NEST: NEtwork Server Tool

Kelly Wilson and John Aycock
Department of Computer Science
University of Calgary
2500 University Drive N.W.
Calgary, Alberta, Canada T2N 1N4
Email: {wilsonk,aycock}@cpsc.ucalgary.ca

Computer Science TR 2004-746-11

***Abstract*— This paper presents a specification language approach to automatic server code generation. The language we present has some similarities to the compiler tools Lex and Yacc, and makes the generation of state machines and parsing of server input simple. One of the main features of this new language is that it can generate three different types of servers: process-based, threaded, and event-driven. In addition, use of a server generation tool can improve programming productivity, abstract away unnecessary details, and eliminate certain classes of error.**

***Index Terms*— Servers, software tools, threaded servers, event-driven servers.**

I. Introduction

Writing code for network servers is both tedious and error prone. In this paper we present NEST, a new tool for automatically generating most of the communications code infrastructure that TCP-based servers require. This offers several advantages:

- 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.
- Reduced bugs and security problems.

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

Using our network server 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, so we have given programmers the ability to choose between different server types. Some previous work suggests there are places where event-driven servers [4] are desirable; other work suggests general difficulties with threaded code [10]. 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. NEST does not impose a server model, giving maximum flexibility.

Some common errors and poor programming practices in Internet applications can be avoided or contained when using a tool such as NEST, like the dreaded buffer overflow [1]. This could lead to code with fewer security holes per application and less programming errors in general than when manually writing server code. Arguably, the prevalence of tool-generated code could also introduce a single commonly exploitable error into hundreds of different types of servers. While this is possible, tool-generated code need only be security audited once, without having to inspect hundreds of different implementations. There are also new techniques [11] being explored that could be used to effectively mitigate the effects of these common errors. The trade-off between time saved versus security problems introduced by this type of tool is an area of future study.

One real benefit of this tool is in rapidly prototyping new high-level protocols for testing before widespread deployment. With tools for prototyping, it is possible to do extensive protocol testing and code modification, and still be ahead of schedule at project completion. Other protocol verification and/or generation tools [9] could also be used but are geared toward low-level protocol stacks and not the higher level protocols such as HTTP or SMTP. If we use the OSI reference model to explain the level at which our tool operates, then we are concentrating on the application layer rather than the transport layer.

In the remainder of this paper we discuss the structure of NEST, followed by an in-depth presentation of the specification language with examples, then related work and conclusions.

II. Methodology

Our preliminary implementation of NEST has a structure like that found in Figure 1. The user supplies a NEST specification to the NEST translator, the NEST translator reads in the specification, then generates the appropriate C code for the chosen type of server. The generated code is then compiled and linked, along with any extra C code that the user wants to supply and any library code they may want to use, to form the final server executable.

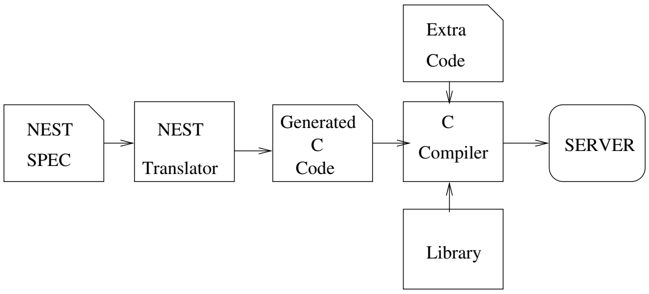

Fig. 1. Server generation

NEST also provides the user with the ability to choose between two different types of server executable. They may choose a stand alone server or a server that can be started via the Internet "super-server" `inetd`. Switching between these two choices can be accomplished by changing one line of code in the NEST specification.

III. The Specification Language in Depth

The specification language design is inspired by the design of Lex and Yacc [5]. There are three sections, separated by `%%`:

```
declarations
%%
rules
%%
C code
```

The last section is simply C code which is copied directly to the output file. Example specifications are shown in Figures 2 and 3.

A. 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. This C code is copied verbatim into the output file of the NEST tool.

The remainder of the declaration section is used for defining NEST options, states and macro substitutions. Notice that `%option` is used to inform the translator that a single server option will follow. In Figure 2, for example, the `processServer 150` option indicates that a process-based server should be generated by NEST, 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` in this portion of the sample code. In Figure 3 we use the `threadedServer 400` option to indicate a threaded server with a maximum of 400 active threads. We also choose the `inetd` option in this example to indicate that the daemon manager will be used to start the resulting server.

Some options in a NEST specification can be overridden via a special file whose default name is `config.nes`. (This file name can, of course, be changed in the NEST specification.)

```
{%
    #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 [^\r\n]*

%%

char user[256];

START: {
    $reply("Username: \n");
    $seen(EXPECT_USER);
}

EXPECT_USER: ANY NL {
    strlcpy(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 ANY NL {
        char *URL = $3;
        $sendFile(URL);
    }

    "PUT" WS ANY NL {
        $receiveFile($3);
    }
}

%%

// code_in_main_function
```

Fig. 2. Sample of simple authentication and HTTP-like commands

```
{%
    #include <string.h>
%}

%option threadedServer 400
%option inetd

%state GOT_HELO GOT_MAIL GOT_RCPT
%state VALID INPUT

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

%%

char mailFrom[256], rcptTo[256];

"HELO" WS ANY NL {
    $reply("HELO ", $3, "\n");
    $seen(GOT_HELO);
}
"MAIL FROM:" WS ANY NL {
    strlcpy(mailFrom, $3,
            sizeof(mailFrom));
    $seen(GOT_MAIL);
}
"RCPT TO:" WS ANY NL {
    strlcpy(rcptTo, $3, sizeof(rcptTo));
    $seen(GOT_RCPT);
}

GOT_HELO && BOTH {
    "DATA" NL {
        $seen(INPUT);
    }
    INPUT: NL "." NL {
        mailSavedInput(mailFrom, rcptTo);
        $clear(INPUT);
    }
    INPUT: .*|\n {
        saveInput($1);
    }
}

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

%%

// code_in_main_function
```

Fig. 3. Sample code for an SMTP-like protocol

This file is supplied in order to give network administrators the ability to customize a server's properties without having to recompile the server.

The `%state` line is used to declare 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.

The macro substitutions in this section may be used to increase the readability of large or complex regular expressions and boolean state expressions. There are no limitations on the complexity of these substitutions. The syntax of substitutions is simply one collection of characters followed by a space and then the collection of characters to substitute, followed by a newline.

B. Rules Section

Following the declarations section is the rules section of our tool which starts and ends with `%%`. We start off with any global variables that are needed for each connection.

In the rest of the rules section, patterns are specified which NEST will try to match against the server's input. If an input pattern is matched, the corresponding 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 performing 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; a pattern only applies if its qualifying boolean expression is true. Valid boolean operators in NEST include the and (`&&`), or (`||`), and not (`!`) operators. The boolean predicate `lastseen` may also be applied to a state, returning true if and only if that state 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 the initial incoming data. (Some startup code can also be placed at the start of the last section in this tool but a dedicated start state is useful as well.) The `END` state can be used to clean up before dropping a connection if the user wishes, as this state will always be entered before a connection is closed.

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

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

The patterns themselves are regular expressions, extended to permit extra whitespace for readability. NEST also allows syntactic sugar, so that one expression may qualify a number of patterns, in the form shown below. There is an implied boolean "and" between nested qualifying expressions.

```
qualifying-expression {
        [qualifying-expression:]
             regular-expression
        {
                action-code
        }
        [qualifying-expression:]
             regular-expression
        {
                action-code
        }
        ⋮
}
```

The code NEST generates will attempt to match the server's input against all patterns whose qualifying expression is true. If there is ambiguity regarding which regular expression matches, the longest match is chosen or, failing that, the pattern specified first.

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.

`$clear`
: "Forgets" that a state has been seen by setting it to false. The wildcard `*` refers to all states.

`$reply`
: Sends a reply from the server to the client.

`$close`
: Closes the network connection.

`$sendFile`
: Used for binary data file transfers.

`$receiveFile`
: Accept incoming file data.

Any number of other functions may be added in the future to help automate server infrastructure.

Inside the action code we also have access to the matched regular expressions of the pattern using a Yacc-like mechanism. In Figure 2 we have `EXPECT_USER: ANY NL`. Each "word" 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 `\r?\n`, after macro substitution). We can use these references in the action code anywhere that a character string can be used.

IV. Examples

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

A. Simple Authentication and HTTP

In the start state of Figure 2, we reply to any incoming connection with a username prompt. We then indicate that we have `$seen` the state `EXPECT_USER`. This means that `EXPECT_USER` is the only active state when the next incoming data is analyzed. The incoming data will be matched against the regular expression `[^\r\n]*` (which means everything except for line terminators) followed by a newline. When the data is read in and matched, we enter the `EXPECT_USER` state where the input matched by `[^\r\n]*` is copied to the user character array. Other similar steps will eventually authenticate the user and put us in the `VALID` state, or all states will be set to false via the `$clear(*)` statement.

Once in the `VALID` state, our HTTP-like `GET` and `PUT` commands are recognized.

B. Simple SMTP Protocol

The simple SMTP-like server in Figure 3 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.

This threaded server specification shows macro substitution used as part of a qualifying expression, and also illustrates how the `QUIT` command can always be recognized. It has several calls to user-specified functions (`saveInput`, `mailSavedInput`, and `cleanup`), and makes use of nested qualifying expressions.

In the `INPUT` state, two different regular expressions are used which result in an ambiguity. NEST's disambiguation semantics ensure that the input is matched by the correct regular expressions.

V. Related Work

We are aware of several frameworks and libraries that can be used for rapid server implementation but we have seen only one instance of relevant work using a new language to produce server code automatically.

A specification language approach to server code generation has been described and implemented in Java by Melvin Douglas and Philip Chan [2]. Their language, called My Simple Protocol Language (MSPL), can be used to easily describe an Internet protocol. The compiler will then use this description to generate the low-level communication and protocol modules for both the client and server applications. Some of the drawbacks of MSPL are that it does not support multiple server models, and it gives no immediate access to the underlying implementation language.

BEA's Tuxedo [3] is a commercial library that supports four different communication methods for software in a business environment. The functions in this library can be accessed from Java or C++. The library supports event-driven and multithreaded applications on several platforms but the internals of the program are not easily deduced.

Twisted [8] is an 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 features if a programmer wishes to exhibit more control over their system.

The Staged Event Driven Architecture (SEDA) [4] is also 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 other process-based or threaded servers.

Serveez [7] is a library written in C that provides much of the functionality necessary for quickly writing Internet servers. One of its main goals is portability 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.

The Adaptive Communications Environment (ACE) [6] is a large framework for writing concurrent communications software. It is written in C++ to support object orientation and patterns, or recurring solutions to a standard problem, which increases its flexibility, re-usability, and modularity when writing communications software. It is being used for many projects including embedded and real-time systems which is an indication of its speed and robustness.

VI. Conclusions

In this paper we have presented an outline of our network server tool, NEST. This tool can be used to quickly generate C code that implements three different types of server: process-based, threaded, or event-driven. NEST can be used to rapidly prototype new high-level protocols, improve programmer productivity, and reduce bugs and security holes related to server communication code. Other areas, like compiler construction, make extensive use of software tools, and the potential uses for tools in networking have only begun to be explored.

VII. Acknowledgments

Thanks to Kevin Wilson for proofreading an early draft of this paper. The second author was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada.

References

[1] Aleph One, Smashing the Stack for Fun and Profit, Phrack 7(49), 1996.

[2] M. Douglas, MSPL: A Protocol Language for Generating Client-Server Software, MS Thesis, Florida Tech (2000)

[3] BEA, Programming a Distributed Application: The BEA Tuxedo Approach, White Paper. http://www.bea.com

[4] M. Welsh, An Architecture for Highly Concurrent, Well-Conditioned Internet Services, PhD Thesis, University of Berkeley, (2002)

[5] J Levine, T Mason, D Brown, Lex & Yacc Second Edition, O'Reilly (1992)

[6] D. Scmidt, An architectural Overview of the ACE Framework, A Case-study of Successful Cross-platform Systems Software Reuse, USENIX login magazine, Tools special issue, November, (1998)

[7] S. Jahn, Serveez: http://www.gnu.org/software/serveez/manual/index.html

[8] G. Lefkowitz, I. Schtull-Trauring, Network Programming for the Rest of Us, USENIX 2003 Annual Technical Conference, pp. 77-90 (2003)

[9] M.B. Abbott and L.L. Peterson, A Language Based Approach to Protocol Implementation, IEEE/ACM Transactions on Networking, pp. 4-19 (1993)

[10] J. Ousterhout, Why Threads are a Bad Idea (for most purposes), Powerpoint slide presentation with no paper available, http://home.pacbell.net/ouster/threads.ppt (1995)

[11] S. Bhatkar, D. DuVarney, R. Sekar, Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits, 12th USENIX Security Symposium, pp 105-120 (2003)