The University of Kansas

Information and
Telecommunication
Technology Center

NetSpec: Philosophy, Design and Implementation

Roelof J.T. Jonkman
Joseph B. Evans

Information and Telecommunication Technology Center
The University of Kansas

ITTC-FY98-TR-10980-28

February 1998

Sponsored by:
Sprint Corp.
under prime contract to
Advanced Research Projects Agency
Contract DABT63-94-C-0068

Abstract

NetSpec is a tool designed to provide convenient and sophisticated support for experiments aimed at evaluating the function and performance of wide area networks. NetSpec was conceived as a general purpose mechanism for conducting a wide range of network level reproducible experiments. The key concepts behind NetSpec are: general purpose, portable, network level, and reproducible. Conventional tools for conducting network performance experiments, e.g., ttcp and netperf, have addressed the tests associated with a single connection reasonably well, but have not explicitly addressed the fact that a network level test requires sets of connections. In contrast, NetSpec was developed to support network level experiments, to permit constant expansion of the test types, and to ensure testing methods which were as accurate and reproducible as possible. NetSpec has been successfully applied to the AAI, this tool was the enabling technology for the performance evaluation of the AAI, a national scale terrestrial / satellite high speed network.

Contents

List of Figures

List of Programs

Chapter 1

Introduction

Currently there are two global networks logically, the telephony network and the Internet, which is essentially one big data network united by a protocol known as the Internet Protocol. Physically these two networks share infrastructure but at a logical level the networks are strictly separated. One of the key differences between the networks is that the telephone network provides a guaranteed constant service, whereas the Internet provides a best effort service. This seemingly small difference has a major implication; the user requirements and expectations are orthogonal. It seems apparent to the user that an integrated network for both services would be feasible, in reality this is still impossible, first because of the aforementioned reason, and second because of the explosive growth of the Internet, and the resulting lag of the infrastructure supporting the Internet. In todays research and standards development, one of the key issues has been the ability to combine various types of services on one logical network. Various approaches have been taken in order to achieve this goal, most notably, the next generation of the Internet Protocol, IPV6 or IPng, and Asynchronous Transfer Mode (ATM). The next generation of the Internet Protocol deems necessary for reasons other than combining services too; most notable the address space problem. Aside from that ATM and IPng address the problem at a different layer of the protocol stack, ATM is a link / network layer protocol, whereas IPng is one layer up, at the network / transport layer. Both of these protocols are still mainly applied in data communication environments.

The problem with emerging new technologies is always the question: “Does it work?” There are various ways to go about proving a new technology works, usually these stages represent the implementation process that results:

1. **Analysis** Analytically prove that a technology is feasible.

2. **Simulation** Simulate the system from the previous stage, see how it should work.

3. **Implementation** Build physical hardware that implements the functionality required.

4. **Testing** Test the implemented hardware.

The most interesting phase of the process is testing. Most equipment seems to be tested a couple of ways, conformance, interoperability and specifications, essentially does it adhere to the rules set. Most companies that work on new technologies already have a large investment in the R&D department and as such do not have a lot of money left for extensive large scale testing. This particular area seems to be left to the companies that buy the equipment and deploy it. The last stage of testing seems to be bypassed all too easily, and becomes more of a trial and error process. Rather than understanding systems by means of analysis, we tend to revert to empirically determining the workings of complicated systems. A very good example of this somewhat bold statement is the various large ATM networks that exist. Most of these networks are being used for bulk transfer of large quantities of data, rather than running real or artificial data traffic over it; they suffer from the "Big Pipe Syndrome". Obviously these large quantities of data represent a fairly ideal type of data traffic for a big pipe, certainly considering the fact that most of these networks are virtually empty. As a result of these shortcomings there is a strong need for testing tools, either hardware or software that can adapt to an almost violently changing environment. Hardware is obviously expensive, quite a bit harder to adapt to new developments and limited in terms of deployment since it usually requires expertise that is not necessarily available at every given Point of Presence (POP) in the network. For these reasons software is preferable, certainly if it can be used on (semi) commodity hardware (PC's, workstations). Therefore one would think there is quite a wide collection of software available for this purpose, oddly enough neither commercial nor free software excels in availability or functionality.

There are principally two types of testing tools, active and passive, the distinction being actively generating data and passively monitoring. In most operational networks there is hardly bandwidth for the use of active testing tools, passive measurements are usually the only thing that can be done on most operational networks currently. This also more or less explains why there is a considerable lack in the area of active testing tools for networks. Most active testing tools that are available currently are primarily focussed on point to point links. While these metrics are useful, they are not particularly suitable to obtain functional information about new networking technologies. Another "feature" of the available active testing tools is that they focus on the "Big Pipe Syndrome" that is bulk data transfer, rather than mimicking known traffic

patterns. Most of the above problematic issues were very obvious on various large scale ATM test beds, most notably AAI[1] and vBNS[32]. Out of these large scale networks grew the awareness of the fact that current tools are too limited in terms of realistic tests. The biggest problem that needed to be addressed was the control of many point to point connections, since that is what real traffic currently is predominantly made up off. The second issue that arose is that in order to know what you are doing there is a need for the integration of passive measurement tools also. NetSpec's intentions are to provide a framework that allows both the use of passive and active tools in a convenient manner. In order to also alleviate the limitations of current point to point tools, NetSpec also includes a traffic generator that pioneers a couple of new concepts which under certain circumstances gives considerably better precision.

In summary, these are the key design criteria that were initially set for the development of NetSpec.

Scalability The framework that is provided needs to be scalable, networks generally carry hundreds of flows simultaneously.

Flexibility The framework needs to flexible enough to be able to cope with both passive (probes, measurements) and active nodes. (e.g. traffic generators)

Reproducible The results need to be reproducible, assuming that the state of the network did not change.

Integration Measurements and tests need to be integrated in a "seamless" manner.

Extendible Addition of unknown new components should be anticipated.

1.1 Software Engineering Aspects

The other major goal of this research is to outline the somewhat non-traditional path NetSpec has taken to develop. Most academic software tends to be developed in an ad-hoc fashion, while the project is going, more and more requirements and ideas are tacked on to the original specification. Often requirements surpass the resources. Though NetSpec's development initially went along similar lines. Starting with version 3.0 the design decisions were not deadline driven anymore. From a true software engineering perspective these are how the three versions of NetSpec can be classified:

Version 1.0 Concept.

Version 2.0 Prototype.

Version 3.0 Alpha / Beta version

Version 4.0 The future

One of the major problems during the development of NetSpec was the constant pressure of the need for this tool on AAI. After 2.0 was in a usable state, there was a short amount of time in which there was some space for proper software engineering techniques that could be applied. One thing to note at that point though, so much of the conceptual model was already well built, that only at an implementation level proper engineering could be applied.

1.2 Organization

Since NetSpec currently consists of a general control framework, a test daemon and a simple tap (passive measurement) daemon the main focus of this thesis will be:

- The control framework
- The test daemon

First though the general specification, philosophy and a brief history will be discussed in Chapter 2, followed by the discussion of the implementation of the control framework in Chapter 3. Chapter 4 explains some of the somewhat programming challenges that were faced in the development of NetSpec. Chapter 5 will briefly discuss the tap daemon since this will be a starting point for any other daemons that can be implemented. Chapter 6 will be devoted to discussing the test daemon, since this daemon pioneers varies concepts that so far are not commonly used. Chapters 7 and 8 will discuss results and recommendations respectively.

Appendix A contains the module structure of the various parts that make up NetSpec.

1.3 Terminology

In order not to create any initial confusion it is useful to define some “NetSpec” terminology. The assumption is that the reader is familiar with standard Internet terminology. NetSpec has various parts to it:

User interface In version 3.0 a separate user interface came into existence, this is the executable that the user uses to interact with the top level controller. In version 2.0 the controller and the user interface were combined in one executable.

Controllers Although controllers in version 3.0 are daemons themselves, in general they are referred to as controllers.

"Trunks" Sometimes it is handy to refer to a part of the control tree as a "trunk".

Daemons A somewhat generic term that is being used for task specific daemons that fulfill a particular function, a traffic source or an SNMP daemon for example. Confusingly it also refers to daemons as we know those on UNIX systems.

"Leaves" Same as the above mentioned daemons, usually used in association with "trunks".

Server The beta release of 3.0 uses a multiplexer that multiplexes the various pieces of NetSpec onto one TCP port.

Nodes Generic term that refers to both leaves and controllers, basically components that can be part of the control tree. It does not include the user interface and the server.

Tree NetSpec 3.0 uses an arbitrary tree structure to control daemons, the term tree refers to the collection of nodes that make up the control tree.

Chapter 2

History and Design Philosophy

There are various testing tools freely available, most notably:

TTCP[21] TTCP is a tool that emerged from a simple tool to copy files from one Unix station to another, using TCP as opposed to uucp which uses UDP, to a benchmarking tool for point to point connections. TTCP is limited in its functionality to full stream, constant sized write() calls.

NetTest[29] NetTest is an heavily enhanced version of TTCP that supports multiple connections. It was written at Cray research. It does not incorporate a single point of control. Each of the endpoints has to be set up by hand.

NetPerf[14] NetPerf is in a lot of ways the answer to TTCP, it is a highly improved, very well engineered tool, including a results depository that allows users to deposit an compare results. The sole author is Rick Jones at Hewlett Packard.

TG[24] Traffic Generator is a tool written by Paul McKenney, Danny Lee and Barbara Denny at SRI. It is a tool that provides multiple connection control, and produces traffic according five basic random distributions. Though it looks very promising its language is very cryptic and it does not meet the requirements for NetSpec. (An entire parser for a C compiler is included as the parser for the language.)

2.1 History

The first version of NetSpec was simply a daemonified version of TTCP. It had a central point of control that enabled a user to control multiple TTCP sessions simultaneously. The first ver-

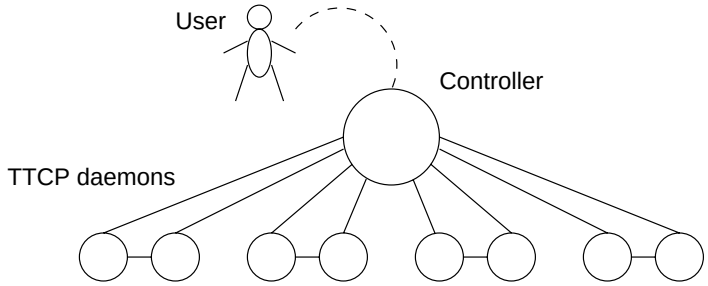

Figure 2.1: NetSpec version 1 process structure

sion of NetSpec was already obsolete by the time it was completed. The initial ideas[16] were already beyond what “remote TTCP” would offer. See Figure 2.1

So work started on NetSpec 2.0[15]. The design still revolved around a central controller, the controller also implemented the user interface. See Figure 2.2 The major differences with version 1 were:

- Formalized script language
- Introduction of the reporter daemon, designed to offload the controller, in reality was painful to deal with.
- The traffic sources and - sinks were designed from scratch.
- The synchronization of the traffic sources and - sinks was designed to achieve better synchronization of start and ending of connections.

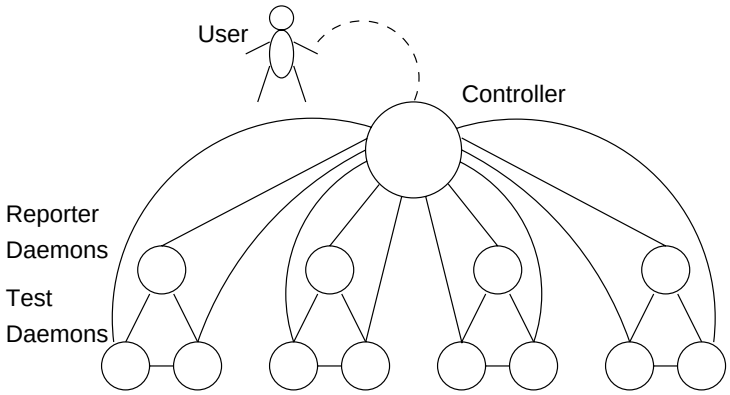

Figure 2.2: NetSpec version 2 process structure

- A binary control protocol

2.2 NetSpec Version 3

Shortly after version 2.0 was released it showed its weaknesses, and as such a thorough overhaul of the concept was deemed necessary. A couple of the major problems that were encountered during its use were:

- A single controller is cumbersome in not full mesh connected networks (A can see B, but not C, B can see C though, true in most ATM networks that have configuration problems.)
- The operating system limits impose limits in case of a single controller, which leads to scalability problems.
- The binary control protocol imposes tough limitations on extension of the protocol itself, and maintaining backward/forward compatibility at the same time. (Adding new parameters already posses multiple problems for the sake of transparency.)
- Distribution of the controller, a centralized controller poses too much scalability problems.
- Semantic problems within the scripting language, it was not very well defined.
- Framework extension with new leaves too complicated.

It more or less can be said that version 2.0 became a victim of its own success. The control framework had far more potential than what was implemented in version 2.0.

2.3 Philosophy

Throughout the project a couple of goals were constantly striven after:

- Portability, portable to various UNIX's, up to and including Windows NT.
- Modular, readable code, it should be easy to extend and modify the code.

Due to the first requirement C[2] [31] was the only language that could be chosen. C is very general and flaws that will easily leads to common programming mistakes. C is far from strict unlike languages like Pascal or Modula. This has one key disadvantage; mistakes are taken for granted by the compiler. One of the ways to avoid making mistakes more often than needed, is

to program according a certain philosophy, and stick to that, consistency is a key ingredient to successful programs.[18] This Section outline the ideas and disciplines behind the development of NetSpec. Of course ideas evolve over time, and some of the code is more mature than other parts.

Function length

To keep a good overview of the code, a general rule of thumb can be applied, functions should not be longer than one can fit lines on ones screen. Within NetSpec there are a couple of exceptions, the test functions themselves are longer than a screen. The reason for this is clear; coding efficiency is more important than readability in that case.

Prototypes

Function prototypes (ANSI requirement) declared in header files allow the compiler to check for the correct data type of each of the arguments that is being passed to the function. This will warn at compile time what's wrong, instead of having to figure out why the program fails when the pointer you passed was actually an integer value 0.

Header files are the "interface" definitions to the modules, it should define the types and functions needed to use to the module. Header files should not include C code, there is only one exception to that, and that is in line code. In line code needs to be present at compile time of the module, the only way to do so, is by "including" the source code for those functions that need to be in line. Compared to C++ classes there are striking similarities, in line function need to have the body of the code included within the class definition, which resides in a header file.

Naming conventions

Since it takes an average programmer about 20 minutes to write one line of code, the time it takes to type a slightly longer name is nothing compared to the lines it is being written in. In other words, using meaningful naming for functions and variables only adds to clarity and readability of the code. If done right the code should almost read like a book.

Bounds checking

C has one really major drawback that makes it hard for people to use, and that is that it does not do any type of length checking on arguments. This simplifies the compiler and the run-time overhead considerably, but leaves a gaping problem for the programmer. Any buffer that

stores user input has practical finite limits. In order to assure that the input is not exceeding the buffer limits, explicit checking has to be done in order to avoid buffer overflows. Buffer overflows usually results in corrupted data, and is hard to track down, since the programs behavior appears to be erratic. In particular for manipulation of character strings there is quite a variety of functions provides within the standard C libraries. The problem with most of those functions is that they do not have explicit length checks. There are however counterparts of those same routines that do have length checks. (The strxxx() vs. the strnxxx() routines.) Some of the standard input routines, most notably sprintf(), fread() and fprintf() do not have explicit length checks. This is fundamentally flawed. The only way around this is to use inferior functions, that don't provide the functionality that fread does.

Globals and static's

First thing you ever learn in programming class is not to use global variables. Partially right and partially wrong, there are certain cases in which global variables are indispensable, a good example within C is the variable errno, every system call that fails leaves an error code in errno. This mechanism is efficient and is consistent across any of the system calls. Another good example is the use of globals within modules, in C declared as static's. The scope of these variables is valid for the module it is declared in. It is almost equivalent to a variable declared in the private section of a class definition. The main advantage of using globals this way is that variables can be passed in between functions, without the user of the module having to interfere with the intrinsics of it. Static's are allocated within the data segment, and as such are there for the entire program duration, unlike regular variables which are allocated on stack. Note that this can lead to some confusion, namely, you can refer to a static variable while its out of scope by means of a pointer, this is considered bad practice though.

An very good example of using a static variable within a function is strtok(), it uses a static to retain the pointer to the tail of the string, for iterative calls, while it returns tokens one by one. A very viable way to retain state without passing variables that do not have a meaning for the user.

Modularity

Object oriented programming is in some ways just merely an extension of structured programming, in that it forces the programmer to piece up a problem in smaller chunks. Classes in object oriented programming are roughly the equivalent of modules. The problem with modules is that it is a relative easy mistake to incorporate too much functionality into one module,

Program 2.1 Inefficient if construction

```
if (someFunctionCall() == ERROR) {
  /* Uh Oh problem */
  return 1;
} else {
  /* Do what we need to do really */
  ...
  return 0;
}
```

this is the crucial distinction between modules in C and classes in C++. C++ classes enforce segregation of larger problems whereas modular programming is a discipline.

Coding efficiency

Efficient coding is one of the harder parts of programming. Two examples of efficiency are given here. Consider the if statement given in Program 2.1.

At a first glance there is nothing wrong with this, but if one considers bus arbitration, caching technology and relative slow memory speed one can see that the branch of the if statement under normal circumstances (when no error does occur) there is always a jump, this inherently corrupts the cache. And since we return directly from either branch there is always a function call return to be made at the end. A more optimal is given in program 2.2.

Now of course if after the if statement the body of the function continues instead of returns it is less of an issue. (Still the cache branch prediction mechanism could be helped, but that depends on the algorithm used for branch prediction.)

Another good example is the use of do { ... } while(); versus while() { ... } statements. If the loop is at least executed once a do { ... } while(); should be used, if the loop is executed

Program 2.2 More efficient if construction

```
if (someFunctionCall() != ERROR) {
  /* Do what we need to do really */
  ...
  return 0;
} else {
  /* Uh Oh problem */
  return 1;
}
```

zero or more times a while() { ... } should be used. The reason are the same as above, avoiding unnecessary branches, since this easily leads to cache corruption, and can have a fairly major impact on performance. Though it is harder to prove that a program is correct in mathematic sense when using do { ... } while(); constructs it is applicable in applications where provable working code is less of an issue versus correctly functioning code.

Chapter 3

Control Framework

The control framework is the part of NetSpec that by its definition is inherently hard to deal with. From the previous two versions of NetSpec it was quickly learned that a single central controller did not scale sufficiently. Aside from that issue it was also discovered that the coupling between the leaves and the controller is a fragile, the two most striking aspects are that the control of the leaves needs to be fairly tight, on the other hand, the information being passed between the leaves and user needs to be fairly loosely coupled. A good example of that is that the controller should only have a syntactical understanding of the parameters being passed to the leaves, it should not in any way attribute semantic values to the parameters being passed. On the other hand in order to achieve synchronized execution the controller should be able to tell the leaves exactly when they need to execute.

While this chapter revolves around the controller, two other instrumental components will be discussed also:

User Interface merely an extension of the controller.

Multiplexer/Server provides the gateway to all the executables.

3.1 Specifications

The controller needs to provide the glue between the user and the leaves. The user interface itself is assumed to be providing a simple pathway to the controller. The conceptual picture of the controller and its position in the framework is depicted in Figure 3.1.

Distributed The controller should be capable of being distributed across multiple machines and or processes. The reason for this is twofold; connectivity between nodes can not be

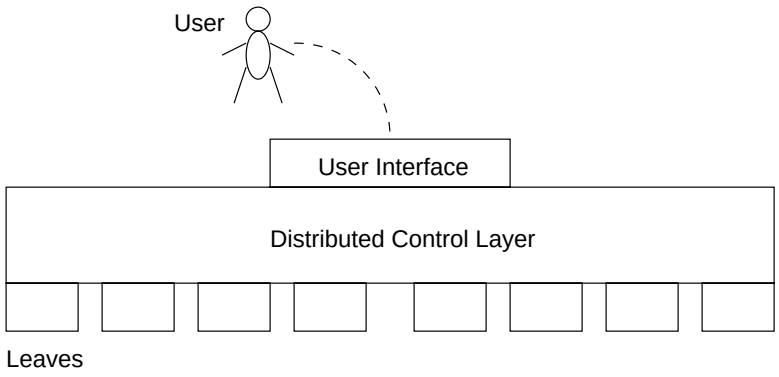

Figure 3.1: Conceptual controller model

expected to be complete, second, in order to circumvent host limitations, either multiple processes and or distribution over multiple hosts is in order.

Also the distribution aspect should if at all possible not impact the control language if there is no need for distribution. If the mechanics still use some form of distribution across multiple processes and or nodes that is not a a problem, as long as it is oblivious to the user.

Control Language The user should be able to specify an experiment in a simple block structured language. The language should be able to specify parallel and serial execution constructs. Other control mechanisms should even be considered. (e.g. Looping constructs, as mentioned before this could be hard in the sense that at the controller level some semantic value has to be attributed to the language in case of looping constructs or conditionals. This problem could be split into two, as will be explained in Section 8.3.)

Transparency The controller should be oblivious to the parameters that are passed to the leaves. No semantic value can be attributed to the parameters that are passed to the leaves. The reason for this is to guarantee controllers to be independent of the leaves, otherwise recompiles of at least the controller deem to be necessary whenever daemons or parameters are added. This would inherently lead to forward - and backward compatibility problems, which in turn would inhibit extension of NetSpec with new leaves.

Portability Of all attributes of NetSpec the controller needs to as portable as possible, for leaves this is less of an issue. (The main reason for that is that there is no good unified interface for tapping interesting internal information of the average machine, that inherently leads to a variety of non portable solutions on different architectures.)

Distinction If at all possible there should be no distinction between slightly differently functioning controllers. e.g. one that is just part of the control structure, one that talks to the user interface, and one that talks to the leaves.

Stateless The controller should impose as little state as possible. It is inevitable that the controller has some state to it, due to the parsing nature of the controller. The problem that potentially exists is that the controller and the leaves both have state machines that can collide. The leaves will very likely be state machine based.

3.2 Design

As mentioned before much of the concept and as such much of the underlying ideas were already well established when the design of NetSpec 3.0 was started. As a result of that most of the design is done in a bottom up fashion rather than the more usual top down method.

Two main design issues of the controller are, the text based protocol and the need to be distribute across multiple nodes. A text based protocol will require some kind of parsing in order to be able to pass on parameters, on top of that it needs to be able to interpret commands. Usually interpreters, compilers and the like can be divided into various sub functions. The two fundamental functions that form the first two stages of a compiler are a lexical analyzer and a parser. The lexical analyzer takes the input stream and "tokenizes" that, it chops a stream of letters and punctuation marks into words. The parser that controls the lexer then interprets the groups of words and tags a meaning to the sentence that comes out. Of course NetSpec is not the first thing to use a parser based control protocol.

FTP and Telnet are 2 good examples of text based control protocols also, they use fairly simple parser based keyword recognition mechanics. Based on early experiences with Lex and YACC in version 2.0, the decision was made to use these two tools to provide the protocol parsing engines for NetSpec 3.0.

3.2.1 Lex and YACC

On UN*X platforms there are two almost standardized tools, lex and yacc. [13] Lex is a lexical analyzer generator and yacc is a parser generator. (If one wonders; YACC stands for Yet Another Compiler Compiler)

Lex takes a specification file that is a combination of C and regular expressions[8]. Regular expressions are the practical form of expressing set theory in computer understandable form. After having lex decode the somewhat cryptic token descriptions, it generates a C source file

containing the entire lexer specification turned into C. Usually only one function is used *yylex()*. Various definitions and functions can be overridden, all of these make it possible to customize the code slightly.

YACC is similar in the way lex works, it takes a specification, and turns that into C code. The input specification file for YACC is more or less along the lines of Extended Backus Naur Form (EBNF). For both Lex and Yacc yields that within actions (the thing that happens when a match occurs.) C code is embedded.

NetSpec does not use yacc or lex that gets shipped with about any variety of UN*X currently. For the sake of portability mainly it uses flex[26] and bison[5] instead. Flex is a more rigid and more optimized version of lex that generates faster code, and besides contains less flaws and is highly portable. Flex was written by Vern E. Paxson at Lawrence Berkeley Laboratories. Bison is a YACC compatible parser generator written by Robert Corbett and made YACC compatible by Richard Stallman at the Free Software Foundation. Beyond that Bison is just like flex very portable, fast and efficient.

3.2.2 Distribution through recursion

The basic implementation and design of the language and the protocol is fairly straight forward. The biggest difficulty is not the language, nor is the control protocol, the aspect of being able to distribute across multiple nodes in an arbitrary and for the user convenient fashion poses a far bigger problem.

Most programming languages contain some form of circular definitions; definitions that define itself. Consider the following EBNF for a simple block structured language:

```
block :== '{' block | ... '}'
```

By applying the definition almost directly to the parsing code, the parsing code would be recursive. One of the parsing techniques that is used in a parallel parser is a technique called recursive descent. Recursive descent is a technique in which every semantic block is parsed by an instantiation of the parser. Most block structured languages have properties that allow a recursive descent parser to function easily. Considering the language specified in Program 3.1 which is fairly block structured a recursive descent principle would be somewhat applicable. Instead of a parser iterating itself in order to parse the blocks, it can also be done by multiple threads of execution, and this is why it is being applied in parallel parsing techniques. In Figure 3.1 a block structure is given with the potential of being parsed in different threads.

This is exactly what the distribution of the controller is driving at; different threads of ex-

Program 3.1 Block language

```
A {
    B {
        D {
            ...
        }
        E {
            G {
                ...
            }
            H {
                ...
            }
        }
        F {
            ...
        }
    }
    C {
        ...
    }
}
```

ecution. The only major disadvantage of using a recursive descent technique is the fact that a resulting structure is inherently some hierarchic (parse) tree like structure. The other problem with recursive descent is that the interaction between the blocks determine the level of parallelism one can achieve. If the interaction is fairly minimal, e.g block a does not affect its enclosed block b than using different threads is possible. In programming languages this is by far not always true, and as such a recursive descent technique applied in parallel compilers is not all that common.

A related problem with recursive descent techniques is that in principal a LL(k) based parser is required. One of the fundamental problems with a LL(k) based parser is that both syntax - and semantic checking is more complicated. As a result of that error recovery is also more complicated. This is one of the main reason why the slightly more general LR(k) parsers are used most of the time. The drawback of LR(k) parsers is that they are in general more complicated than LL(k) based parsers.

However for the type of structure that the controller requires, an arbitrary tree structure is fairly well suited. The user interface represents the root of the tree. The proposed idea here is to use a client server connection for each branch in the tree. This seems farfetched, but in reality simplifies implementation matters considerably. Since each recursion is transparent to the

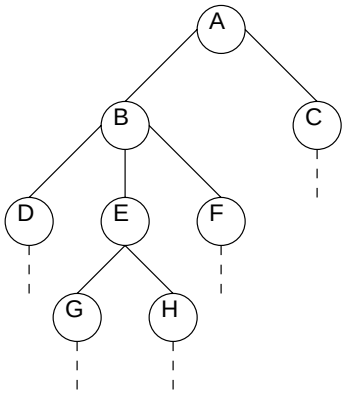

Figure 3.2: Block structure

previous one, and can be on a different host. Another major advantage of this approach is the problem of dealing with multiple nested constructs for parallel execution. If other techniques were to be applied; the controller would need to parallelize itself in order to control more than 1 nested block. In Figure 3.2 the resulting tree structure of the parsing of the blocks in Figure 3.1 is shown.

The application of a recursive descent technique without collapsing the results of the parse back to the root of the tree is somewhat unusual. In our case the result of the parse sets up the entire control structure, after which the actual connections between the instantiations of each of the blocks provide the required communication infrastructure.

Not really recursive descent?

The use of the term recursive descent is somewhat confusing in conjunction with YACC based parsers. YACC based parsers are LALR(1) parsers and as such can strictly speaking not be used to create recursive descent parsers. For a truly recursive descent parser a LL(1) parser is required. The presented technique here does however exhibit very strong similarities with a recursive descent technique.

In essence the parser calls itself in a recursive manner, albeit that the executable is forked off by means of a client server connection and the arguments are being passed by means of that connection. Next each nested block is parsed by a different instantiation of the parser. Each new instantiation of a parser is caused by its parent.

Program 3.2 Basic block syntax

```
parallel <address> {

  ...

}
```

3.2.3 Language design

The language should have familiar syntactical properties; C like features, this should form a minimal obstacle in learning. The next requirement is that it needs to be block structured.

Execution constructs

Though the controller can not attribute any semantic value to most of the script it still needs a way to derive connection addresses and execution behavior. Principally there are two basic forms of execution;

- Parallel
- Serial

Intuitively the idea is to to use a syntax like outlined in Program 3.2. This way there exists a clear distinction between the parts that the controller needs to understand, the keyword that determines the execution type and the keyword that represents the address at which to contact the next node. This construct also nests easily, and is suitable for the recursive descent distribution approach. The address can be derived implicitly also, if the assumption is made that an empty address means local host. As such the default assumption for the address is local host unless otherwise specified by the user.

There is only one subtle problem with this syntax, the top level block defines the current node. Obviously the current node can not contact itself, in other words, the address lost its meaning, but the execution keyword did not. The simple way around this is to just ignore the address within the current block's scope. That way the current node can derive its intended execution behavior.

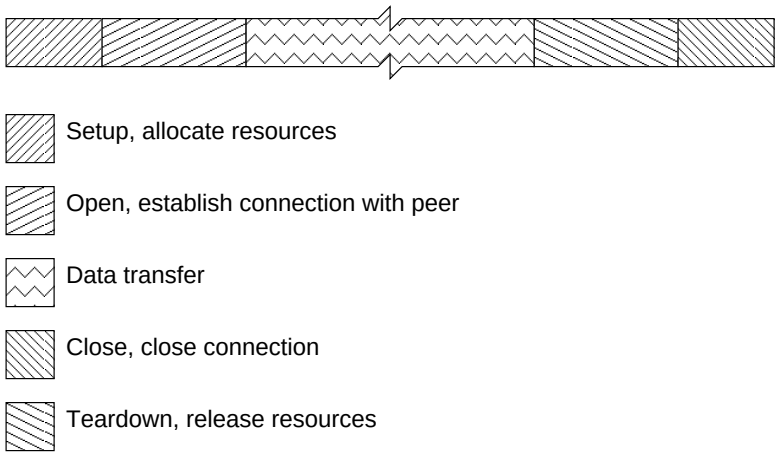

Figure 3.3: The five distinctive phases of a network connection.

Addressing leaves

A special case of execution constructs is needed in order to distinguish between various leaves. It is stated clearly that the controller can not attribute any semantic value to the parameters specified within the blocks. So that leaves the keyword and the address to specify all the information the controller requires. The keywords parallel and serial address the controller. Similarly for each of the leaves we could create unique keywords that could be used to address the specific leaves. Like the controller it would be possible to use multiple keywords to address one leave, though that would be somewhat obscure. (The controller is addressed by the keywords parallel and serial.)

3.2.4 Control protocol design

Since the primary objective of the control framework is to control traffic sources and sinks the control protocol is tailored to execute network connections with as precise control as possible. The way any network connection can be characterized as shown in Figure 3.3.

The five phases of the connection as such dominate the protocol that is used to control the leaves. Other leaves that are associated with doing measurements related to the connection will fit this protocol reasonably well, the only difference is that most measurements lack the open and close phases. Within the implementation this simply results in a null action. There is also a fairly distinct difference between unreliable and reliable protocols, within the open phase of a connection the reliable protocols will establish a connection, and as such will exchange data.

Unreliable protocols in principle don't need to establish a connection and as such won't need to exchange data at this point *. To control the five phases of the connection the following commands need to be implemented:

setup Allocate memory, fork processes, create service access points.

open Establish connection, e.g TCP initiates the 3 way handshake.

run Start the data transfer or measurement.

finish Finish the data transfer or measurement.

close Close the connection

tear down Free up all resources allocated, do not however discard test results.

Among these five core control commands there is also a need for some administrative commands:

report Tells the leave to return its results.

reset Reset the state of the leave to it's initial state if possible.

kill Over and out.

parameters Transfer parameters, do nothing else.

config Request configuration information.

Since the protocol and language are very closely related the intuitive block structure that dominates the language can be found also in the control protocol. Principally every command has a syntax as shown in Program 3.3.

Parameter specifications

The outside of the block structure has been discussed in the preceding paragraphs. That leaves the parameters that go within the blocks. As mentioned before, the controller nodes should not have a semantic understanding of the leaf parameters, but should have an understanding of the syntax. The reason for this is dual, first the controller can otherwise become derailed

*Though the argument can be made that a "ferret" packet needs to be send at this point. Networking technologies like ATM are connection oriented and as such require dynamic connection setup and tear down. The actual circuit setup can take a considerable amount of time[30], and in case of relatively short duration tests the initial connection establishment affects the results in a somewhat unpredictable manner.

Program 3.3 Command syntax

```
keywordCommand {

  ...

}
```

simply by problems within the parameter Section, next there is a need for checking at the highest level possible, so simple syntactical errors don't have to propagate the entire control tree, before getting detected. The syntactical checking has one drawback though, it will slow the process of setting up the control tree down. But setting up connections for each recursion takes considerably longer. Seen in that light the overhead of syntactic checking vanishes.

The general specification of a parameter is <keyword>=<value> in which the keyword is an identifier and the value can be any of the following:

IP address Either the numerical or DNS representation of an IP address, e.g. peer = stephens or own = 129.237.125.220. There is one variation of this that is non standard, and that is whenever there is a need to identify specific ports, the IP address needs to be extended with the port number which is an unsigned short integer. e.g. "peer = stephens:36644", in this the colon is used as delimiter. The forthcoming IPV6 addressing scheme uses colons as delimiters also, this might pose some minor problems.[28]

Identifier A typical C definition of an identifier e.g. NetSpec

String A string constant, e.g. "Anything goes here".

Integer A plain integer e.g. 16, there is also a slightly more involved version of this, for example an integer with a unit multiplier, e.g. window = 16K, which would be 16 Kilo byte.

Real A floating point double precision number e.g. 1.04e-5.

Function A function is a concatenation of parameters, e.g. protocol = tcp (window = 131072, tcpmtu=536)

Boolean 'TRUE', 'FALSE' and lowercase equivalents.

The preceding list shows the currently implemented parameter types. Extensions to this are to be expected.

All the parameters can be internally dealt with as plain strings, there is no need for conversions until the parameters are handed off to the actual leaves. Refer to Chapter 5 for the specifics of the implementation of the parameter and slave control control protocol parsers.

Protocol replies

In order to respond to any of the commands issued there needs to be a set of defined acknowledgments

acknowledge Command is executed correctly.

error Command could not be correctly executed.

report Has the same meaning as acknowledge, the way the text is handled is different, purely for performance purposes. (The possibility of data compression needed to be provided, however data compression can impact the propagation of messages considerably.)

All of the replies can be accompanied by text which is passed on to the user interface in a verbatim manner. Due to the data volume that reports can represent the report command is dealt with in a different way than the acknowledge and error replies. Acknowledge and error replies are accumulated on a per node basis and are as such propagated to the parent levels. An error that is received by the controller from one of its leaves results in an acknowledge to the parent. (Does not hold for all constructs, see the Section 3.2.5.)

There is no warning reply, the reason for this is that a warning is basically a non-fatal reply. So it serves an information purpose only, this can be represented by a acknowledge accompanied by a message.

3.2.5 Directed broadcast

Aside from a potential test connection that leaves might have, there are certain situations in which it would be useful if the various leaves could communicate. In particular traffic sources, and - sinks have a need to exchange address information and synchronization information. In particular with address information the problem occurs before an connection is established, as such there is a need for a communication link between the two or more nodes that is established on fore hand. Aside from that it needs to be noted that a test connection between nodes can not assumed to provide any type of reliable data transfer. The immediate solution that comes to mind is to use the control channels to implement the communication channel.

The only major problem associated with using a recursive descent technique is that the communication between various nodes through the use of the control channels is somewhat cumbersome. In principle if leaves have a need to communicate they could potentially set up a direct connection to each other. This would at a minimum require the user to supply more information in the script in order to enable the leaves to set up a peer connection. The obvious way around this problem would be to use the existing control channel. The problem with this is that in principle the controller does not have any knowledge of relationships between leaves. The solution that early on was chosen, was to enlighten the controller somewhat as far relationships between leaves goes. Next to the earlier on discussed parallel{} and serial{} constructs, a third one is introduced cluster{}. The cluster{} is in behavior very similar to the parallel construct. The difference is that with cluster the controller can conclude that the leaves that are enclosed within the construct are associated with each other.

This opens up the potential of using the control channels as communication channels for the peer leaves also. The initial contact phase is still somewhat cumbersome though, the leaf that needs information needs to emit a request to the controller that the controller rebroadcasts in a sequential manner to the other leaves within the cluster. If the peer leaf gets the request it turns around and responds with the requested information. From there on the controller “knows” which peers are associated with each other.

Another advantage is that a one to N relation is easily being solved using this approach. Relating this for example to a multi cast type test, multiple clients could query a single server using this approach.

This feature has not yet been implemented in NetSpec 3.0, though the code provides hooks for this. The mechanism is called directed broadcast, it is a broadcast, but directed at a certain part of the tree, and aside from that it is sequential in nature, which is not the true definition of broadcast. (In some of the earlier documentation this is referred to as “tunneling”, which applies to tunneling information.)

Protocol replies for a cluster{} construct

Replies generated by one of the leaves within a cluster are dealt differently with than in the case of a parallel or a serial construct. The reason for this is that within a cluster leaves interact with each other, so if one fails, the entire cluster can be considered failed. This means that if one of the leaves of a cluster generates an error reply, than the controller itself also needs to generate an error reply. Unlike the parallel and serial constructs which in those cases acknowledge with a message associated with the error. The reason for this difference is that when an error is

considered fatal for either of the 3 constructs a single error somewhere in the tree becomes fatal for the top level controller and as such will essentially fail the experiment. While this most certainly will not be true, only a part of the experiment will have failed. This subtle drawback can be contributed to the choice for a tree structure, in any given tree structure with a single root this would have proven to be a problem. Again this is also justifies the use of a separate construct for the conglomeration of closely related leaves.

3.3 User Interface

The user interface has to perform three functions

- Read the provided script and hand it off to the top level controller.
- Generate the commands to run the experiment.
- Strip the protocol overhead of the returned information.

Since the controller has its own parse engine, there is no strong need to have a replicated copy of that within the user interface. As a matter of fact there is a reason against this; it would be both from an engineering and efficiency perspective cumbersome. A similar problem occurs within the user interface that also occurs in the controller, see Section 3.2.3. It has to extract the address that is used at the top level block, in order to contact the controller daemon that is referred to at the first block.

When the connection is established, the user interface is the source for commands to be generated. Since the tree has only one single root, this is the only place where an experiment can be initiated.

The responses of the controller will be based on the reply format as explained in Section 3.2.4. Since the user is not interested in seeing the inner protocol workings, there needs to be a simple lexical analyzer that strips off the control protocol replies. (Hence the name stripper.) This can be done with a very simple state based lexical analyzer.

Since the controller contains the bulk of the mechanics, the user interface can be kept relatively simple. This leaves options open to incorporate preprocessing and post processing features.

Program 3.4 More efficient if construction

```
#
# config file for NetSpec, netspecd uses this.
# see man inetd.conf
#
# format:
#
#service        userid  executable(abs path)    argv[0..x]

cluster         nobody  /usr/local/bin/nscntld  nscntld
parallel        nobody  /usr/local/bin/nscntld  nscntld
serial          nobody  /usr/local/bin/nscntld  nscntld
test            nobody  /usr/local/bin/nstestd  nstestd
dstream         nobody  /usr/local/bin/nsdstrd  nsdstrd
tap             nobody  /usr/local/bin/nstapd   nstapd
```

3.4 Multiplexer

A service provided by NetSpec has to conform to certain standards. One of the requirements for Internet Assigned Numbers[11] is that a service is only supposed to take up one well known port. In order to ever get NetSpec a well known assigned port number, the requirement of only allocating one single port needs to be satisfied. Since NetSpec consists of at least a controller and some type of test daemon, this implies that there needs to be a way of multiplexing the various daemons on one port. This can be achieved by using a “front door” daemon that provides an access point for all daemons that a certain NetSpec host could run.

For controller purposes every daemon is already identified by its own keyword, see Section 3.2.3. This keyword could also easily be used to uniquely identify the executable that is associated with the keyword. This is exactly the trick that can be used by the multiplexer. It is important that the multiplexer uses a configuration file that associates keywords and executables, hence no recompiles of the multiplexer are necessary in order to add or change keywords and daemons.

To give a reasonable idea of what the configuration file looks like Program 3.4 shows one. Once again note that familiarity is a key point in the design, the configuration file is laid out exactly like the inetd configuration file[10]. As is, the argument is not used to pass any information, but it could be used to multiplex more than one function into one executable. Since the keyword is provided to the leaf itself, it can decide based on that also.

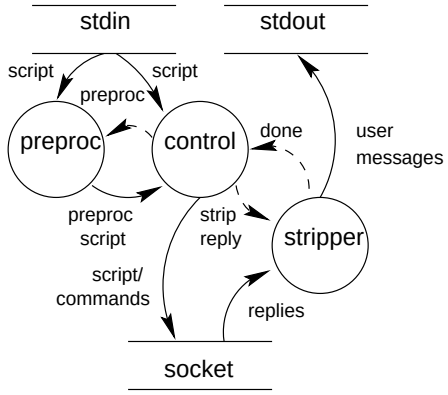

Figure 3.4: User interface

3.5 Summary of the controller design

The controller consists of a single parser engine that parses the control protocol as well as the parameters. The peers of a controller are kept in a linked list, this could be simplified to an array, since on any given architecture and OS there are limits on a per process basis for open sockets and since the maximum number of peers equals that, we could allocate a fixed sized array.

The reverse path of the control stream is extremely simple, a rough error indication, ok or error is enough for the controller to know what to do. The information passed back to the user is crude text, it can be any format, and any character, except a '}', since that is used as the delimiter, with a minor enhancement that could even be fixed. The reverse path only passes through a state based lexical analyzer. The reason the choice was made to pass this only through a lexical analyzer is that also passing it through a parser would slow the process down too much. In principle time stamped packet traces etc can represent quite a challenge to transport, due to their sheer volume. Within the lexical analyzer the actual text is handled by an extreme small piece of c-code, it is already taking a considerable hit due to the fact that it traverses the protocol stack twice. (Each recursion is a socket.)

Figures 3.4, 3.5 show the user interface - and the controller design respectively. The design method used is Ward & Mellor[25], which is a derivative of the Structured Design method[6].

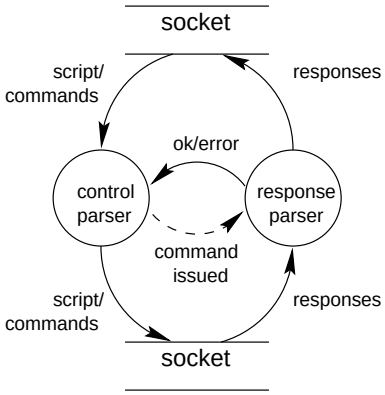

Figure 3.5: The parser structure within the controller

Chapter 4

Portability Issues and Programming Challenges

First of all the design of NetSpec has been somewhat convoluted from the beginning on, mostly because there was always some kind of conceptual picture of what the implementation would look like. Most of the driving decisions were mostly dictated by the environment NetSpec is targeted for; some kind of multitasking, multiuser operating system, UN*X being the first target, the Microsoft suite next. One of the most notable restraints that comes forth out of this, is portability. Since most Operating Systems adhere to some standard this should not be too much of a problem. In reality it is a serious problem though, superficially most UN*X'es are much alike. However as soon as one starts porting code, one stumbles on various subtle differences that have a large impact both in engineering time and limitations.

4.1 Portability Issues

In this Section the subtle differences are discussed using examples taken out of the source code of NetSpec. The most notable 2 mainstreams of UN*X are the BSD, Berkeley SoftWare Distribution, and AT&T System 5 release 4. The latest varieties of the various UN*X'es are more or less hybrid's, they incorporate bits and pieces of both these defacto standards. The only popular UN*X variety NetSpec runs on that tends to deviate from this is Solaris, it tries to adhere to the standard set by System 5.

Building on top of these defacto "standards" are the emerging standards, most notably; IEEE with the Posix specification, the X/Open group with the XPG4 specification, and ISO/ANSI.

Roughly there are three standards which dictate more or less what C is supposed to look like:

ISO/ANSI By far the oldest and most restrained standard.

POSIX Supercedes the ANSI definition.

XPG4 Incorporates all of the POSIX definitions, includes extensions on op of POSIX.

It is clear from the above that ANSI C is the choice for portability sake. As a result most of NetSpec is done in ANSI C. There are some exceptions to this rule, and that is where the feature would gain more, than portability was worth.

4.2 Text vs. Binary

As mentioned before in NetSpec 3.0 the decision was made to go with a text based control protocol. The two key reasons for this decision were portability and interoperability. Major portability problems are endianism * differences, different sizes for various "standard" datatypes (Long's on 32 versus 64 bits architectures are a good example.), and structure alignment problems that inevitably occur.

There are routines that convert from big endian to little endian and vice versa. The most common ones are the "ntoh" network to host and "hton" host to network routines. The network byte order is big endian. These routines are used with the IP address and network address translation routines. The datatypes they handle are limited to 16 bit short integers and 32 bit integers. This is obviously too limited for any slightly more involved application.

The other major and fairly extensive set of conversion routines are the ones included with the Open Network Computing Remote Procedure Calls (ONC RPC)† subsystem on most UN*X'es. External Data Representation (XDR) is the conversion library of function calls that provides a convenient way of converting architecture native formats to architecture independent formats. Two problems arise when using this, first the library is taylored for the use with RPC messages, second that would almost certainly limit the portability of the software to UN*X only. (And not even that much some UN*X'es lack the XDR routines altogether.)

The third option which would be as cumbersome as designing your own protocol in text, come up with our own conversion routines. But then still there is the caveat of how to ac-

*Big endian is most significant byte first, and little endian is least significant byte first. Denoting the way datatypes that span multiple bytes are stored and transported.

†ONC RPC is a defacto standard for remote procedure calls developed by Sun Microsystems. At this time it is dominantly used for implementing NFS, Network File System [20].

complish forward and backward compatibility between versions without the use of complex version numbering schemes.

Altogether there are too many potential problems with a binary protocol, albeit likely more efficient, it would undoubtly increase the development and debugging time. Debugging human readable messages is also quite a bit easier than decoding arrays of hex codes.

4.3 Why not RPC's

As mentioned in the previous section, there is also a mechanism to invoke Remote Procedure Calls, a mechanism to invoke functions from one machine on another. This would be a wonderfull solution for use within the NetSpec framework. There are again problems associated with this. First of all it is not very portable, everything supports TCP/IP, but primarily UN*X supports NFS, which is the only widely used application of RPC's. Second security is considerably harder to implement with RPC's than that it is with plain sockets. * In the original specification of NetSpec there are no provisions for enhanced security features. The framework can be easily secured with the use of TCP wrappers[33]. For more details on security related to NetSpec refer to Section 8.4.

RPC's are likely to be a fairly severe problem in terms of portability, aside from that it becomes quite a bit harder to provide entry points within to code to enable future security enhancements.

4.4 Programming

The test daemon was in various ways a challenge, one of the main challenges was to implement the test daemon in a way that a new protocol could easily be tagged on to the existing core without rewriting the test daemon. The features that were used are obscured within the code somewhat, though they are fairly elegant.

4.4.1 Unions and structures

One of the major problems in version 2.0 was the transfer of binary information, which was different depending on different parameters. The technique used to deal with this was to make a union of structures that each made up an unique set of parameters. This feature was still

*Though there are some serious efforts from Sun MicroSystems to secure the RPC system by use of public/private key mechanism.

Program 4.1 Union of structures definitions

```
typedef enum protocolType_t {
  tcpProtocol = 1,
  udpProtocol
} protocolType_t;

typedef struct tcpOptions_t {
  protocolType_t type;
  int xmtBuf;
  int rcvBuf;
  int tcpMtu;
  boolean tcpNoDelay;
  boolean passive;
} tcpOptions_t;


typedef struct udpOptions_t {
  protocolType_t type;
  int xmtBuf;
  int rcvBuf;
} udpOptions_t;

typedef union protocolOptions_t {
  protocolType_t   type;
  tcpOptions_t     tcp;
  udpOptions_t     udp;
} protocolOptions_t;
```

maintained in version 3.0. Mainly because it simplified the interface between the commands issued by the controller to the actual functions that execute the commands. On a sidenote it is more effective in terms of memory usage, but that is hardly an argument on a machine with at least 64 Mbytes of memory.

In Program 4.1 is a reduced snippet of the code that shows how the union of structure concept is done. The problem is that within the union there still needs to be an identifier that idicates which structure is being used. Otherwise there is no way to implicitly derive that from the data. The identifier used is of the enumeration type, it is in reality a plain integer, the advantage of doing this is that instead of having to use a “unrecognizable” value, one can use a name to identify the value. This increases readability of the code considerably. How a union of structures physically looks in memory is depicted in Figure 4.1. (The picture is strictly speaking not correct in that it does not address the alignment issues that play a role on most newer RISC processors.)

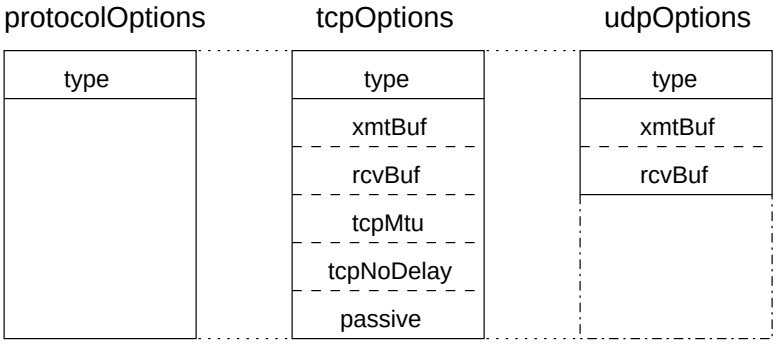

Figure 4.1: Memory map of a Union of Structures construct

Program 4.1 shows how a union of structures can be accomplished in C. Also note the way type definitions (typedef's) are used. The opinions on using typedef's with structures varies.[31] The somewhat confusing aspect of the way the typedef is used is that there are two ways in which the type can be declared, struct protocolType_t foo and protocolType_t bar. In both cases this leads to the same end result. Type definitions are mechanisms in which we can abstract the actual data (structure) and give it a name independent of its data type. In case we leave the struct keyword associated with that the abstraction is not as clean as one would like it to be. On the other hand if we use the struct keyword in the declaration it is considerable easier to see the distinction within the code between plain variables and data structures. Within the NetSpec code the convention is to use the data type that is defined by the typedef ommitting the struct keyword. Note that any type definition has the _t extension, this clarifies the use of either an enum, a structure or an union.

4.4.2 Pointers and functions

One of the features of C is that almost anything can be refered to by means of a pointer, including functions. In principle where ever a switch statement is used to distinguish between different function calls one can use function pointers, assuming that each has the same set of arguments. The advantage is that CPU does not have to traverse an ever ongoing switch statement in order to arrive at the right point, and as mentioned in Section 2.3 it is hard for caching algorithms to cope with branches, which in essence a switch statement is.

Combining function pointers and array's creates a simple table driven mechanism. Based on the index into the array the correct function is picked.

Program 4.2 Function pointers and unions of structures.

```
int openTcp(ptr2Address_t, ptr2ProtocolOptions_t);
int openUdp(ptr2Address_t, ptr2ProtocolOptions_t);
...
typedef int (*ptr2OpenFunction_t)(ptr2Address_t,
             ptr2ProtocolOptions_t);
...
static protocolOptions_t options;
static address_t address;
static ptr2OpenFunction_t openCommands[] = {
  NULL,
  openTcp,
  openUdp
};
...
openCommands[options.type](&address, &options);
```

4.4.3 Combining function pointers and unions of structures

The previous section lined out why function pointers can come in handy, but it does not help a lot if the arguments to the various functions use different arguments. And this is exactly where the use of unions of structures shows its full benefit. The program given in Program 4.2 shows how the aforementioned union can be used.

In reality both the open command for UDP and TCP have quite different arguments. Since the way the mechanism is used the selection is almost implicit, within the functions themselves there is no need to figure out what part of the union to use, it is already known, by the function through the pointer. This simplifies the entire housekeeping and management of multiple protocols as are being used by the test daemon. The mechanism presented here has similar features that overloading does in C++.

Chapter 5

Tap Daemon implementation

The tap daemon is the least complicated daemon that is currently part of NetSpec. Therefore this chapter forms the basis on which the next chapter (test daemon) will continue.

The function of the tap daemon is to provide a gateway to common tools like netstat that are hard to make portable. The basic idea is to have the tap daemon spawn the target executable and provide a wrapper that provides the interaction between the controller framework and the executable. See Figure 5. This approach enables NetSpec to use commonly available tools without the problem of portability.(E.g. netstat, tcpdump, atmstat etc.) The main disadvantage is the control is considerably less tight than required for testing purposes, for measurements however this is not really a problem.

The tap daemon consists of multiple modules, see Figure 5.2. Four of the modules are global modules, that are the same for any of the leaves. One module provides the daemon framework,

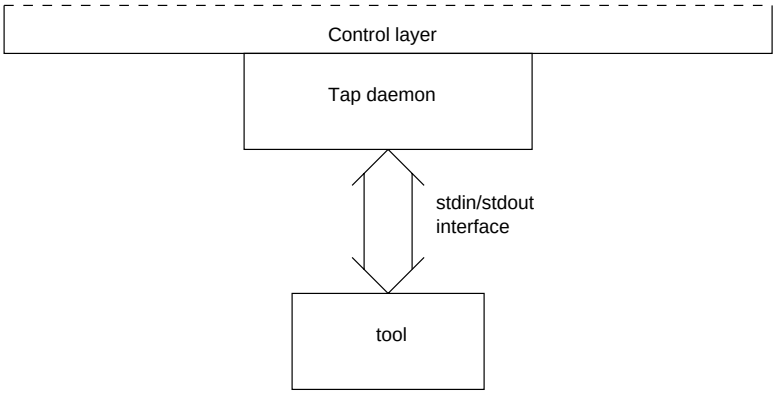

Figure 5.1: Tap daemon structure

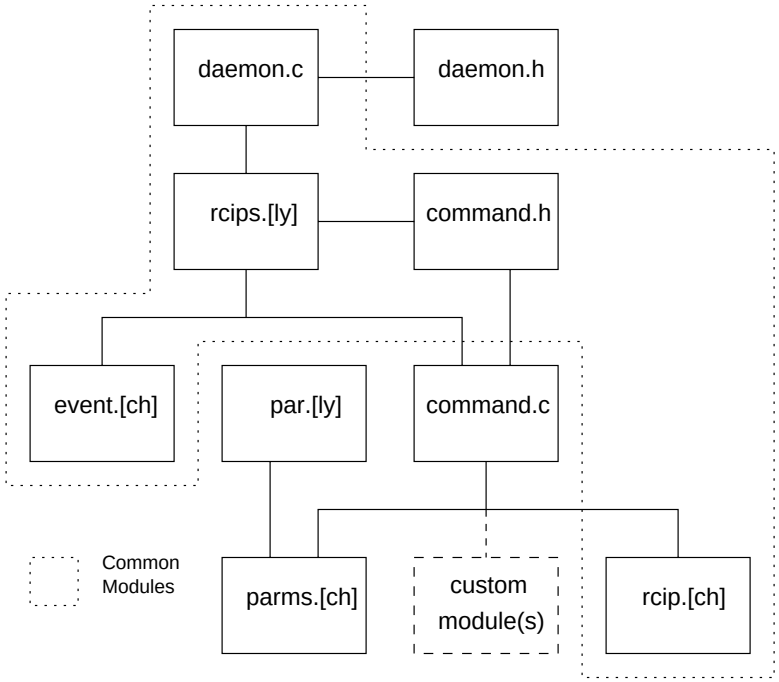

Figure 5.2: Tap daemon module hierarchy

the parts within the daemon framework that need to be customized are provided by means of C pre processor directives in daemon.c. Two modules provides the protocol implementation, rcips.[ly] and rcip.[ch] respectively. The protocolparser rcips.[ly] does not have any need for tailoring, other than the commands that it calls, as such the prototypes for each of the commands is fixed. The module rcip.[ch] contains all the functions that are used to generate report, acknowledge and error replies. The event.[chi] module contains the asynchronous I/O functions. The protocol parser uses these to avoid potentially blocking situations. The 3 main user definable components are:

par.l, par.y the parameter lexer and parser.

command.c the actions associated with each of the commands.

parms.c, parms.h the parameter data store.

Aside from these modules that are mandatory, the following modules are provides in the common library:

error.c, error.h Error and debugging information handlers.

linklist.c, linklist.h Generalized linked list module.

ipaddress.c, ipaddress.h IP / DNS address translation routines.

mesgbuf.c, mesgbuf.h Generic message buffer handling routines.

process.c, process.h Process statistics, rusage based.

peek.c, peek.h Poll / Select wrapper, unifies the interface to poll.

process.c, process.h, process.i Process statistics, rusage based.

report.c, report.h Generic report headers etc.

sockets.c, sockets.h Generic socket routines both UDP and TCP. (The test daemon uses a different sockets module that is slightly better suited for performance testing purposes.)

5.1 Common modules

The common modules, daemon.[ch] and rcips.[ly] are the server / inetd interface and the protocol parser respectively. The daemon module implements a standard front end that includes a stand alone server, interfacing with inetd and some helpfull information. The protocol parser and lexer rcips.[ly]* provide the parsers that is needed to interpret the commands that are issued from the controller. Each command, except for the parameters command results in a function call to one of the commands defined in command.[ch]. The commands match the definitions as given in Section 3.2.4. The parameters command results in a call of the parameter parser.

The following section highlights some of the more intricate details of the control parser and some of the command invocation problems. The main motivation for this is that some of these programming constructions might (will) inadvertendly affect the daemon side in an obscure way.

5.1.1 rcips.[ly]

The complexity of the parser is minimal, each of the commands is fully parsed, and following that the command is invoked. A typical command invocation is given in Program 5.1

The command invocation principally needs to happen in a way that does not relinguish the controller from being able to reach the daemon. The problem with any daemon that uses

*The module rcips.y is about to be replaced by a state based lexical analyzer, so only rcips.l albeit more complex remains.

Program 5.1 Typical command invocation within rcips.y

```
run
  : RUN options
  {
    debug ("rcips","run");
    if ($$2 != OK || invokeCommand(runCommand) != OK){
      rcipError (getError());
      clearError();
    }
  }
  ;
```

potentially blocking system calls is that whenever a command involves a potentially blocking section there are two ways of solving this problem. The first most obvious way is making sure that whatever system call is being done does not block. This is not always possible, simply because not all system calls provide a interface for polling. The second more general solution is using asynchronous I/O. The snippet of code in Program 5.2 shows how we enable a socket descriptor to generate asynchronous messages.

Since the parsers that are being used for this purpose are LR(1) based parsers * and as such got quite an extensive state machine. It would be wise to leave those statemachines undisturbed, the way to do so is by jumping back from within the command to the same place the command was invoked. That way we can continue within the parser without having to recursively call the parser.

*Like mentioned before the parser is about to dissappear, but that still leaves the same problem of having to jump back into the state based analyzer.

Program 5.2 Enabling asynchronous I/O with fcntl()

```
int linkControlChannelEvent(signalFunction aFunction){
  ...
  sigaction (SIGPOLL, &sigAction, (struct sigaction*) NULL);

  if (fcntl(0, F_SETOWN, getpid()) == SYSTEMERROR)
  ...

  if (fcntl(0, F_SETFL, FASYNC) == SYSTEMERROR)
  ...
}
```

Program 5.3 Using setjmp() in conjunction with asynchronous I/O

```
if (!sigsetjmp(returnState, 1)){
  linkControlChannelEvent(stateDispatcher);
  command();
} else {
  unlinkControlChannelEvent();
  debug ("blockstate", "got interrupted by controller");
}
```

The way (sig)setjmp() works is somewhat obscure; the first call stores the current stack context in the data structure 'returnState' and returns '0'. Whenever an non local return is done by means of invoking (sig)longjmp() the stack context is restored according the state stored in 'returnState'. This will cause the program to return to the place where (sig)setjmp() got invoked, in order to make a distinction between the 2 situations that can occur a different return value is returned. With (sig)longjmp() we can indicate any return value but zero, since that is reserved for the call that stores the context.

Considering that in principal the stack manipulation should work transparantly even if used recursively. But it must be noted that combining asynchronous I/O and non local jumps has been a very fragile part of the code on most operating systems that NetSpec runs on.

5.2 Custom modules

The modules that are of particular interest are; the parameter parser consisting of par.y and par.l, and command.c. The module parms.[ch] provides an interface between the parser and

Program 5.4 The interrupt handler.

```
static void stateDispatcher(int aSig){
  /*
   * input   : the signal that caused the interrupt.
   * returns : -
   * output  : -
   * desc    : returns from an non local jump, requires returnState
   *           to contain a valid stack context.
   */
  error ("command", "Something got us");
  siglongjmp (returnState, 1);
}
```

Program 5.5 The lexical analyzer and parser data exchange structure.

```
union {
  char          string[256];
  long          integer;
  double        real;
  boolean       boolean;
  FILE*         file;
};
```

the command module. In principle it would be possible to incorporate the parameter parsing function into the protocol parser. The decision was made not to do so. This way the developer does not have to deal with the protocol, and could focus on the parameters only. This construction also allows the protocol parser rcips.[ly] to be unified for all leaves.

5.2.1 The parameter parser par.l and par.y

The parameter parser is as standard as possible, the only slightly difference is that the lexer is reading out of a memory buffer instead of file stream. This however is entirely preprogrammed, so it should form no obstacle at all. The way flex implements this is the parent function reads the stream of text into the flex buffer and imposes an end of file condition such that the lexical analyzer will not pursue reading more from the non existing stream.

Strictly speaking the tap daemon only needs to use the identifier, string and integer, but for the sake of completeness all possible parameters will be discussed. The ones that do not have an equivalent within the tap daemon are taken from the test daemon. Following here are the parameters and of each an example of how to implement this in both the lexical analyzer and using the parser to store this into a data structure. The data structure that is used to exchange the values between the lexical analyzer and the parser is given in 5.5 Any of the tokens not mentioned explicitly in the lexer that are referred to in the parser are defined as follows:

```
"thisToken" return THISTOKEN
```

Identifier The example used in Program 5.6 shows the use of an identifier as an IP address. One of the problems of an IP address is that it matches more than one pattern defined in the lexer as such it is neccessary to incorporate multiple rules within the parser to create a catch all for an IP address.

Integer In Program 5.7 the tcp window size is used as an example. This shows that the window size for a tcp connection can be assymetric. (The socket buffer size determines the

Program 5.6 Lexical analyzer and parser code for an identifier

```
par.l:

  [a-zA-Z][a-zA-Z0-9_]* {
    strncpy (parlval.string, yytext, yyleng);
    parlval.string[yyleng] = '\0';
    return (IDENTIFIER);
  }

par.y:

  | OWNADDRESS '=' IDENTIFIER
  {
    ownAddress.type = ipAddress;
    ownAddress.ip.address = str2IpAddr($$3);
    ownAddress.ip.port = 0;
  }
```

upperbound on the window size.)

Program 5.7 Lexical analyzer and parser code for an integer

```
par.l:

  [0-9]* {
    parlval.integer = atol (yytext);
    return INTEGER;
  }

par.y:

  | TCPWINDOWSIZE '=' INTEGER
  {
    protocolOptions.tcp.rcvBuf = $3;
    protocolOptions.tcp.xmtBuf = $3;
  }
```

Real The example used in Program 5.8 is until now the only application of the real parameter; random number generators. Strictly speaking the regular expression is not complete in that it does only partially implements the way reals are being expressed.

Boolean Since the definition of a boolean can be either true of false or zero or one, the parser also includes a match for an integer. The way the assignment is done however is such

Program 5.8 Lexical analyzer and parser code for a real

```
par.l:

  [0-9]*"."{1}[0-9]* {
    parlval.real = atof (yytext);
    return REAL;
  }

par.y:

  | MEAN '=' REAL
  {
    ptr2DistOptions->normal.mean = $$3;
  }
```

that any positive or negative value results in true. Only 0 is considered false. Stricter rules could be enforced but are more or less pointless. Refer to Program 5.9.

String The way the string is solved is by using lexer states, in this case an exclusive state is used in order to exclude any of the other matching rules. There are other ways of solving this, but those impact the performance of the lexer slightly. See Program 5.10

IP address Just like the boolean, the IP address is a combination of multiple lexical tokens in the parser. Program 5.11 shows clearly which permutations of an IP address are possible.

Functions Functions are concatenations of other parameters, as such there is no lexer code to go along with, since it is purely a parser imposed construction. (Not entirely true, one could make a state based lexer with inclusive states.) The example in Program 5.12 shows the implementation of a function construct. Although not elaborated, distOptions itself is also a function construct, along similar lines as sinkOptions.

The above list describes the entire lexical analyzer and parser for the parameters and how they are use in conjunction with NetSpec daemons. The documentation that comes with Bison[5] and Flex[26] is eloborate and more than sufficient to cover the problems that might be encountered when implementing a daemon for NetSpec. A usefull reference is also Lex & Yacc[13].

Program 5.9 Lexical analyzer and parser code for a boolean

```
par.l:

  [Tt]{Rr][Uu][Ee]" {
    parlval.boolean = True;
    return BOOLEAN;
  }

  [Ff][Aa][Ll][Ss][Ee] {
    parlval.boolean = False;
    return BOOLEAN;
  }

par.y:

  | SODONTROUTE '=' BOOLEAN
  {
    protocolOptions.tcp.dontRoute = $$3;
  }
  | SODONTROUTE '=' INTEGER
  {
    protocolOptions.tcp.dontRoute = $$3?1:0;
  }
```

Program 5.10 Lexical analyzer and parser code for a string

par.l:

```
  <STRINGSTATE>{
    "\"" {
      BEGIN (INITIAL);
      debug ("stringstate", "leaving string state");
      return (yytext[0]);
    }
    [^\"]* {

      strncpy (parlval.string, yytext, 255);
      debug ("stringstate", "got string");
      return STRING;
    }
  }
```

par.y:

```
  : COMMAND '=' STRING
  {
    strncpy (commandLine, $$3, 255);
  }
```

Program 5.11 Lexical analyzer and parser code for an IP address

```
par.l:

  [a-zA-Z][a-zA-Z.-]* {
    strncpy (parlval.string, yytext, yyleng);
    parlval.string[yyleng] = '\0';
    return IPADDRESS;
  }
  [0-9]{1,3}"."[0-9]{1,3}"."[0-9]{1,3}"."[0-9]{1,3} {
    strncpy (parlval.string, yytext, yyleng);
    parlval.string[yyleng] = '\0';
    return IPADDRESS;
  }

par.y:
  : OWNADDRESS '=' IPADDRESS
  {
    ownAddress.type = ipAddress;
    ownAddress.ip.address = str2IpAddr($$3);
    ownAddress.ip.port = 0;
  }
  | OWNADDRESS '=' IPADDRESS ':' INTEGER
    ...
  | OWNADDRESS '=' IDENTIFIER
    ...
  | OWNADDRESS '=' IDENTIFIER ':' INTEGER
    ...
```

Program 5.12 Parser code for a function construction using identifiers

```
test
  : TESTTYPE '=' SINK '(' sinkOptions ')'
sinkOptions
  : sinkOptions ',' sinkOption
  | sinkOption
  ;
sinkOptions
  : BLOCKSIZE '=' distOptions
  {
    if (!testOptions.common.buffer.maxSize)
      testOptions.common.buffer.maxSize =
testOptions.sink.blockSize.common.max;
  }
  | etc.
```

5.2.2 The command module command.c

The command module consists of the functions that make up the commands that NetSpec leaves know about. The following commands are discussed in the order they are issued.

Initialize Initialize daemon and acknowledge existence to parent.

Setup Allocate resources for the requested task.

Open Establish a connection with the peer, or a null action

Run Run the requested task.

Finish Wrap up the requested task.

Close Close the connection with the peer, or a null action.

Report Generate a report of the results.

Teardown Release all allocated resources.

Kill Exit the process.

The following commands are purely administrative

Reset Reset the daemon to its initial state.

Config Request configuration information.

The only command that is not part of the control protocol is the initialize command. The initialize command is the command that gets invoked automattically on instantiation of the daemon. Every time a daemon is contacted, the controller expects an acknowledge back, just for verification reasons. Occasionaly it does happen that a machine renders in a state where connection request are still being accepted, but no processes are being spawned anymore, this will inevitably result in “hangs” if no measures against this were taken.

Any of the commands does not return any information, and no parameters are passed to each of the functions. The reason for this is that this way the interface with the protocol parser is always fully defined. The drawback however is the interaction with the parameter parser, and that is why the seperate module parms.[ch] is used to exchange data between the parameter parser and the command module.

Commands used within the tap daemon

The only 3 commands that implement anything interesting aside from the standard commands are:

- setup
- run
- report

The main reason for not being able to provide any more accurate control of the daemon is because there is not a clean way of controlling a child process *. Following are respectively the implementation of the setup run and report command. The remaining commands simply acknowledge without any associated action.

Setup

1. Create pipes
2. Create temporary file.
3. Disable buffer on pipes and file.
4. Attach output pipe to temporary file.
5. Catch SIGALRM for later perusal.

Run

1. Spawn the commandLine that was provided in the script.
2. Start timer (This seems backwards, but it is slightly faster to start without processing the timer call first and then spawn the child.)
3. Suspend, wait for SIGALRM to fire.
4. On return from the SIGALRM, kill the child
5. Close the file and pipes.

* Not entirely true, by means of using the signals SIGTSTP (interactive stop), and SIGCONT (continue) one can put a process to sleep and awake it again, this assumes however that the child process is an interactive process, which is true most of the time but is definitly not a general mechanism.

Program 5.13 Using designated ports for the purpose of debugging

```
cluster {
  tap:42000 {
    command = "netstat -I ln0 1";
    duration = 10;
  }
}
```

Report

1. Open the temporary file for reading
2. Funnel the temporary file into the report pipe (stdout)
3. Close the temporary file
4. Remove the temporary file

5.3 Debugging

The problem with multiple processes on multiple hosts is that debugging becomes a cumbersome and elaborate process. First of all inetd and netspecd form two annoying obstacles. These can be taken out simply by running the daemon standalone, each of the daemons includes a full fledged server, that can serve under inetd as well as standalone. As such the daemons can be started on its own port eg.

```
nstapd -s -p 42000
```

The port used is the self defined experimental server port for NetSpec, netspecd and inetd principally listen on port 42003* An example script to attach to this standalone daemon is given in 5.13

Running the daemon under a debugger is fairly trivial:

```
pepe:/users/rjonkman > gdb ‘which nstapd‘
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (alpha-dec-osf4.0), Copyright 1996 Free Software Foundation, Inc...
```

*As with some Internet standards that emerged certain oddities got included, this might be one of those, the port number is derived of the author’s mother’s phone number.

```
(no debugging symbols found)...
(gdb) run -s -f -p 42000
Starting program: /usr/local/bin/nstapd -s -f -p 42000
```

The '-f' flag instructs the daemon not to fork, since that usually complicates matters within debuggers. In standalone mode the daemon still does not do any I/O to the console, so still the functioning has to be derived from the control connection. This once again can be circumvented by using the debugging flag. The debugging flag '-d' takes an integer arguments, and should represent the level of verboseness which reflects the hierarchy of the modules see Figure 5.2. This will emit debug information to the console.

Another feature that the user interface includes is a simple hand controlled single stepper for the protocol, it is invoked as follows (using the script test, see Program 5.14):

```
pepe:netspec/scripts/debug > netspec test -i
NetSpec Version 3.0 alpha 1 Control Daemon pepe[30260] ready
NetSpec Version 3.0 TCPgamma 1, UDPalpha 1 Test Daemon pepe[30267] ready
NetSpec Version 3.0 TCPgamma 1, UDPalpha 1 Test Daemon wiley[3492] ready

NetSpec debugging mode ready:
Type ? for help

rcips > ?

NetSpec RCIPS debugging mode description

Local commands are:

? or help
quit

RCIPS commands are structured as follows:
command; or command { parameter; parameter; .... }

rcips commands are :

setupCommand
openCommand
runCommand
finishCommand
closeCommand
teardownCommand
killCommand
resetCommand
paramsCommand
configCommand
reportCommand
For more details see the programmers manual (Not done yet)
```

The debugging mode is quite elaborate, with each command it is possible to specify parameters as appropriate for the daemon. With this option it is possible to step through each

Program 5.14 Sample test script

```
cluster {
  test pepe.atm {
    type = full (blocksize=8192, duration=30);
    protocol = udp (xmtbuf=65000, rcvbuf=65000);
    own = pepe.atm:43006;
    peer = wiley.atm:43006;
  }
  test wiley.atm {
    type = sink (blocksize=8192, duration=30);
    protocol = udp (rcvbuf=65000, xmtbuf=65000);
    own = wiley.atm:43006;
    peer = pepe.atm:43006;
  }
}
```

command by hand, and allows carefull observation of the actions of the DUT (Daemon Under Test).

Another option is to telnet directly to the port of the daemon, since the control protocol is fully text based it is almost equal to the interactive option of the user interface with the subtle difference that the replies are not stripped of the protocol overhead, whereas with the interacive interface the replies are stripped.

The error (error.[ch]) module contains routines for both error and debugging information messages. They have a standarized form, such that the distinction between different hosts and processes can be made. Debugging messages are only generated when the executables are compiled with the 'DEBUG' symbol defined. Otherwise only error messages are generated. This way debug messages can be put all over the code without having to be concerned about performance issues, as soon as the daemon is stable, the executable can be compiled without the debug information.

5.4 Procedure for building a NetSpec daemon

1. Classify the code to be applied to the framework according to the given commands. See Section 3.3.

2. Classify the parameters:

   - names

- values
- functions

3. Implement the parser and lexer. Use the report function to report all the parameters.
4. Implement the code, pay attention to the acknowledges / errors that are generated and why.
5. Test using the procedures described in Section 5.3
6. Done.

Chapter 6

Test Daemon Implementation

The chapter relies on the basis that is provided in the previous chapter discussing the tap daemon. The test daemon is by far the most crucial part of NetSpec. It provides the capability to generate and sink a wide variety of TCP - and UDP traffic. The module structure is given in Figure 6.1 The common part as shown in Figure 5.2 is ommitted.

As can be derived from figure 6.1, TCP and UDP are almost synonimous. The differences are very marginal. Within the tcp.c and udp.c only two functions differ substantially; setupUdp / Tcp() and openUdp / Tcp(). The difference is that TCP requires the three way handshake process, whereas UDP is devout from that.

Which module is what?

defaults Provide routines that default each of the data structures declared in parms.c. The problem is that each parameter can not be assumed to be specified within the script. As such each of the data structures needs to be filled with sensible data. The defaults module provides a global place to do this.

udp The functions associated with the UDP class of tests.

tcp The functions associated with the TCP class of tests.

testtypes The definitions for the data structures used in conjunction with tests, this is independent from the protocols. Refer to Section 4.4.1.

protocoltypes The definitions for the data structures used in conjunction with the protocols. Refer to Section 4.4.1.

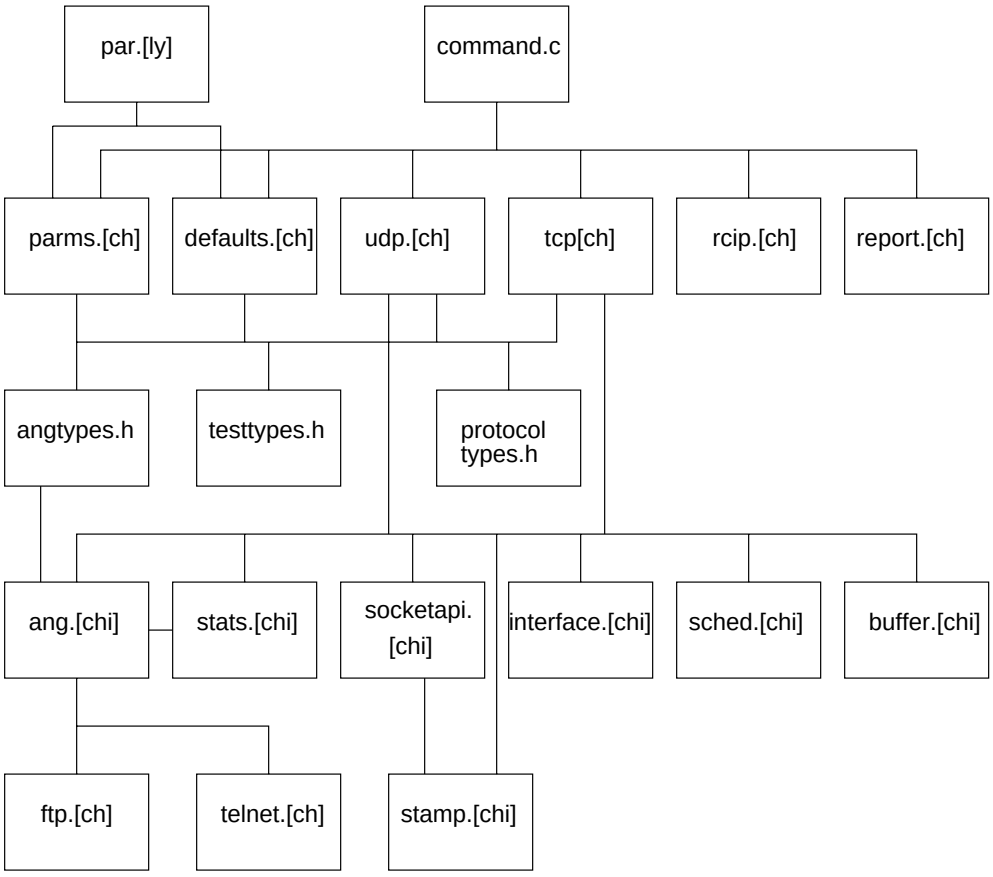

Figure 6.1: Test daemon module hierarchy

socketapi The high(er) performance implementation of the standard socket API provided in the common library. There are certain aspects that as a result of the various traffic types had to be done considerably different from the common API.

stamp Time stamp routines for stamping the system call that take place within the various protocol modules. For detailed analysis this is an essential tool. (See chapter 7 for details on this.)

sched The scheduling facilities, this is both for scheduling tests, and is also used for traffic generation purposes. Provides convenient wrappers around the cumbersome implementation of UN*X signals. (Mostly BSD and SYSV tend to collide on definitions and results of the same function. POSIX tries to set a standard, and is slightly different from the two previous standards.)

buffer Buffer routines, currently partially implemented, but provides the framework to cycle through a linked list of a buffers, so less optimal (no caching influence) conditions can be

interface Interface statistics, currently unimplemented, there is no portable way of deriving physical interface statistics on a per connection basis. For each architecture this module needs to have a seperate implementation.

stats Statistical routines, keep statistics on the various random numbers.

ang Arbitrary Number Generation (ANG) routines. A generalized interface to all random / constant number generators. Not hyper efficient, uses floating point calculations for most distributions. It could be slightly optimized by shifting the calculations to long's. Problem is that long's on 32 bit machines (including Sparc Ultra's running Solaris 2.6) are 32 bit. As such there is too little precision to do some of the calculations. The numbers that are returned are always integers.

angtypes ANG data structure definitions. Refer to Section 4.4.1.

ftp Tables for the File Transfer Protocol random distributions.

telnet Tables for the Telnet protocol random distributions.

6.1 Traffic generation

There are roughly two types of traffic generation, full speed, and paced. The full speed traffic generation is merely a while loop with a write() system call. The paced traffic generation

Program 6.1 sleep() based traffic generation

```
...
while (...){

  write(socket, data);
  usleep(period);

}
...
```

algorithms are much more complicated. This section will focus entirely on the paced traffic generation. There are basically two well known algorithms to generate traffic:

6.1.1 Using (u)sleep()

Using (u)sleep() to interspace the write() calls. This algorithm does not compensate for the time it takes to execute a write call. (See Program 6.1.) As a result this algorithm is always off by the amount of time it takes to execute the write() system call. Usually the way usleep() is implemented is by using the timer facility, of which there is only a single one. As such it renders the timer inoperable for any other use. The algorithm used in the test daemon is roughly equivalent to the inner workings of usleep().

The fundamental problem with an algorithm based on this mechanism is that each period some non deterministic time is added to the period due to the write() system call. Aside from that whenever a write() system call blocks because of external conditions this algorithm will not explicitly fail. It will simply continue, as such there is never any certainty that the traffic generated is actually what was wanted. (Unless of course a suitable network analysis method is coupled to this; a sniffer.)

6.1.2 Using select() or poll()

Using select() or poll(), instead of using sleep the socket itself is used to perform the sleep, by means of performing a blocking select() call on the socket descriptor. The timeout that is specified is the the spacing between writes. (See Program 6.2.) This algorithm frees up the timer for test house keeping purposes. This algorithm however still does not aleviate any of the problems that the previous mentioned algorithm suffers from.

Program 6.2 select() based traffic generation

```
...
while (...){

  write(socket, data);
  select(socket, period);

}
...
```

6.1.3 NetSpec's burst algorithm

Like mentioned before NetSpec uses an algorithm that is roughly equivalent with the inerts of usleep(). In Figure 6.2 a rough idea of the data/control flow is sketched. The UN*X timer facility is used to generate the period impulses, which drives directly the write() calls. The way we circumvent the problem of not accounting for time spend within the write() call is by arming the timer for the next period, while the write for the current period still has to be performed. This effectively replaces the non deterministic time that is spend within the write() call with a small deterministic amount of time that is used for arming the timer. The resulting problem of arming the timer while a write() call is being performed is that there needs to be a mechanism to avoid write() on write() calls. This is done by redirecting the signal catcher to a function that counts failed periods. This also provides us with a good idea if the traffic we generated actually complied. And if not, it will be indicated by a number of failed periods. Refer to Program 6.3 for some detail on the coding of the algorithm.

A slight optimization that can be done due to the fact that the timer facility within UN*X provides an automatic reload for periodic events is that even the call to arm the timer can

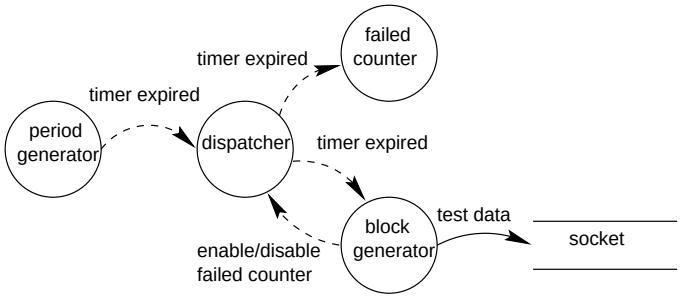

Figure 6.2: Burst algorithm

Program 6.3 Burst algorithm

```
void overflow (int aSignal) {
  ...
  singleShot (thisPeriod, overflow);
  ...
  failed++;
}
void burst(int aSignal) {
  ...
  singleShot (thisPeriod, overflow);
  ...
  bsdWriteSocket (sockDesc, getPtr2Buffer(), blockSize);
  ...
  singleShot (thisPeriod, burst);
}
...

do { /* main loop */
  pause();
} while (...);
...
```

be taken out. This results in an almost ideal Constant Block Rate (CBR) generator. The only problem the traffic generator is subject to is the load on the host machine, and any scheduling peculiarities. As a result of this there are two different functions for traffic generation, one for constant period (CBR) traffic generation, and one for random traffic generation. In case of the CBR generator the blocks themselves can still be according a random distribution, this is particularly usefull for MPEG type traffic. There is a small wrapper around both the CBR and random functions, such that only one access point has to be provided for both. The arguments provided can be used to select automatically the type of function that needs to be used.

6.1.4 NetSpec's burst queue algorithm

The previous section discussed the burst algorithm, there is one minor problem though. There are some random distributions that have fairly extreme peeks, most notably the FTP and WWW protocols. Using the previous algorithm this will inherently lead to an extraneous number of failed periods. The solution to this is to insert a queue between the period generator and the function that writes the blocks, see Figure 6.3 The algorithm is in matter of fact somewhat simpler than the previous one. There is still a reasonable indication if the traffic complied to the specified pattern, by means of monitoring the queue length. The program code is given in

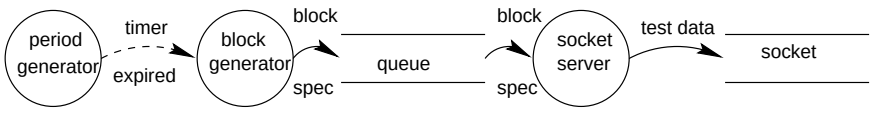

Figure 6.3: Burst queue algorithm

Program 6.3

6.1.5 Drawbacks of the burst and the burst queue algorithm

There are several drawbacks to the algorithms used. Performing a write() call within a signal handler is principally safe, however write() calls can block, and as such are somewhat risky to use.

Next to that it needs to be mentioned that there is compared to real dedicated traffic generators a whole range of problems that stem from the fact that a stock UN*X is used. Classifying and recognizing these problems is hard. A good rule of thumb is that for the CBR variety of the algorithm the period needs to be 2 to 4 times the system clock tick (10ms on most systems, ≈1ms on some others.) at the minimum. For the random variety of the algorithm it is slightly

Program 6.4 Burst queue Algorithm

```
void burst(int aSignal) {
  ...
  singleShot (thisPeriod, burst);
  ...
  addQEntry ((char*)&blockSize);
  ...
}
...
singleShot (thisPeriod, burst);
...
do {
  if (getQStatus() == EMPTY)
    pause();
  getPtr2QEntry((char**)&ptr2BlockSize);
  bsdWriteSocket (sockDesc, getPtr2Buffer(), *ptr2BlockSize);
  deleteQEntry();
  ...
}

...
```

worse, a factor 3 to 5 needs to be applied. The reason that there is no hard number for this, is that the algorithm works on anything from a Sparc 1 running SunOS upto a Dec Alpha 500 Mhz, running Digital UNIX. These machines have vastly different performance numbers and are invariably more or less sensitive to intensive CPU loads. One note that needs to be taken into account is that one can push the algorithms pretty close to the clocktick of the machine, but the variance of the traffic produced at that point is relatively high. *

6.1.6 Further considerations regarding random traffic generation

Since most traffic generation patterns are partially or completely based on well known random distributions one needs to take into account that these can generate some fairly extreme numbers. Next to that most protocols unlike TCP have upper limits on the number of bytes that can be written at once. Therefore it deemed neccesary to implement a segmenter, that could take a block and segment it into reasonable chunks without disturbing the traffic pattern. (As opposed to cutting it off at an unreasonably low maximum.) In general write calls of more than 64K are well over the boundary where there any impact can be attributed to the user/kernel context switching. Aside from an automatic segmenter certain protocols can be modeled by means of repetition. (FTP) As such there is also a variable repeat count for each period. This repeat count also facilitates the simple specification of more than 64K in case of UDP. Otherwise one would be limited to 64K per period, which considering the granularity of the period specification is not sufficient for current networking technology. The program shown in Program 6.5 shows part of the burst algorithm augmented with the repetition counter and the repetition counter.

6.2 Adding a protocol

Since the concept of unions of structures is used heavily throughout the code of the test daemon adding a protocol is relatively painless. Within this example we assume that we want to add a hypothetical protocol called “fuzzy”. †

The protocol type

Adding a structure to a union of structures consists of three steps:

*The comments made pertaining to the precision of the algorithm are empirical, there has been some experimentation[3] performed in order to derive the accuracy, however these experiments are not complete.

†It is the authors wish to once have the chance to rewrite the TCP flow control mechanism using fuzzy logic.

Program 6.5 Segmentation and repetition

```
    do { /* segmentation */
      ...
      segmentSize = ptr2Options->common.buffer.maxSize;
      do { /* repetition */
        ...
blockSize -= segmentSize;
bsdWriteSocket (sockDesc, getPtr2Buffer(), segmentSize);
        ...
      } while (blockSize > 0 ...);

    } while (--rptCnt > 0 ...);
```

Program 6.6 Protocol type

```
typedef enum protocolType_t {
  tcpProtocol = 1,
  udpProtocol,
  fuzzyProtocol
} protocolType_t;

typedef struct fuzzyOptions_t {
  protocolType_t type;
  int someFuzzyOption;
} fuzzyOptions_t;

typedef union protocolOptions_t {
  protocolType_t  type;
  tcpOptions_t     tcp;
  udpOptions_t     udp;
  fuzzyOptions_t   fuzzy;
} protocolOptions_t;
```

Program 6.7 Fuzzy parameter parser

```
%token FUZZY
%token SOMEFUZZYOPTION
...
protocol
   ....

   | PROTOCOL '=' FUZZY { defaultProtocolOptions(fuzzyProtocol); }
    '(' fuzzyOptions ')'
   ....

fuzzyOptions
  : fuzzyOptions ',' fuzzyOption
  | fuzzyOption
  ;

fuzzyOption
  : SOMEFUZZYOPTION '=' INTEGER
  {
    protocolOptions.fuzzy.someFuzzyOption = $$3;
  }
  ;
```

1. Create the data structure.

2. Add a new identifier to the enumeration type.

3. Add the structure to the union.

The 3 steps are shown in Program 6.6.

Adding fuzzy parameters to the parser and setting defaults

The parser sections for fuzzy would look as specified in Program 6.7 The setting of the defaults needs to be tacked onto the function defaultProtocolOptions, see Program 6.8

The fuzzy module

Next step is to provide a module that provides the following functions:

setupFuzzy

initializeFuzzy

openFuzzy

Program 6.8 Fuzzy defaults

```
void defaultProtocolOptions(protocolType_t type){
  ...
  switch (type){
  ....
  case fuzzyProtocol:
    protocolOptions.fuzzy.someFuzzyOption = 911;
    break;
  }
  ...
```

testFuzzy

closeFuzzy

teardownFuzzy

reportFuzzy

The respective prototypes are specified in Program 6.9

Adding the commands to the commands jump table

The last part is to add the fuzzy functions to the commands jump table in command.c. See Program 6.10 This concludes adding a protocol to the test daemon framework.

Program 6.9 Fuzzy prototypes

```
int setupFuzzy (ptr2TestOptions_t);
int initializeFuzzy (ptr2Address_t, ptr2ProtocolOptions_t,
                     ptr2TestOptions_t);
int openFuzzy (ptr2Address_t, ptr2ProtocolOptions_t);
int testFuzzy (ptr2TestSchedule_t, ptr2TestOptions_t);
int closeFuzzy (void);
int teardownFuzzy (void);
int reportFuzzy (ptr2ReportOptions_t, ptr2Address_t, ptr2Address_t,
                 ptr2ProtocolOptions_t, FILE*);
```

Program 6.10 Command interface jumptable entries

```
...
{
  setupFuzzy,
  initializeFuzzy,
  openFuzzy,
  testFuzzy,
  closeFuzzy,
  teardownFuzzy,
  reportFuzzy
}
...
```

6.3 Adding an arbitrary number generator (ANG)

Tests use the ANG's extensively, almost each test parameter can be specified as a random number. The implementation of a new ANG once again based on the union of structures concept.

ANG type

In this example we use the ANG ABR as an example. ABR is a hypothetical distribution that mimicks available bit rate. See Program 6.11. The major difference with the protocol type discussed before is that the common section consists of the parts given in Program 6.12. The only exception to the naming convention given in Program 6.12 is the constantDist options, here max is equal to the value, and the minimum is a "bogus" value. In case of a constant there is no maximum or minimum, however for buffer allocation purposes it is handy if the value of the constant is mapped into the same space as where for other distributions the maximum is kept.

Defaulting ANG's and parsing ANG's

In the parameter parse section, see Program 6.13 the assumption is made that the ABR options only include the standard options, see Program 6.14 no more parameters can be specified. Within the commonoptions there is another section provided for random number generators, various implementations for uniform random number generators can be provided, currently only the buildin (The rand48 set of routines) UN*X random number generator is used. The extension of the random number generator is similar to extension of an ANG or a protocol.

The defaults need to be provided in the function defaultDist in defaults.c, see Program 6.15.

Program 6.11 ANG type

```
typedef enum distType_t {
  constantDist = 1,
  uniformDist,
  ...
  WWWItemSizeDist,
  ABRDist
} distType_t;

typedef struct ABROptions_t {
  distType_t type;
  long max;
  long min;
  generatorOptions_t generator;
} ABROptions_t;

typedef union distOptions_t {
  distType_t type;
  commonAngOptions_t common;
  constantOptions_t constant;
  ...
  WWWItemSizeOptions_t WWWItemSize;
  ABROptions_t ABR;
} distOptions_t;
```

Program 6.12 Common ANG type

```
typedef struct commonAngOptions_t {
  distType_t type;  /* common denominator */
  long max;
  long min;
  generatorOptions_t generator;
} commonAngOptions_t;
```

Program 6.13 Parsing ang's

```
%token SOMEABROPTION
...
distOptions
  ...
  | SOMEABROPTION { defaultDist (ABRDist, ptr2DistOptions); }
    '(' ABRoptions ')'
  ;
  ...
ABROptions
  : ABROptions ',' ABROption
  | ABROption
  ;
ABROption
  : commonDistOptions
  ;
```

Since ABR does not have any parameters out of the ordinary, the standard settings for these suffice, and nothing has to be done.

Adding the code to ang.[chi]

Most random distributions are either table based, or somewhat complex formulae. As such there is no need to create seperate functions, rather a couple of lines of c code within the function suffice most of the time. See Program 6.16. If a table is considered, than it is slighly more eloborate. Tables can be either statically or dynamically allocated, they need to be in memory for the simple reason of speed, for each call to the number generator we can not expect to have

Program 6.14 Parsing common ANG parameters

```
commonDistOptions
  : generatorOptions
  | MAX '=' INTEGER
  {
    ptr2DistOptions->common.max = $3;
  }
  | MIN '=' INTEGER
  {
    ptr2DistOptions->common.min = $3;
  }
  ;
```

Program 6.15 Defaults for an ANG.

```
void defaultDist (distType_t type, ptr2DistOptions_t ptr2Options){
  ...
  ptr2Options->type = type;
  ptr2Options->common.max = 0;
  ptr2Options->common.min = 0;
  ptr2Options->common.generator.type = buildinGen;
  ptr2Options->common.generator.common.seed = tv.tv_usec;
  switch (type) {
  case constantDist:
  case uniformDist:
  case ABRDist:
    break;
  ...
```

enough time to do extensive system calls. Currently the telnet and ftp distributions use table driven generators. The structure number_t is used both as a passing mechanism, and a state preservation method. In between consecutive calls, state based random number generators need to maintain their state, this is done by means of values within the structure.

This concludes the implementation of a new ANG.

6.4 Adding a test

Adding a test is slightly less complicated compared to adding a protocol. Tests can be added on a per protocol basis, however all of the tests are implemented for both UDP and TCP. Here we will consider the implementation of the "ping/echo" test. The ping/echo test is a verbose version of a request response test. (It has more similarities to ping echo than it does towards request response.) A request response test in the broadest sense is where the communication peers exchange traffic, unlike a transmit/receive test. *

The test type

Adding a test type structure to the framework is almost synonymous to adding a protocol type structure. Basically ping and echo are each others counter parts. Both share the same data structure, it is usefull if each knows what the counterpart is about to embark on.

Adding a structure to a union of structures consists of three steps:

*The ping/echo test is currently partially implemented within the test daemon, though not functional.

Program 6.16 Coding an ANG.

```
...
typedef struct number_t {
  distOptions_t userOptions;
  ptr2Stats_t   ptr2Stats;
  long sceneLength;
  long currentState;
  long tempBuffer;
} number_t;

typedef number_t *ptr2Number_t;
...
INLINE long getValue (ptr2Number_t ptr2Data){
  ...
  switch (ptr2Data->userOptions.type){
  ...
  case ABRDist:
    value = (ptr2Data->userOptions.ABR.max +
             ptr2Data->userOptions.ABR.min) / 2;
    break;
  ...
```

1. Create the data structure.

2. Add the two new identifiers to the enumeration type.

3. Add the data structure to the union.

The 3 steps are shown in Program 6.17. The slightly more complex variation that the test type shows is that there is more to the common part of the union than the just the enumeration type. For all tests it can be assumed that data buffers are required to generate data. Aside from that the time stamping facility is set up in such a way that it is common to all tests too. The common part of the union is given in Program 6.18.

Defaults and parser additions

Due to the fact that the tests have common parts to the parameters, the parser is also slightly different, Program 6.19 shows the common part of the parameter parser. The actual parser, see Program 6.20 is similar to the previous shown parser for the protocol parameters.

The parser is slightly more complicated since it uses the ANG's extensively. Since the ANG's can be the same for each test parameter, there is a standarized ANG section within the parser. However to get the ANG section to refer to the right data structure without having

Program 6.17 Test type

```
typedef enum testType_t {
  sinkTest = 1,
  fullTest,
  burstTest,
  burstQTest,
  pingTest,
  echoTest
} testType_t;

typedef struct pingOptions_t {
  testType_t type;
  stampOptions_t stamp;
  bufferOptions_t pingBuffer;
  bufferOptions_t echoBuffer;
  distOptions_t pingSize;
  distOptions_t pingRptCnt;
  distOptions_t echoSize;
  distOptions_t echoRptCnt;
} pingOptions_t;

typedef union testOptions_t {
  testType_t type;
  commonTestOptions_t common;
  sinkOptions_t   sink;
  fullOptions_t   full;
  burstOptions_t  burst;
  burstQOptions_t burstQ;
  pingOptions_t   ping;
  pingOptions_t   echo;
} testOptions_t;
```

Program 6.18 Common test options

```
typedef struct commonTestOptions_t {

  testType_t type;

  stampOptions_t stamp;
  bufferOptions_t buffer;

} commonTestOptions_t;
```

Program 6.19 Common test parameters

```
commonTestOptions
  : ALIGNMENT '=' INTEGER
  {
    testOptions.common.buffer.alignment = $3;
  }
  | OFFSET '=' INTEGER
  {
    testOptions.common.buffer.offset = $3;
  }
  | BUFFERSIZE '=' INTEGER
  {
    testOptions.common.buffer.maxSize = $3;
  }
  | STAMPS '=' INTEGER
  {
    testOptions.common.stamp.stamps = $3;
  }
  | scheduleOptions
  ;
```

to copy it from a dummy data structure here we use a pointer that the ANG section refers to. Each of the parameters assigns the pointer to point to its respective data structure.

The extension to the defaults function is straightforward, see Program 6.21

Adding a test function

Assuming that this is added to an existing protocol module, the program in Program 6.22 shows the addition of the test function within the main test function. In this case a switch statement is used, not a jump table, a jump table won't buy us anything but a little speed which at this point in the code is of no concern. Of course the functions need to be implemented at this point.

This concludes the implementation of various extensions that can be done to the test daemon.

Program 6.20 Test parameters

```
%token PING
%token ECHO
...
test
  ...
  | TESTTYPE '=' PING { defaultTestOptions(pingTest); }
    '(' pingOptions ')'
  | TESTTYPE '=' ECHO { defaultTestOptions(echoTest); }
    '(' echoOptions ')'
  ...
pingOptions
  : pingOptions ',' pingOption
  | pingOption
  ;
pingOption
  : PINGSIZE {ptr2DistOptions = &testOptions.ping.blockSize }
    '=' distOptions
  {
    if (!testOptions.common.buffer.maxSize)
      testOptions.common.buffer.maxSize =
testOptions.burst.blockSize.common.max;
  }
  | commonTestOptions
  ;
echoOptions
  : echoOptions ',' echoOption
  | echoOption
  ;
echoOption
  : ECHOSIZE {ptr2DistOptions = &testOptions.echo.blockSize }
    '=' distOptions
  {
    if (!testOptions.common.buffer.maxSize)
      testOptions.common.buffer.maxSize =
testOptions.burst.blockSize.common.max;
  }
  | commonTestOptions
  ;
```

Program 6.21 Test defaults

```
void defaultTestOptions(testType_t type){
  ...
  switch (type){
  ....
  case pingTest:
    testOptions.ping.pingSize.type = constantDist;
    testOptions.ping.pingSize.value = 32768;
    ...
  case echoTest:
    ...
    break;
  }
  ...
```

Program 6.22 Adding the new test function

```
int testTcp (ptr2TestSchedule_t ptr2Schedule,
             ptr2TestOptions_t ptr2Options){
  ...
  switch (ptr2Options->type){
  ...
  case pingTest:
    return testPing (ptr2Schedule, ptr2Options);
  case echoTest:
    return testEcho (ptr2Schedule, ptr2Options);
  ...
```

Chapter 7

Characteristic experiments and results

The results discussed here were obtained doing various experiments, and the most interesting details that came out of those are discussed.

7.1 Detailed sattelite analysis

Figures 7.1 and 7.2 show the throughput of a one way transmission, full speed over the ACTS sattelite link using TCP/IP. The plots shown were generated using the time stamp information that the test daemon can provide. Time stamps of each and every read and write call can provide fairly detailed information, not as good as network sniffing tools like tcpdump and snoop though. The key advantage of user level time stamping of read and write calls is that it is highly portable, the impact on the performance is fairly minimal, whereas network sniffing tend to have quite a bit of impact on the overall performance of the machine under test. Certainly in ATM environments it is hard to do any kind of sniffing, since it is connection oriented, and as such the sniffing has to take place on the same host as the actual test is being run.

The conditions under which the experiment was was as follows:

- The transmitter end was a DEC alpha model 3000/500, 192 Megs of memory running at 150Mhz. The OS was Digital Unix version 4.0 release a. (rev 386).
- The receiving end was a Sun UltraSparc model 2, 256 Megs of memory running at 200 Mhz. The OS was solaris 2.5.1 with a TCP windows scaling enabled driver patch.

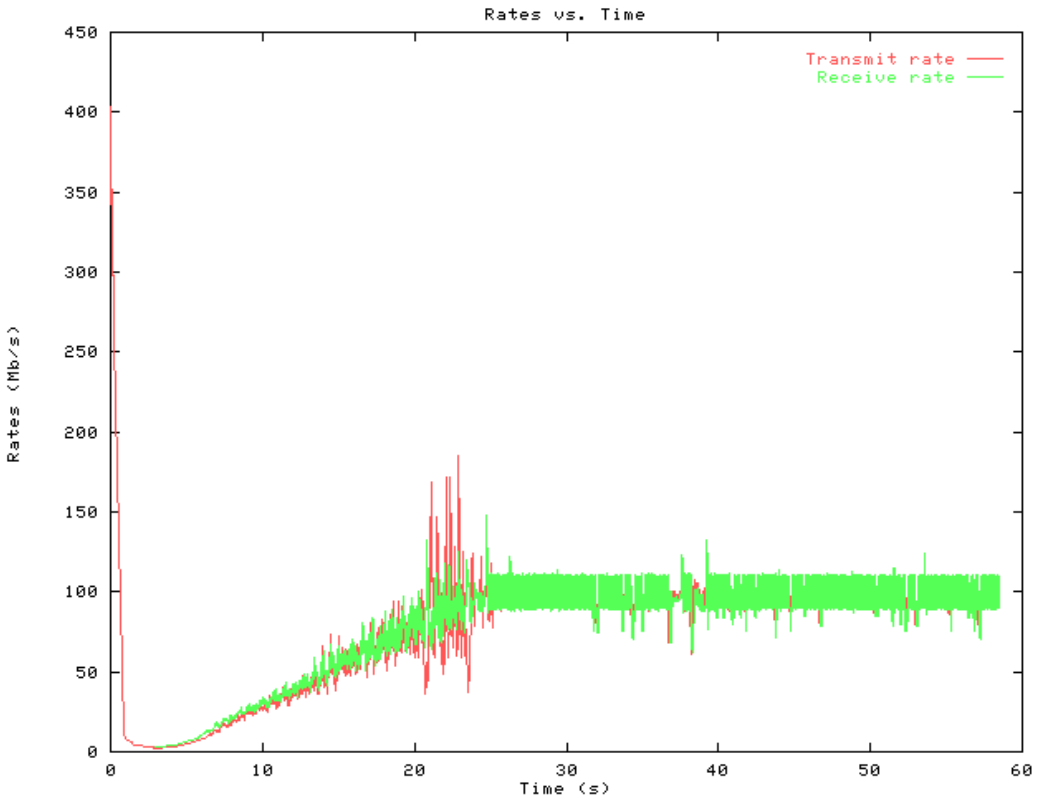

Figure 7.1: Transmit and Receive rates

- The roundtrip time is about 500ms
- Due to clock granularity the samples are averaged using a running average with a window of 50 samples. This distorts the picture considerably, any instantaneous effects are averaged out.
- This experiment was performed early May 1997.

Figure 7.1 shows the rate, vs time. The thing that appears to be odd is the fact that initially the rate is about 400 Mb/s. This is well above line rate, in fact it is about the rate a DEC alpha model 3000 can copy data within memory *. The reason for this is fairly simple to explain. The TCP/IP window simply turns out to be a simple FIFO in memory. In order to potentially fill the an 155 Mb/s OC-3 pipe at the given roundtrip, one needs $500ms * 135000000Mb/s/8bytes = 8437500bits$ worth of window, rounded up this requires a 10 Mbyte buffer within memory.

*The 3000 series use a double banked memory configuration, enhancing the path to a 256 bit wide path, versus the competition at that time using either a 32 bit or a 64 bit wide path to memory.

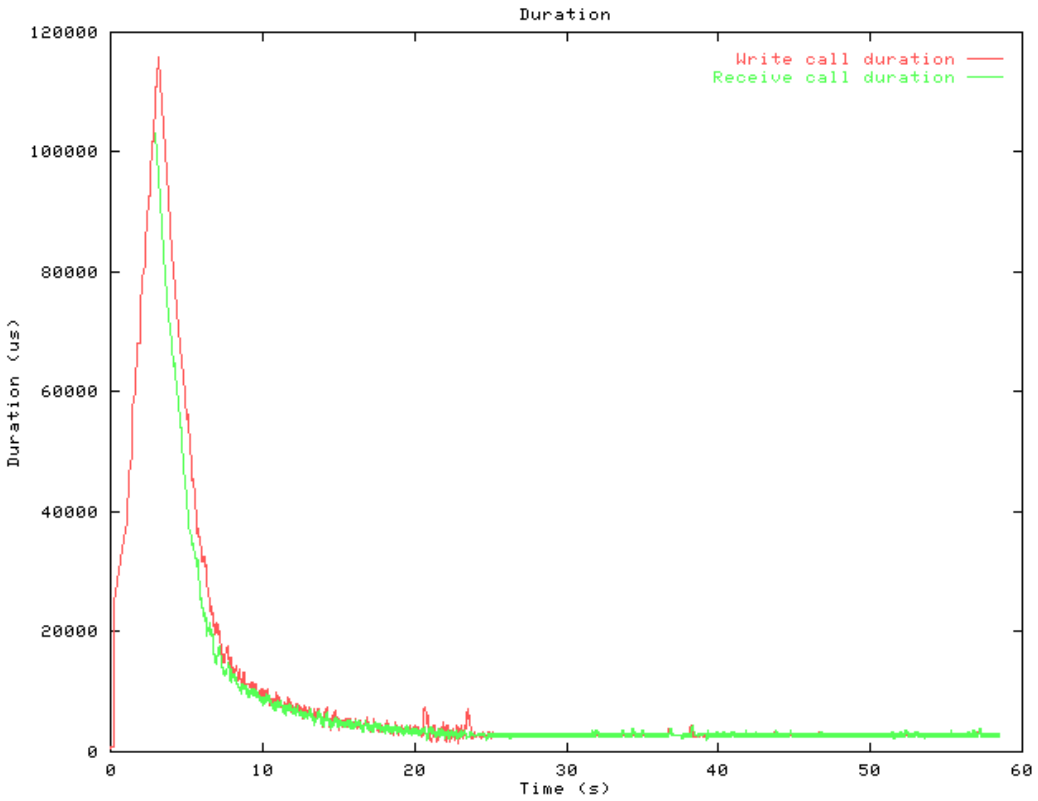

Figure 7.2: System call duration

What happens in the first hundred milliseconds is that the application, NetSpec, sends out data as fast as the kernel it can copy it into the buffer, since the buffer is about 10 Mbyte this results in a large spike. Doing the math, assumming a memory to memory copy rate of 400 Mb/s, that results in 50 Mbyte/s, given a 10 Mbyte buffer, that would take about 200 ms to fill up, If one looks at Figure 7.2 the first 200ms are indeed a flat line.

Aside from the anomaly of the exacerbated effect of a extreme large window the time it takes to reach stable state takes about 25 seconds. This is a very good indication that running any type of short duration test over long bandwidth delay product links is bad practice. It also advocates for a parameter to be implemented in the test daemon that allows the user to specify more selective measurements, implementation wise this is hard though.

The throughput that the test daemon reported was around 75 Mb/s, this is abviously not wrong in terms of average, but in reality as soon as the connection reaches stable state this appears to be around 100 Mb/s. This is still not the theoretical maximum of $\approx$ 134 Mb/s. The reason for this is the way the FIFO at the transmit side is implemented in the kernel, in essence

it is a linked list of small buffers, traversing the linked list takes time, the problem is that at one end the FIFO is drained and at the other end the fifo gets filled. The net result is that the CPU spends most of its time traversing the linked list of buffers back and forth *. A solution to this problem would be to introduce a head an tail pointer in the FIFO management code, that way the CPU could directly jump to the beginning and end of the list. The problem would be that out of sequence delivery of ip packets results in problems once again with this approach. (In case of a non routed environment with a single path like this one that is highly unlikely though.)

7.2 Small buffers

This experiment[7] was performed by Luiz DaSilva in early summer of 1996. Before the experiment was performed the suspicion was that limited buffer space on one of the switches that was part of the AAI at that time had a severe impact on some of the traffic. In figures 7.3 and 7.4 the thruput vs. burst size is shown, on a linear and on a log scale respectively. The major problem of AAI at that time was that the core of AAI consisted of a DS3 based network, and the edges all ran at OC3 rates, this resulted in a 4 to 1 rate mismatch. By using a constant stream of UDP/IP traffic with relatively small bursts, slowly increasing the burst size with each experiment it showed that the buffer started dropping packets. Then in the reverse direction the same was tried, and it resulted in a margin that was magnitudes higher than the forward direction showed.

This experiment showed clearly that with fairly simple measurements based on simple thruput numbers fairly detailed network characteristics can be determined, the reason the choice was made for UDP is that way the user has an exact idea of what went over the wire, whereas TCP/IP will initiate successfull retransmits, that will inherently obscure the picture. This is almost the sole reason that advocates for using UDP with carefull monitoring the loss for setting baselines for maximum performance figures. Also one key advantage is that whenever UDP is used that whatever propagation delay the packet experienced can entirely be attributed to protocol stacks and networking equipment, whereas any conclusions towards that in case of TCP would at least be questionable unless the places where retransmits occured are known, and eliminated out of the measurements.

Whenever TCP/IP was used on this particular link, and the traffic was not modulated the throughput was well below expected values. Conclusion is that TCP/IP requires quite a bit

*This explanation was provided by Hall Murray at Digital Systems Research Center.

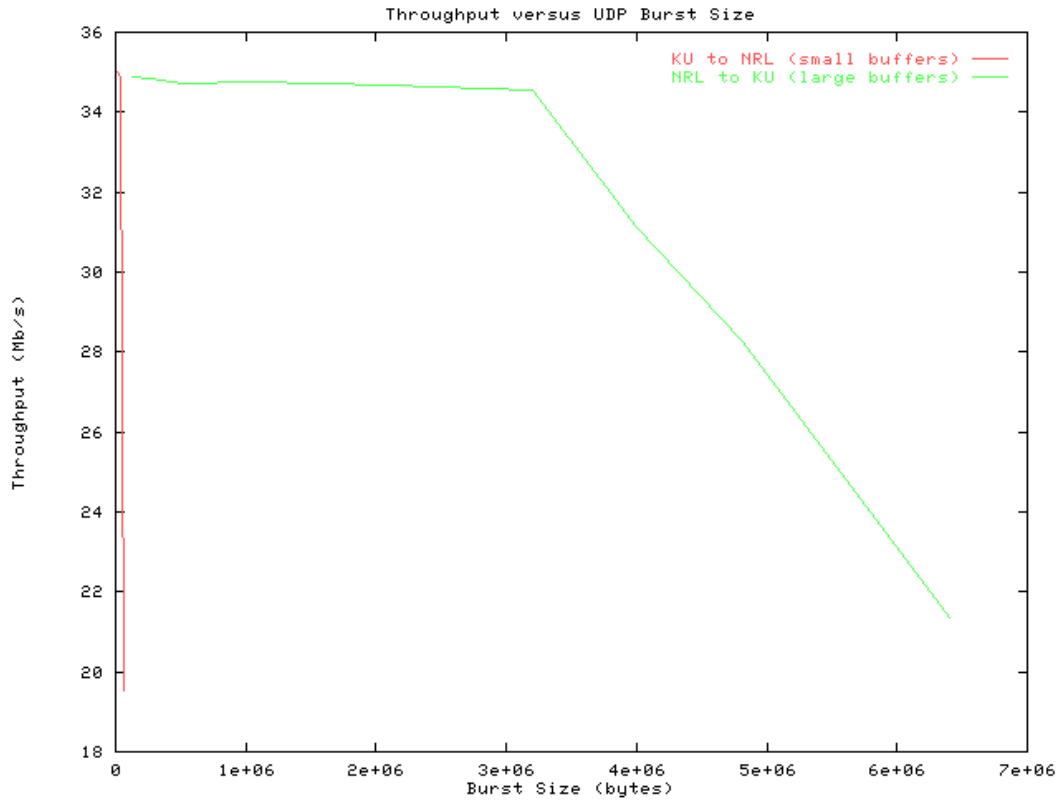

Figure 7.3: Switch buffer on lin scale

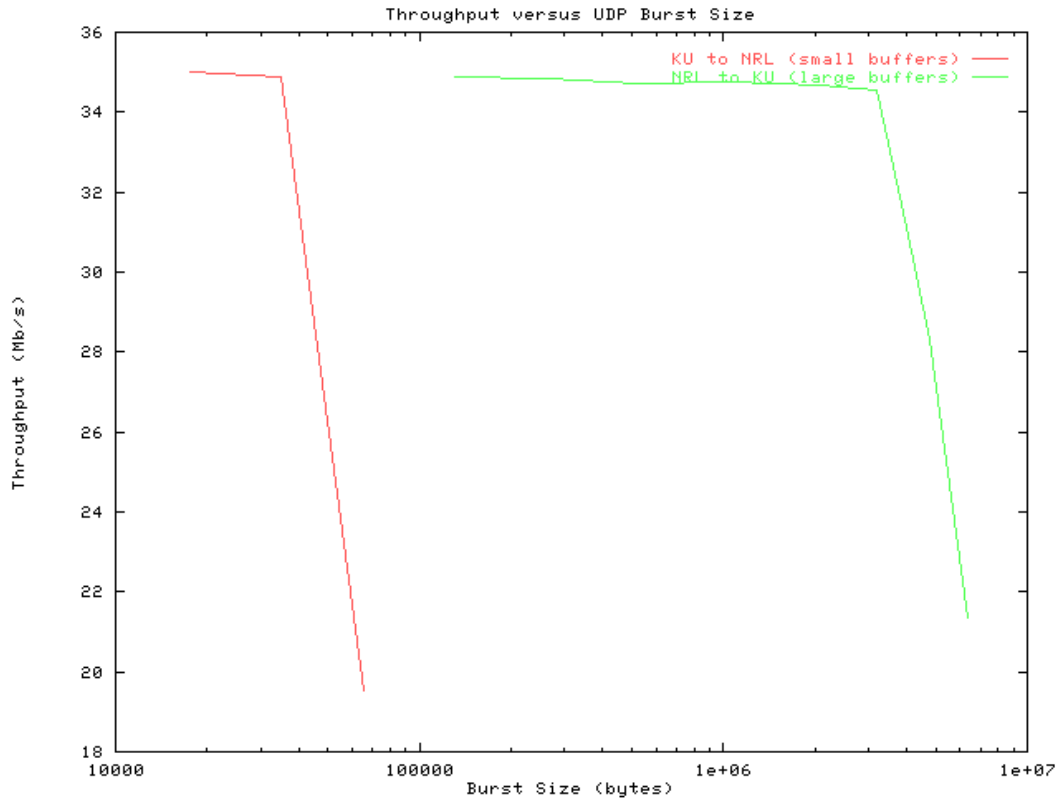

Figure 7.4: Switch buffer on log scale

of network resources in order to perform decently under circumstance like this. In order for TCP/IP to overcome a rate mismatch, it needs likely buffer space in excess of half to three quarters of the size of the window at the point where the rate mismatch occurs. It also depends on how the traffic got affected by other traffic before it arrived at the point of the rate mismatch.

The specifics of the problem that occured here was that in one direction the mismatch occured at a first generation network module in a Fore ASX100 switch. This particular switch and net module had a fixed buffer size of 256 cells per port. One classical IP mtu (media transfer unit) takes 9180 bytes including headers, this results in 192 cells, that is leaves about a quarter of an mtu left in terms of buffer space. Likely every other TCP/IP packet that gets transmitted gets dropped. This results in very poor performance. NetSpec already showed that it was possible by using the user level pacing algorithm that by means of fairly simple flow control this problem could be solved. The reverse direction was using revision C network modules on a Fore ASX200BX switch, which has roughly 13000 cell buffers that are shared between 6 ports assuming a DS3 module. If one single port gets allocated the entire amount of cell buffers, that would result in about 65 classical ip mtu's, 600Kbytes worth of buffering.

Chapter 8

Recommendations

Though NetSpec has been successfully used, in fact it was essential in conducting ATM WAN measurements for the AAI research effort, there are however quite some problems and caveats. In order to distinguish clearly between programming flaws and limitations identified first the current bugs will be discussed, followed by the proposed extensions.

8.1 Bug fixes and improvements

At the point of this writing there is quite an extensive list of problems identified. There can be made three classes of improvement:

1. Programming bugs
2. Porting problems
3. Improvements

Programming bugs

Programming bugs won't be discussed here mostly because they are relatively volatile in nature, and are often fixed as soon as they are discovered.

Porting problems

1. Asynchronous IO, the definitions of how to perform asynchronous are vague and differ considerably from one system to another
2. Non-local jumps, using setjmp() is not functional on solaris 2.5.

3. Polled IO (select()) on a blocked accept() call does not work under linux. (version 2.0 kernels.)

The first two problems inhibit the use of any type of asynchronous intervention with the leaves. Sofar this has not posed to be a major problem, other than that tests on occasion “hang”. The third problem causes somewhat classic (Similar problems were prevalent in NetSpec version 2.0) “hanging” problems.

Improvements

1. SNMP expressions can be integrated natively instead of reverting to using string constants.
2. Instead of using a hardcode value for the default port number, it should be derived out of /etc/services.
3. Return codes of various functions should be less binary, currently most functions indicate failure or not, no provision has been made to enable functions to indicate a warning condition.
4. There should be a provision within the reverse parsers to incorporate reserved characters by means of escape codes.
5. Consolidate the various leaves into one source tree.
6. Within the server it should be possible to incorporate a protocol monitor fairly easily, this would be a considerable improvement of debugging facilitities.
7. The slave control protocol parser could be eleminated by using a state based lexical analyzer. (This is currently being worked on.)
8. Instead of using the timer within the test procedures a busy wait type test could also be used. This would be usefull for dedicated test boxes where precision is of very high importance. (Conformance testing maybe.)

8.2 Proposed extensions

8.2.1 Scaling extension and improvement

Currently the control framework does not scale upto the point it is entirely constrained by the operating system. The main reason for this is that the control framework currently does

not exploit the inherent parallelism within the structure very well. This is partially due to the somewhat unclear definition of the various commands as discussed in Section 3.2.4. The clear solution would be to define the meaning of each of the commands within the control framework, but the problem with this is that there is a symbiotic relationship between the definition and how much parallelism can be achieved.

Somewhat orthogonal to the previous issue, in case of serial{} execution constructs a more conservative approach in creation and parsing the script could help scaling considerably. Fundamentally any serial type execution does not have the need to create the entire control frame work on startup, only the parts that are needed to start the first enclosed block within the serial block is sufficient. Here is again where the use of the "recursive descent" parsing technique appears to be a problem. A simple solution would be to simply temporarily store the blocks that are not needed at the point of execution. And here is where the both the relative merit of the use of LR(1) parsers shows. A LL(1) parser would simply start executing as soon as it read the first block, leaving the syntax checking for later, which in case of syntax errors would only show after partial execution of the test. An LR(1) parser on the other hand has to evaluate the entire block in order to execute parts of it, which implicitly will do the syntax checking.

Another challenge that is present is the problem of the reported data volume, most measurements performed on high speed networks can produce high volumes of data. Since all the reports are fully text based the reports could benefit substantially from the use of data compression[12]. The cautionary note to make here is that only the reports can be compressed, compression of each of the commands and or replies will very likely result in very slow performance and considerable loss of accuracy of control.

The way the test daemon is implemented it does not allow for external synchronization. The problem that occurs with longer ($>$ 10 minutes) duration type tests is that the timers that are maintained within each instantiation of the test daemon will get slightly out of sync. Most of this is attributable to the fact that UN*X only has one timer available for use within a user level process. In order to perform both traffic generation and overall timing of the test the same timer needs to be used. The requirement of being able to generate traffic more precisely than to be able to time the test accurately is considered to be more important. Another somewhat related problem is that for tests longer than approximatley 36 minutes the resolution of the used data type (long) is exceeded. This is due to the fact that the tests are timed in microseconds, there is no need for that precise a resolution, but it simplifies the code considerably. Note needs to be made that this is only valid for 32 bit architectures, where a long only has 32 bits, on 64 bit architectures a long is 64 bits, resulting in a maximum test length of approximately

81 weeks. Instead of relying on asynchronous timers it would be usefull to implement external synchronization. This can be achieved relatively easily by using two facilities asynchronous IO and the directed broadcast mechanism described in Section 3.2.5.

8.2.2 Integration of post processors

The user interface (netspec) currently only has pre processing capabilities, this is achieved by using either the C pre processor or M4, which is a more universal preprocessor. The post processing will almost inherently involve some sort of parsing. It can however be as simple as a couple commands in a shell script. As such it would be usefull to provide a shell that can be used to insert a post processor. Likewise it would be usefull to provide a mechanism for a custom pre processor. These are relative minor efforts. Along with the provision of the post processor there needs to be a standardized way that all leaves format their output. This way a (semi-) standarized post processor could be written.

8.2.3 Request response tests

There is real need for a somewhat advanced request response test. Currently the test daemon only has the capability of generating uni directional traffic. A simple request response test will not be much more usefull than a simple estimate of latency. A somewhat more eloborate implementation could be used to mimic known traffic patterns, and could even go as far as mimicing protocols. The provision is that it needs to have some simple timers and incorporate feature of a windowing protocol like TCP for example.

8.2.4 Intermediate reports

Another usefull feature that would be mainly of use for monitoring purposes is intermediate reports. The framework currently facilitates the use of intermediate report replies. The problem is that most daemons are so intrically involved in performing the tests that intermediate reporting is hard without disrupting the test. As mentioned before UN*X does only have one timer available at the user level this limits capabilities severly, once again a timer could be used to initiate the generation of intermediate reports.

8.3 Language Extensions

One of the things that NetSpec currently lacks is iteration and conditional constructs. The fundamental hard part about any of these constructs is how to implement this. In this section various suggestions will be made in order to accomplish this, and its potential use.

One of the major caveats that hinders the implementation of looping - and conditional constructs stems from the spit personality of the framework. Essentially there are two disjoint places within the framework where constructs need to be interpreted; the controller and the leaves. In terms of syntax this leads to the problem that the language as is constrains the way we can express complicated constructs like loops etc. Associated with that is the problem that at both the controller and the leave level the constructs need to be syntactically consistent, otherwise confusion on the users part could be expected. The two previous problems present the main design challenge.

Simple distinctions can be made between the various constructs:

Loops A straight forward for loop with one iterated variable.

Conditionals An if-then-else statement.

Conditional loops

8.4 Enhancing security

The major problem with NetSpec is that it can be a very powerfull tool, upto and including network sniffers. This is by far considered the ultimate security breach. As such it is almost mandatory for NetSpec to implement some measure of security. Currently NetSpec does not allocate a port within the reserved range of available port numbers. A first measure would be to get a port assignment for NetSpec within the reserved range. Most firewalls will by default block most reserved ports unless explicitly permitted, whereas in the unreserved range (portnumbers $\geq$ 1024) most firewalls allow most traffic to pass.

Next to firewalls tcp wrappers provide a basic level of security, NetSpec can be put under control of a tcp wrapper[33]. Or NetSpec could be extended to include similar features. This prevents from superficial attacks, but not from determined attacks, as soon as an attacker impersonates an ip address that is allowed according the tcp wrapper the wrapper does not impact the use.

The previous two suggestions provide a basic level of security, but do not provide any way of authenticating the users. The only real way of solving this problem is by use of a public / private key encryption scheme, such as secure shell (SSH)[34] implements. This would also enable NetSpec to grant selective privileges, based on identity. The only drawback of this approach is the key management that inevitably is associated with public private key encryption schemes. SSH implements a basic keyserver that could be utilized for this purposes, another more global solution would be the use of the existing pretty good privacy (PGP) key servers.

Chapter 9

Conclusions

Both NetSpec 2.0 and 3.0 have been extensively used. Its use has revealed some remaining flaws and shortcomings. There is still quite a bit of work that can be done to enhance NetSpec as is explained in Chapter 8. The NetSpec framework has a lot of potential, and along with tools as NetPerf[14] it can become a major player as a network testing and analysis tool.

9.1 Success

It is somewhat questionable on how to measure success, NetSpec has proven to be successful. One major accomplishment achieved with NetSpec was performed during the summer of 1997 by Beng Ong Lee; 772 successful experiments on AAI, 150 hours total, about 1 Gigabyte of data that took 24 hours to process.[17] These experiments involved 10 hosts distributed over a wide area ATM network spanning the United States from east - to west coast.

Other projects that NetSpec has been used for, Firewall testing[27] performed by Steven Pennington and TCP Pacing[3] done by Brian Buchanan. In both these projects NetSpec was an instrumental tool that simplified the problem of extensive testing. Most recently NetSpec was and is used by Charalambous P. Charalambos (Pambos) extensively for the ACTS ATM Internetwork sattelite tests.[4]

For the currently ongoing MAGIC-II project NetSpec is being tailored to function within an active network framework. This work is done by Yulia Wijata who also co-wrote some of the parts of NetSpec, and served as one of the most resourceful debugging resources.

The current version of NetSpec was made available to the general public. To date about 190 organizations have down obtained copies of NetSpec.

9.2 Software engineering considerations

Developing software in an academic environment is not easy. There are usually no established software engineering practices, and as such the student is left to develop their own. NetSpec's development and design over time have been a useful experience in trying to establish an software engineering practice that could coexist with the pressing needs for the tool itself.

Various students within the Informations and Telecommunications Technology Center (ITTC) have successfully and independently implemented daemons that run under the NetSpec framework, most notably:

SNMP daemon Yulia Wijata

ATM call generator daemon Shyam Murthy

ATM Traffic Reference Source ARTS daemon Balaji Srinivasan

Data stream driver interface daemon Yulia Wijata

Corba Performance daemon Anil Gopinath

Each of these people required very little help ($<$ 30 minutes from the author), even though there was little or no documentation available on the implementation. This clearly showed that the software was written is a readable and understandable way.

9.3 Testing considerations

NetSpec by far is not as stable as we like it to be, this is partially due to the subtle differences between various varieties of UN*X. And some of the idiosyncrasies of the algorithms used within the test daemon are predominantly responsible for this. Somewhat hardened users of NetSpec tend to be able to recognize some of the problems, but most beginners are struggling the most with these obstacles.

As mentioned before, documentation and enhancements of NetSpec itself will accelerate the acceptance and distribution of NetSpec.

Appendix A

Module Structure

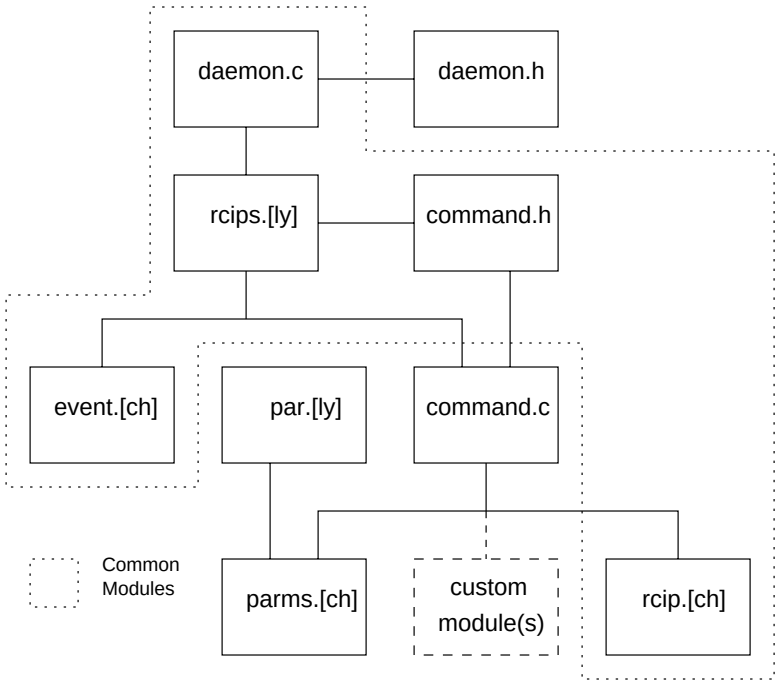

Figure A.1: Tap / general daemon module hierarchy

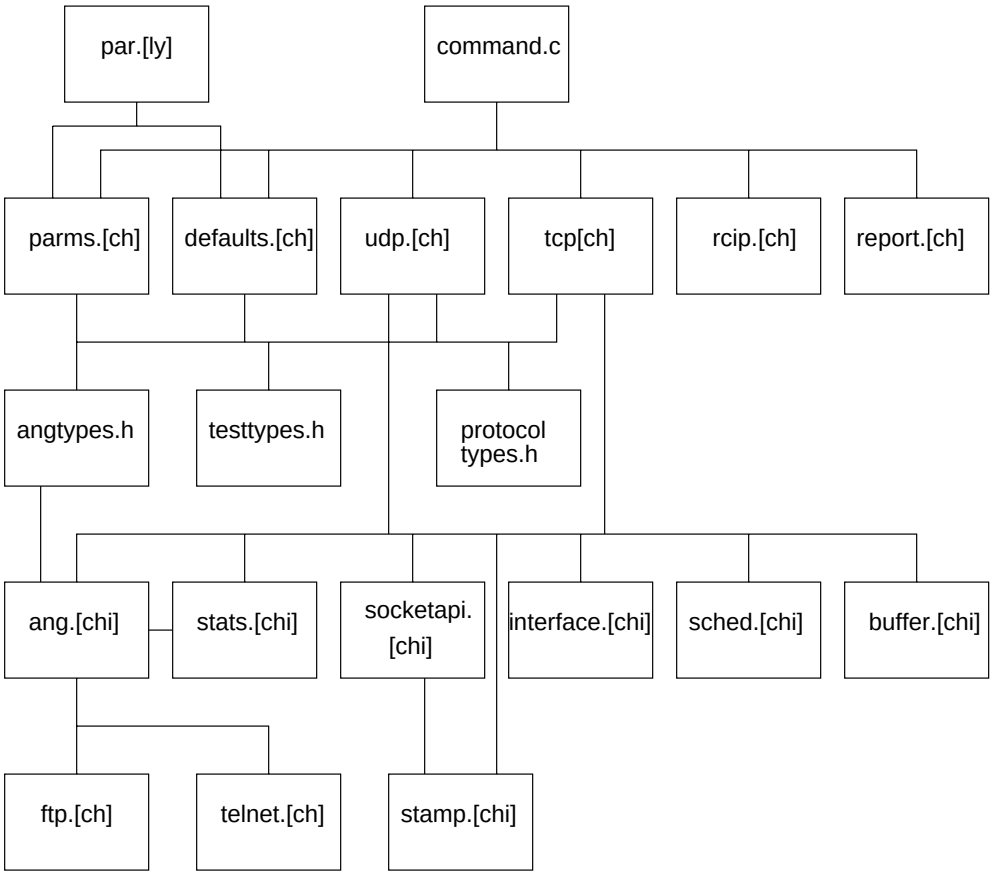

Figure A.2: Test daemon module hierarchy

Bibliography

[1] Aai, acts atm internetwork. http://www.arl.mil/AAI.

[2] Morris L. Bolsky. *C Programmeurs handboek*. AT&T Prentice Hall, 1985. Translated from english.

[3] Brian Buchanan. Header compression on atm networks. Master's thesis, University of Kansas, January 1998.

[4] et al. Charalambous P. Charalambos. Experiments and simulations of tcp/ip over atm over a high data rate satellite channel. Technical report, University of Kansas, February 1998.

[5] Richard Stallman Charles Donnelly. *Bison*. Free Software Foundation, 1992.

[6] Yourdon Constantine. *Structured Design*. Yourdon Press, Prentice Hall, 1978.

[7] Luiz A. Dasilva et al. Atm wan performance tools, experiments, and results. *IEEE Communications Magazine*, 35(8):118–125, August 1997.

[8] Jeffrey E.F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, Inc., July 1997.

[9] IEEE. Posix signals. http://www.pasc.org/abstracts/posix.htm.

[10] Internet super server (inetd) man pages. see man inetd.conf & man inetd.

[11] J. Postel J. Reynolds. Rfc1700 assigned numbers & std0002. ftp://ftp.isi.edu/in-notes/iana/assignments/.

[12] Mark Adler Jean-loup Gailly. zlib. http://www.cdrom.com/pub/infozip/zlib/.

[13] Doug Brown John R. Levine, Tony Mason. *Lex & Yacc*. O'Reilly & Associates, Inc., February 1995.

[14] Rick Jones. Netperf. http://www.cup.hp.com/netperf/NetPerfPage.html.

[15] Roelof J.T. Jonkman. The design and implementation of netspec. University of Kansas, Informations and Telecommunications Sciences Laboratory, May 1995.

[16] Roelof J.T. Jonkman. The specifications for netspec. University of Kansas, Informations and Telecommunications Sciences Laboratory, May 1995.

[17] Beng-Ong Lee. Wide area atm network experiments using emulated traffic sources. Master's thesis, University of Kansas, January 1998.

[18] Steve Maguire. *Writing Solid Code.* Microsoft Press, 1993. Book was written using tex.

[19] P. Rhagavan Menon. Large scale distributed real time computation: A time driven framework. Master's thesis, University of Kansas, January 1998.

[20] Sun Microsystems. Rfc's 1014, 1057, 1094 and 1790. ftp://ftp.internic.net/rfc/rfcxxxx.txt.

[21] Terry Slatery Mike Muuss. Ttcp. http://ftp.arl.mil/pub/ttcp, October 1985. Modified at Silicon Graphics in 1989.

[22] David L. Mills. Network time protocol (version 3) specification, implementation and analysis. ftp://ftp.internic.net/rfc/rfc1305.txt, March 1992.

[23] David L. Mills. Improved algorithms for synchronizing computer network clocks. *IEEE/ACM Transactions on Networking*, 3(3):245 254, June 1995.

[24] Barbara A. Denny Paul E. McKenney, Danny Y. Lee. Traffic generator software release notes. Technical report, SRI International, Menlo Park CA, November 1995.

[25] Stephen J. Mellor Paul T. Ward. *Structured Development for Real-Time Systems volumes 1,2 and 3.* Yourdon Press, Prentice Hall, 1985.

[26] Vern Paxson. *Flex, version 2.5.* The Regents of the University of California, 1990.

[27] Steven G. Pennington. Gauntlet firewall performance benchmarks. http://www.ittc.ukans.edu/spenning/firewall.

[28] S. Deering R. Hinden. Rfc 1884. ftp://ftp.internic.net/rfc/rfc1884.txt.

[29] Cray Researh. Nettest. ftp://ftp.sgi.com/pub/src/nettest.

[30] Scott W. Shumate. A performance analysis of an off-board signaling architecture for atm networks. Master's thesis, University of Kansas, June 1996.

[31] Peter van der Linden. *Expert C Programming Deep C Secrets*. SunSoft Press, Prentice Hall, 1994.

[32] vbns,. http://www.vbns.net.

[33] Wietse Venema. Tcp wrappers. ftp://ftp.win.tue.nl/pub/security/tcp_wrapper.ps.Z.

[34] Tatu Ylonen. Secure shell. http://www.cs.hut.fi/ssh.