is the possibility of blocking in our server (i.e., when the non-blocking I/O mechanism kicks in to save the day). This means the entire structure is swapped, including the raw data bytes representing the file-global variables. The lazy solution to our problem only swaps the event structure if there is a section of action code that accesses one of the file-global variables. The way that we determined that a file-global variable was being accessed in the action code is by using Linux memory protection. This code snippet shows the protection mechanism.

```

%%
char buffer[256];

#ifdef LAZY
    struct sigaction sa;
    sa.sa_sigaction = NEST_segv_handler;
    sa.sa_flags = SA_SIGINFO;
    sigemptyset (&sa.sa_mask);
    sigaction (SIGSEGV, &sa, NULL);
    NEST_area = (unsigned long) ((unsigned long) &bss1 & 0xfffff000);
    if ((mprotect ((void *) NEST_area, PAGE_SIZE * NEST_pages, PROT_NONE)))
    {
        perror ("mprotect");
        exit (1);
    }
#endif

```

In English, if we are using lazy evaluation we set up a new segmentation violation handler to replace the default supplied with Linux. We then protect an area of memory using “mprotect()” so that if it is accessed by user written action code, a segmentation violation is triggered. The variables `NEST_StartBuff` and `NEST_EndBuff`, shown in the previous code snippet, are used to calculate the size of `NEST_area` (`NEST_dataStartBuff` and `NEST_dataEndBuff` are not shown for the data segment calculations). In Linux the area being protected needs to fall on a virtual memory page boundary, and it needs to be at least one page long. There were some problems with these requirements that we will discuss below.

We ran into a problem when protecting the memory pages and we needed the one-page long `NEST_StartBuff` and `NEST_EndBuff` variables to solve the problem. What we discovered was that if a user was to define file-global variables that span, for example, 1.2 pages of memory (approximately 4916 bytes), and their server’s bss area of memory starts four bytes *before* a page boundary, then we have a major problem. This scenario, shown in Figure 4.1, would mean that our `NEST_area`

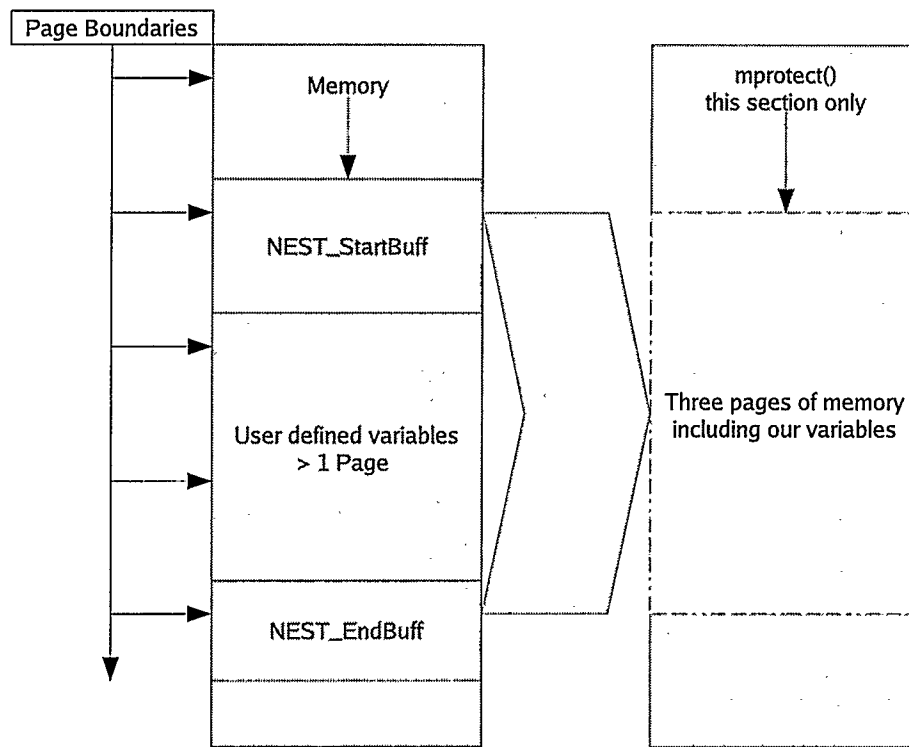


Figure 4.1: Protecting Memory Pages

calculation would return two pages, because we round up, when we would actually need to protect three pages from being accessed via “`mprotect()`”. We need three pages since the first page has four bytes that need protecting, the entire second page needs protecting and the first 816 bytes (this is equal to $4916 - 4096 - 4$) of the third page would need protection. That is three total pages that need protecting. The `NEST_Start` and `End` buffers reserve enough space so that we end up protecting the memory pages that begin and end on the correct boundary.

We could have gotten the same effect as above by forcing the gcc compiler to keep the file-global variables starting on a page boundary. We could accomplish this

by using the “`-start-section .ourBssSection=PAGE.BOUNDARY`” option with the GNU linker (and similarly for `.ourDataSection`). This would mean that we would always have our first declared file-global variable on a page boundary. We would then have to guarantee that our file-global variables took up exactly X pages that would subsequently be “`mprotected()`” by our tool. This is largely the same thing we have done with our solution above, but our solution was slightly easier to control in the resulting executable. Manually placing sections of data, using the “`-start-section`” method, can also lead to problems if done incorrectly when linking, which we don’t have to worry about in our solution.

As a small optimization, the eager and lazy solutions actually check to make sure that the old event structure in memory is different from the new incoming structure before any swap is made. In other words, if we swap out an event structure to avoid blocking on I/O, and that same event structure is going to be swapped immediately back in because it finished its blocking operation before any other event, then we will not swap the new structure in. We will just use the event structure already sitting in memory. This process could be optimized further by reordering the event queue to maximize consecutive events for a connection, thus avoiding event structure switches. We would need to do this optimization while preserving fairness to all clients, if possible.

4.4 Common code

Each type of server has its own “`main()`” function startup code. This startup code just sets up the server according to the chosen model. For instance, the process-based

model sets up some signal handlers and gets ready to accept incoming connections. When a client connection is made, the server forks off a new process to service the client. The threaded model does the same sort of setup for incoming connections but it also initializes the thread pool and it adds new work to the thread pool queue when a client connects. The event server model runs in a single process and its incoming connections are accepted after being taken off the event queue.

4.5 Summary

In this section we have presented some of the work that went into implementing NEST. We have shown some pertinent example code that corresponds to the NEST specification language, along with some explanations of the code. Each of the three models is covered in detail.

Chapter 5

Evaluation

Evaluating tools like NEST may be broken into two parts: the development time saved by using this kind of tool, and the resulting performance of the generated servers.

5.1 Development Effort

The savings in development effort that can be shown with this tool are immediately apparent. We can change one line in the NEST specification, recompile and test out a new server model in seconds, literally, whereas manually rewriting the server code would take days or weeks. This kind of savings in development effort is extreme and self-explanatory.

One measure of the development effort is the number of lines of code (LOC) written to produce any given server. Since we didn't implement full-featured servers to compare our generated code against, we do not compare lines of code. Instead, we consider some features of the specification language:

- The use of regular expressions to easily match input from a client cuts down on development effort. Though this type of regular expression matching can be done manually by a programmer, the substitution and automatic regular expression compilation done by our tool will save time and effort, overall.
- NEST can be used to easily set up a state machine. The use of substitutions,

boolean expressions using these substitutions and an intuitive state naming mechanism (with qualifying expressions) can be used to easily produce state machines.

Development effort can also be reduced, when using a tool such as NEST, by avoiding or reducing some common programming errors. Code generation tools will produce correct code (assuming proper debugging of the tools output) according to the input specification. Of course, if there is a logical or coding error in the tools input specification then the tool will not produce correct output. The tool may produce no output at all, until the errors are corrected; this depends on the tool in use. NEST helps reduce programming errors by automatically generating correct networking code for all three server models. Since many programmers have a preferred model for their networking applications, NEST may save a great many errors when a programmer attempts to switch models.

5.2 Performance

Though our original design goal with this project was to focus on saving programmer effort by generating all of the networking code automatically, we realize that we should also have server code that is moderately efficient so that the tool may be shown to be practical.

In our experience, we have found that performance for network servers is primarily compared using HTTP servers. This is because web servers can experience severe spikes in demand for static as well as dynamic content. Other servers, like FTP servers, may have high throughput demand for file downloads but they usually don't

have a lot of user interaction and a need to push out many small files for viewing by users.

When comparing our example HTTP-like server to any other HTTP servers we need to realize that most HTTP servers have been very highly optimized. Our modest attempts at optimization are more general and are used to prove that many different types of optimization can be incorporated into our tool. Our servers do perform competitively well in our tests.

The setup for all test results in this section is as follows: four computers were used for testing, with a single switch between them. The computers are Xeon 1.4GHz, with 512MB RAM, 100 Mb/s Ethernet cards, running the Linux 2.4.22 kernel. Though these machines were not on a dedicated network, all tests were performed during off hours and the machines were checked before each test to make sure that no one else was using them. The number of processes running on the server and clients was minimized as much as possible.

Network latency from each client, through a switch, to the server is always an issue during the testing of servers. Some of the researchers cited in this thesis[45, 46] avoid latency by removing the network altogether. The authors test their servers using clients on the same machine. While this removes the network latency issues from testing, it also produces contrived results that may be unrealistic when a server is deployed in the networked environment. We have tested our servers over a network with the belief that the results we get will reflect a real networked environment more accurately. In Section 5.3 the threaded server model can be seen to service many clients, even at very high connection rates, without any errors. This suggests that the actual server processing is the bottleneck for other server models, and not the

network itself, though some small percentage of errors may arise from the network.

5.2.1 httpperf

The tool that we have chosen for our tests is httpperf-0.8[34]. This tool is a Hewlett-Packard Research Laboratory tool developed by several authors. It is used with HTTP 1.0 or 1.1 to test a number of performance metrics. It can be used with several client machines to request a large amount of web pages from a server. Each client keeps track of many statistics about the server's performance. The httpperf tool has many options to help tweak tests for differing server and networking environments. There are many other testing tools available, but we decided to use this simple command line tool for testing our servers, rather than larger distributed, or commercial systems.

We use httpperf to perform a large number of GET requests from three client machines to our server machine. All tests had our client machines requesting 32,000 web pages per client, each consisting of 2KB of data. We have kept the file transfer size small since our NEST servers do not use any file caching mechanisms for large files, or similar optimizations.

5.2.2 Performance Graphs

Our HTTP server, for all three model types, is a simple server that can accept GET, PUT and POST operations. It does not accept CGI requests, support SSL/TLS connections, or serve any dynamic content. The aim here is to show acceptable performance of our automatically-generated code sections. If we tested very complex web server features, then we would be testing more and more hand coded actions

and less of our automatically-generated code.

The graphs in this section represent performance statistics for several different server types. Each of the three server models that we generated were compared to two other open-source servers that use the same model. All nine servers in our tests used the following `httpperf` command line:

```
httpperf --hog --timeout=5 --client=0/1 --server=ict618a
--port=8080 --uri=/www/lex.ll --rate=??? --send-buffer=4096
--recv-buffer=16384 --num-conns=32000 --num-calls=1
```

The server rate option is the only variable. This variable changes from 200 connections per second to 400 and finally to 600 connections per second, for our tests. These connection rates were rates per client, so that the servers were actually being bombarded with 600, 1200 and 1800 connections per second (recall that we used four machines, making three clients and one server). The `lex.ll` file that is being requested is exactly 2KB long.

Our results in this section show the connection time for clients as well as the transfer time. The connection time is the amount of time it takes for the client to “connect”, or receive an initial answer back from the server. The transfer time is the amount of time that it takes the client to receive the entire file that was requested. In Figure 5.1 the connection time and transfer time are separated by a line, with the connection time under the line and the transfer time over the line. Together these two rates would appear to a web user as the time to view a complete web page download. All of our results in this section are averages over the three clients connecting to the

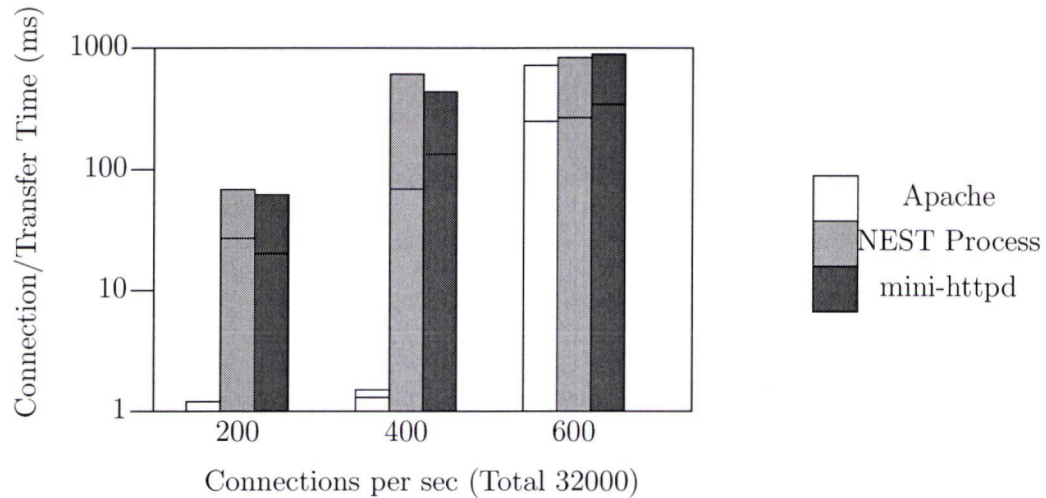


Figure 5.1: Benchmark results for process-based servers (Lower is better)

server. These averaged results were then averaged a second time over three separate httpperf runs per server.

Process-based Server Performance

The first tests that we performed were on our process-based server and two other process-based open-source HTTP servers, all written in C. One of these open-source servers is the Apache Web server, which is the *de facto* Internet standard for HTTP servers. The Apache server is the most popular and widely used Web server on the Internet[6]. For these tests we used Apache version 2.0.52. The other process-based server we tested is the mini_httpd-1.19 server[36]. This is a simple “mini” server, which means that it is not full-featured and according to the author it was developed to see “just how slow an old-fashioned forking web server would be with today’s operating systems”[36]. The author claims that mini_httpd server runs at about 90% the speed of Apache on FreeBSD 3.2. There is no evidence given to back

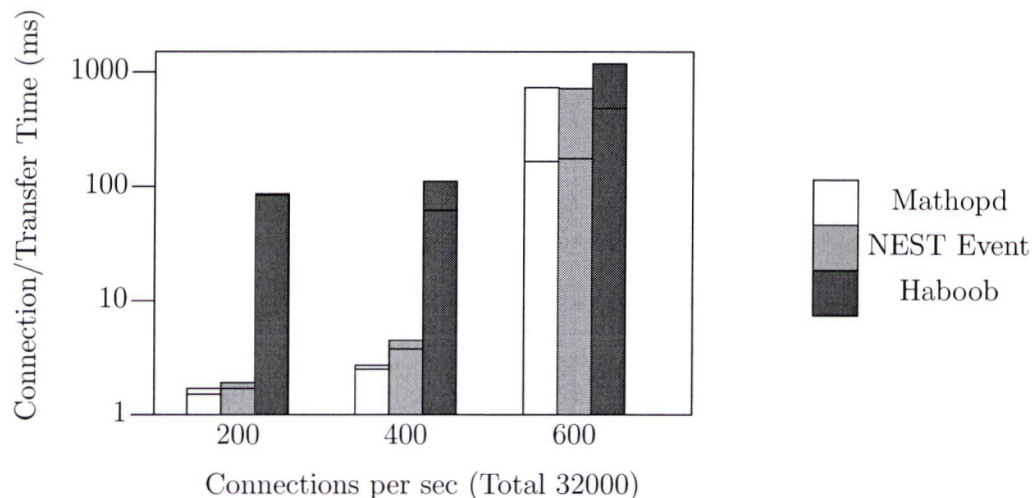


Figure 5.2: Benchmark results event-driven servers (Lower is better)

up these performance claims, however.

Our results in Figure 5.1 are for our process-based NEST server, `mini_httpd` and Apache. They show that the Apache Web server is quite a lot faster, especially at low connection rates. We believe this difference is due to Apache optimizations, such as pre-forking, and the fact that `mini_httpd` and our NEST server are not optimized. We can see, in fact, that our process-based server is quite comparable to the `mini_httpd` server on these simple test runs.

Event-Driven Server Performance

Our event-driven performance tests were done on the NEST-generated event server, `mathopd-1.5p4` which is a “very small, very fast HTTP server for UN*X systems” [12], and the SEDA-based Haboob[46] Web server, release date 2002/07/12. Haboob and `mathopd` are both open-source Web servers, with `mathopd` being written in C and Haboob in Java. We have included the Haboob server in our tests, even though it is

written in Java, due to the fact that it has been touted as a “high-performance HTTP server” [46], with previous results in the cited thesis showing increased throughput when compared to Apache. The Haboob server is a full-featured Web server while mathopd supports php and CGI scripting. We have used an unmodified version of Haboob, which is to say that we did not change any optimization settings for the Sandstorm service platform that Haboob is built on. We have not included Apache in these tests as there is no event-driven version we are aware of.

Our results in Figure 5.2 show that Haboob appears to have some connection overhead that is quite significant at the lower connection rates, though the actual transfer rate at 600 cconnections per second is the best rate of the three servers. Our NEST generated server performs well when compared to the hand-coded mathopd server. Our server is only slightly slower at the lower connection rates, while performing on par with the other two servers at the highest connection rate. The connection overhead that is apparent with the Haboob server may be due to the context switching related to its stages, and its large amount of queueing operations as cited in the work done on Cappricio[45].

Threaded Server Performance

The threaded servers we tested were our NEST generated thread server, the threaded version of Apache-2.0.52 and the Adaptive Communications Environment (ACE)[40] based JAWS threaded server. The JAWS server is written in C++ and built on frameworks supplied by ACE. It is also a full-featured server with complex internal strategies and optimizations for dealing with severe server load conditions, including a cached virtual filesystem. The JAWS server we tested was last changed Aug 26,

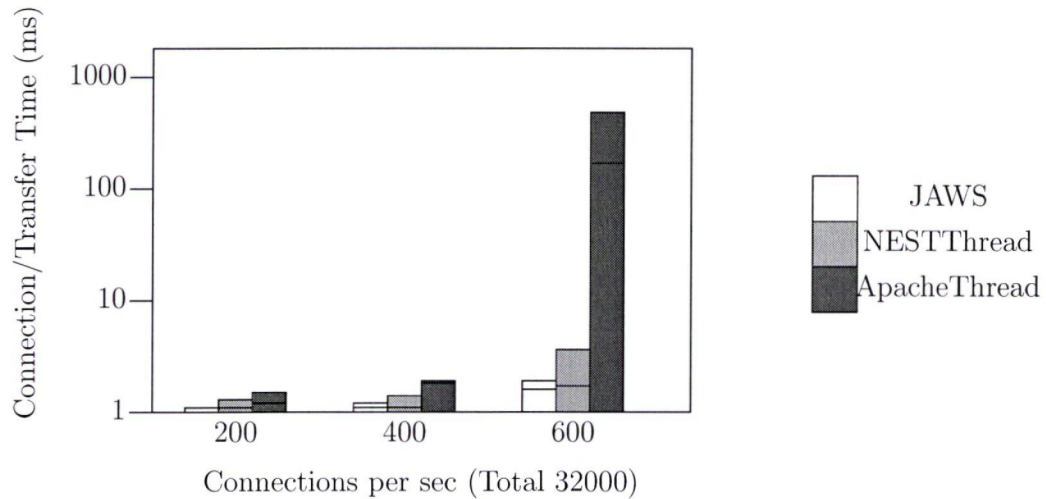


Figure 5.3: Benchmark results for threaded servers (Lower is better)

2004.

The results in Figure 5.3 are more than a little surprising. We are not sure why Apache performs so poorly at 600 connections per second compared to our NEST generated server and the JAWS web server. Please notice that the Y axis of these graphs is logarithmic, which means that Apache is significantly slower for this model. However, these results for Apache closely resemble those for the process-based model; they are only slightly faster. The results for the other two threaded servers are exceptional compared to the other server models. For our server we suspect that this is due to our thread pooling optimization and the Linux operating system efficiently using these threads. The JAWS Web server has many optimizations that lead to performance that almost doubles that of our server. Though we do not explicitly cache files, as JAWS does, we are not traversing a directory structure or requesting many files during our tests. We surmise that JAWS would perform much better than

our threaded server under these conditions.

The results for these servers were so surprising that we actually ran our tests several dozen times to make sure we were not seeing some sort of anomalies in the testing procedure, or the `httperf` tool. This includes testing Apache many times, with similar results.

This type of result is one of the main reasons that NEST has been developed. Under these testing conditions, with an HTTP server receiving many requests for small files, a threaded model would appear to be the best server model choice. We had previously assumed that an event-driven model would perform comparably, which means that if we had only written a server using the event-driven model we may have wasted some time rewriting the server, using a threaded model, to increase its performance.

5.3 Error Rates

When we measured error rates with `httperf`, each of the servers acted in a similar manner, which is to say that error rates increased rapidly when errors first started to appear on the client side. There is also a correlation between error rates and the server's performance. This is reasonable, since the servers would start to have more client time-outs as server load increased past a threshold where the server became overloaded and couldn't transfer the complete file prior to the time-out. All of the clients received many client time-outs and "connection refused" errors when the connection rates were extremely high, which led to the large number of errors at the 600 client connections per second rate for each server model.

Server	Connections Per Second		
	200	400	600
Apache	0	0	6054
NEST_Process	30	165	5953
mini_httpd	55	1841	7073

Table 5.1: Process-based Server Error Rates

Server	Connections Per Second		
	200	400	600
Mathopd	0	0	3522
NEST_Event	0	0	4093
Haboob	0	55	4824

Table 5.2: Event-driven Server Error Rates

The process-based results in Table 5.1 show that the Apache server had no errors until a threshold was reached between 400 and 600 connections per second, then there were plenty of errors. The NEST-generated server and the mini_httpd server are not optimized and showed errors even at the lower connection rates. Error rates are similar at the highest rates because the servers simply become overloaded.

In Table 5.2 we can see similar behavior to the process-based servers, except at the lower connection rates. These servers have no errors at the lowest connection rate, which corresponds to the improved connection and transfer times seen in Figure 5.2. As with the process-based servers, each event-driven server hits a threshold and then error rates increase rapidly. The threshold for these event-driven servers appears to be between 400 and 600 client connections per second, much like the process-based Apache server.

The threaded server error rates represented in Table 5.3 have almost no errors, which correspond to the exceptional performance times for these servers (except Apache at 600 connections per second). These error rates for the NEST threaded

Server	Connections Per Second		
	200	400	600
JAWS	0	0	3
NEST Thread	0	0	13
Apache Thread	0	0	5494

Table 5.3: Threaded Server Error Rates

server and Jaws server indicate that we are primarily testing server performance and network latency is not the primary reason for increasing error rates. We did run tests at higher rates on our NEST threaded server and the JAWS server, just to see if their error rates increased in line with all the other servers. This was the case as they started showing error rates similar to all the other servers: only at a proportionally higher connection rate of 800 connections per second.

5.4 Summary

In this chapter we concentrated on a performance evaluation of the Web server code that NEST generates. We have shown that NEST performs comparably to several industrial strength Web servers and outperforms some other servers. We have also argued that NEST can save time and minimize some errors in server development.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis we have presented a comprehensive overview of our network server tool, NEST.

We have shown that our tool can be used to quickly generate C code that implements three different server models: process-based, threaded, or event-driven. NEST has a simple input language that is similar to other domain-specific tools and generation of these three model types is accomplished by simply changing one line of code in the NEST input specification.

We have also shown that NEST can be used to rapidly prototype high-level protocols and test them using each model type. This can lead to improved server performance since a programmer can choose the most appropriate model for the protocol they are prototyping.

NEST can reduce bugs and security holes related to server communication code due to the fact that much of the buffer handling is automated. Low-level networking details are abstracted away by the use of our network server tool which means that programmers can concentrate on the protocol specification and not have to “reinvent the wheel” for each server model.

Our testing has shown that NEST-generated servers can perform competitively when compared to hand-written servers of each model type. Optimizations contained

in the generated server output, including thread pools and asynchronous I/O, help our NEST-generated servers perform on par with other optimized high-performance servers.

Other areas, like compiler construction, make extensive use of software tools, and the potential uses for tools in networking has only begun to be explored.

6.2 Future Work

We believe that NEST can be made more robust, as well as being able to produce code that performs as well as many hand-written servers. The process-based code should be optimized with pre-forking to be useful in a less restricted environment. More optimizations should be investigated for all server models, especially file caching mechanisms. New models similar to a Cappricio-like model and a stack-ripping model could be added to quickly compare these newer model types to the already established models that NEST supports.

Some work that may increase the speed of NEST would involve using Lex instead of libpcre for pattern matching. Subsequent code profiling of each solution would indicate which method is faster.

The actual amount of time that is saved by using NEST to generate a server is an area of some interest. It seems like a huge amount of effort and time are saved by using NEST, but there may be subtle, time consuming, and as yet unforeseen, problems that crop up with fully functional servers. Some programmers could find it very difficult to fine tune and modify NEST output code to their exact specifications, thus reducing the advantage of automated output, to some degree.

Appendix A

The NEST grammar

```
start      := INIT_TOOL def_section PERCENT_PERCENT rules PERCENT_PERCENT
def_section := def_section SUB NAME |
              def_section PERCENT_STATE name_list |
              def_section PERCENT_OPTION NAME VALUE |
              epsilon
rules      := rule rules |
              epsilon
rule       := LOCAL START_RULE action |
              END_RULES action |
              condition ':' pattern action |
              condition ':' multistate |
              pattern action
multistate := BRACE rules C_BRACE
action     := BRACE code C_BRACE
pattern    := S_NAME pattern |
              REGEX pattern |
              epsilon
condition  := boolean |
              STR_LIT
boolean    := and_bool |
              or_bool |
              not_bool |
              last_bool |
```

```

        PAREN boolean C_PAREN |
        S_NAME
and_bool    := boolean AND_OP boolean
or_bool     := boolean OR_OP boolean
not_bool    := NOT_OP boolean
last_bool   := LAST_SEEN enclosed boolean |
code        := code SEEN enclosed |
             code CLEAR enclosed |
             code SUB_FOR_PATTERN |
             code ESCAPE_D |
             code REPLY OPTIONS |
             code FILE_REPLY OPTIONS |
             code FILE_RECIEVE OPTIONS |
             code RETURN |
             epsilon
enclosed    := PAREN S_NAME C_PAREN
name_list   := name_list S_NAME |
             epsilon

```

References

- [1] Androutsellis-Theotokis, S. and D. Spinellis, A Survey of Peer-to-peer Content Distribution Technologies. *ACM Computing Surveys* 36, 4, pp. 335-371 (2004)
- [2] Abbott, M.B., and L.L. Peterson, A Language Based Approach to Protocol Implementation, *IEEE/ACM Transactions on Networking*, pp. 4-19 (1993)
- [3] Adya, A., J. Howell, M. Theimer, W. Bolosky and J. Douceur, Cooperative Task Management without Manual Stack Management or, Event-driven Programming is Not the Opposite of Threaded Programming, Microsoft Research, (2003)
- [4] Aleph One, Smashing the Stack for Fun and Profit, *Phrack* 7(49), (1996)
- [5] Andersen, H.R. An Introduction to Binary Decision Diagrams, Course notes for C4330 E96, Department of Computer Science, Technical University of Denmark, (1996)
- [6] Apache, HTTP Server Project, <http://httpd.apache.org>
- [7] Basu, A., G. Morriset and T. Von Eicken, Promella++: A Language for Constructing Correct and Efficient Protocols, *IEEE Infocom - The Conference On Computer Communications*, pp. 455-462 (1998)
- [8] BEA, Programming a Distributed Application: The BEA Tuxedo Approach, White Paper. <http://www.bea.com>
- [9] Berry, G., and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics; Implementation. *Science of Computer Programming*, 19, 2, pp. 87-152 (1992)
- [10] Bhatkar, S., D. DuVarney and R. Sekar, Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits, 12th USENIX Security Symposium, pp. 105-120 (2003)
- [11] Birrel, A., and B. Nelson, "Implementing Remote Procedure Calls", XEROX CSL-83-7, (1983)

- [12] Boland, M., Mathopd <http://www.mathopd.org/>
- [13] Candea, G., S. Kawamoto, Y. Fujiki, G. Freidman, A. Fox, Microreboot - A Technique for Cheap Recovery, 6th Symposium on Operating Systems Design and Implementation, pp. 31-44 (2004)
- [14] Castelluccia, C., and W. Dabbous, HIPPCO: A High Performance Protocol Code Optimizer. INRIA Research Report No. 2748, (1995)
- [15] Dijkstra, E., Go To Statement Considered Harmful, Communications of the ACM, 11, 3, pp. 147-148 (1968)
- [16] Douglas, M., MSPL: A Protocol Language for Generating Client-Server Software, MS Thesis, Florida Tech (2000)
- [17] Drepper U., [Announce] Native Posix Threading Library 0.1, Linux Kernel Mailing List, <http://www.ussg.iu.edu/hypermail/linux/kernel/0209.2/1075.html> (2002)
- [18] Drepper, U., Security Enhancements in Red Hat Enterprise Linux (SELinux), (2004)
- [19] Free2code, <http://www.free2code.net/plugins/articles/read.php?id=336>
- [20] Gran, M., <http://lonelycactus.com/guilebook/book1.html>
- [21] Hazel, P., <http://www.pcre.org/>
- [22] IBM Corporation, <http://publib.boulder.ibm.com/infocenter/pseries/index.jsp>
- [23] Jahn, S., Serveez: <http://www.gnu.org/software/serveez/manual/index.html>
- [24] Johnson, S., YACC: Yet Another Compiler Compiler, CS TR 32, Bell Labs (1975)
- [25] Jones, D., How (Not) to Code a Finite State Machine, Association for Computing Machinery Special Interest Group on Programming Languages Notices, 23, 8, pp. 19-22 (1988)
- [26] Klog, The Frame Pointer Overwrite, Phrack 5(55), (1999)

- [27] Koenig, A., Language Design is Library Design, *Journal of Object-Oriented Programming*, (1991)
- [28] Ladd, D.A. and J.C. Ramming, Programming the Web: An Application-Oriented Language for Hypermedia Service Programming, *Fourth International WWW Conference*, pp. 567-586 (1995)
- [29] Lauer, H.C. and R.M. Needham, On the Duality of Operating System Structures, In the *Second International Symposium on Operating Systems*, IRIA, (1978)
- [30] Lefkowitz, G. and I. Schtull-Trauring, Network Programming for the Rest of Us, *USENIX 2003 Annual Technical Conference*, pp. 77-90 (2003)
- [31] Lesk, M. and E. Schmidt, Lex - A Lexical Analyzer Generator, *Computer Science Technical Report No. 39*, Bell Laboratories, (1975)
- [32] Levine, J., T. Mason and D. Brown, *Lex & Yacc*, Second Edition, O'Reilly (1992)
- [33] Lippman, S., and J. Lajoie, *C++ Primer*, 3rd edition, Addison-Wesley (1998)
- [34] Mosberger, D. and T. Jin. httpperf: A Tool for Measuring Web Server Performance, *Proceedings of the 1998 Internet Server Performance Workshop*, pp. 59-67 (1998)
- [35] Ousterhout, J., Why Threads are a Bad Idea (for most purposes), Powerpoint slide presentation,
<http://home.pacbell.net/ouster/threads.ppt> (1995)
- [36] Poskanzer, J., mini.httspd-small http server, <http://www.acme.com/software/mini.httspd/>
- [37] Provos, N., <http://www.monkey.org/~provos/libevent/>
- [38] Pu, S., H. Massalin and J. Ioannidis, The Synthesis Kernel, *Computing Systems*, 1, 1, pp. 11-32 (1998)
- [39] Rinard, M., C. Cadar, D. Dumitran, M. Roy, T. Leu and W.S. Beebe Jr., Enhancing Server Availability and Security Through Failure-Oblivious Computing, *6th Symposium on Operating Systems Design and Implementation*, pp. 303-316 (2004)

- [40] Schmidt, D., An Architectural Overview of the ACE Framework, A Case-study of Successful Cross-platform Systems Software Reuse, USENIX login magazine, Tools special issue, (1998)
- [41] Shah, M.A., S. Madden, M.J. Franklin and J.M. Hellerstein: Java support for data-intensive systems: Experiences building the Telegraph dataflow system. SIGMOD Record, 30, 4, pp. 103-114 (2001)
- [42] Tanenbaum, A., Computer Networks, 4th Edition, Prentice Hall (2003)
- [43] Twisted Matrix Labs, Twisted:
<http://www.twistedmatrix.com/index.html>
- [44] Van Renesse, R., K. Birman, and Silvano Maffei, Horus, A Flexible Group Communication System, Communications of the ACM, (1996)
- [45] Von Behren, R., J. Condit, and E. Brewer, Why Events are a Bad Idea (for high-concurrency servers), In Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS IX), (2003)
- [46] Welsh, M., An Architecture for Highly Concurrent, Well-Conditioned Internet Services, PhD Thesis, University of Berkeley, (2002)
- [47] Williams, R., Buffer-Overflow Attacks: Perimeter Defenses No Panacea,
<http://www.eweek.com/article2/0,1759,1563046,00.asp> (2003)
- [48] Wilson, K. and J. Aycock, NEST: NEtwork Server Tool. Technical Report 2004-746-11, Department of Computer Science, University of Calgary, (2004)
- [49] Wilson, K. and J. Aycock, NEST: NEtwork Server Tool. 5th International Conference on Internet Computing, poster, p. 700 (2004)
- [50] Wilson, K. and J. Aycock, NEST: NEtwork Server Tool. 11th Asia-Pacific Conference on Communications, to appear, (2005)

- [51] Zadok, E., FiST: A System for Stackable File System Code Generation. PhD thesis, Columbia University, (2001)