Design of the new and improved NetSpec controller

by

Radhakrishnan R. Mukkai

B.E. (Computer Science and Engineering)

College of Engineering, Osmania University, Hyderabad, India

April 2001

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science

Dedicated to my parents

Ramakrishnan and Devika for their infinite love

Acknowledgements

I would like to thank Dr. Jerry James and Dr. Douglas Niehaus for guiding me throughout the course of my thesis. My association with Dr. James and Dr. Niehaus has helped me become a better Software Engineer. I have thoroughly enjoyed working with them. I would like to thank Dr. David Andrews for serving as a member of my thesis committee.

I would like to thank Leon Searl for helping me during the course of my thesis. I have learned a great deal about programming and project management from Leon.

I am forever indebted to my parents, Ramakrishnan and Devika, and my sister, Ammu for their infinite encouragement, support and love. I owe all my success to them.

I am grateful to Jennifer Holvoet, Dr. Prasad Gogineni and Dr. David Braaten who gave me an opportunity to work as a Graduate Research Assistant and funded me during the course of my thesis.

I would like to mention my roommates Pavan, Visu and Rajesh who have made my stay in Lawrence memorable. I would also like to thank all my friends who have directly or indirectly helped me with my thesis work.

Abstract

NetSpec is a tool designed for network experimentation and testing. NetSpec provides a framework which enables a user to centrally control daemons running on other machines. The power of NetSpec comes from the centralized controller.

However, the design of the older NetSpec controller was very complex and not modular. One of the major problems with the older design was that both parsing of the NetSpec user script and control operations (like connecting to the appropriate machines and invoking the daemons) were performed simultaneously. There were no data structures present in the controller to store the parsed values. The input to the NetSpec controller was present in two separate files - one file, called the schedule file, had information required to schedule the daemons participating in the experiment and the second file, called the NetSpec script file, had information about the daemons which the user planned to invoke.

The NetSpec controller was hence redesigned with the goal of making it simple, elegant and modular. In the present NetSpec controller, parsing and control functionalities have been separated. First, we parse the NetSpec script storing the parsed values in well-defined data structures. These data structures are then used to perform the necessary control operations. The schdedule file and the NetSpec script file were combined to ensure that information about scheduling and daemons are present in one place.

Other major changes include having many error handling routines and increased modularity of the NetSpec code. The major functions performed by the controller have been delineated by having them in separate source files. These major functions have been further modularized, so that each function performs a single task.

Contents

List of Tables

List of Figures

List of Programs

List of Scripts

Chapter 1

Introduction

1.1 What is NetSpec?

NetSpec is a software tool developed by researchers at the University of Kansas for the ACTS ATM Internetwork (AAI) project. NetSpec was originally intended to be a traffic generation tool for large-scale data communication network tests with a variety of traffic source types and modes. NetSpec provides a simple block structured language for specifying experimental parameters and support for controlling experiments containing an arbitrary number of connections across a LAN or WAN. However, NetSpec can be used for purposes other than traffic generation and this will become clear when we explain the workings of NetSpec in more detail.

1.2 Why NetSpec?

"Quality assurance is an important phase in the networking software development cycle. To date this phase is marked by the presence of various non-interacting components, each designed for certain networking functionality. Some of them are described below:

- Components involved in setting up an entire test network, e.g., *ping* and *traceroute*.
- Utilities that measure network performance at the application or protocol level

and therefore are helpful in both troubleshooting and tuning.

- Utilities that measure each network element's performance in terms of percentage CPU utilization and memory utilization, e.g., KU's *Data Stream Kernel Interface* (DSKI).
- Network monitoring components such as *tcpdump* which play a passive role in helping the security and integrity of the developed network.

Absence of a network performance tool that can integrate the above-mentioned components severely limits the two most important characteristics associated with testing: scalability and reproducibility. NetSpec is designed to overcome these limitations by integrating these components and controlling them via a centralized command and control system." [6]

1.3 Features of NetSpec

- **Scalability**

  The framework that is provided is scalable to carry out multiple tests. By multiple tests, we mean the ability to invoke various daemons from a single script file. This is very desirable since networks generally carry hundreds of flows simultaneously.

- **Flexibility**

  The framework is flexible, incorporating both passive (probes, measurements) and active nodes (traffic generators).

- **Reproducibility**

  One of the benefits of NetSpec is that it automates the experiment, which emphasizes reproducibility. NetSpec provides the ability to run an experiment many times and each time we are assured that the experiment has run the same way.

- **Integration**

  Measurements and tests are integrated in a seamless manner.

- **Extensibility**

  The design has been done with a provision to add new components. New daemons can be incorporated in the existing framework with ease.

1.4 Why the New and Improved NetSpec Controller?

Before proceeding to discuss the reasons for redesigning the NetSpec controller, let us take a look at the older NetSpec architecture (henceforth, referred to as NetSpec version 5.0). By examining this architecture, one can get a clear picture of what NetSpec is. We will then discuss the limitations this architecture posed and how these limitations are overcome with the newer architecture (henceforth, referred to as NetSpec version 6.0). Figure 1.1 [6] shows the NetSpec version 5.0 architecture. An entire experiment is controlled by a centralized controller. The controller is invoked by the NetSpec user program. The NetSpec user program accepts a script file from the user and passes it to the controller (also referred to as the control daemon). The script contains the name of the daemons that the controller needs to invoke in specified hosts and contains user data in the form of *parameter = value* pairs to be passed to the daemons. Let us take a look at a portion of a NetSpec script shown in Script 1.1

Script 1.1 Sample NetSpec Script

```
netTraffic testbed61 {
  type = full (blocksize=32768, duration=10);
  protocol = tcp;
  own = testbed61:8002;
  peer = waldorf:8002;
}
```

In the above script, *netTraffic* refers to the NetSpec traffic daemon and we would like to invoke this daemon on host *testbed61*. The controller looks at the user data as just parameter and value pairs. This data make sense to the daemons to which the data is passed and these daemons parse this data and accomplish the task for which they were designed. This means that any NetSpec user could design a daemon, which performs a specific task, and then use the NetSpec script and the controller to invoke these daemons on the specified hosts. An experiment might involve participation of different daemons, each designed for unique network functionality.

The *netspecd* daemon in Figure 1.1 acts as a service multiplexer by ensuring that several daemons of the same or different type can be invoked on a particular host. This daemon is basically a server and the control daemon usually contacts the *netspecd* daemon when it needs to invoke a daemon on a particular host. The controller and the *netspecd* daemon exchange messages, based on which the *netspecd* daemon spawns the required daemon and then waits for requests from the controller. The (connection-oriented, TCP) socket thus set up is then used by the spawned daemon and the controller to communicate.

Some of the limitations in this design are:

- The controller performs the parsing and control operations simultaneously. This causes the controller to become unnecessarily complex and non-modular. In the new design, the parsing and control operations are performed separately and data structures act as the glue that ties them together. The controller first parses the NetSpec script and fills the appropriate data structures. These data structures are subsequently used by the controller to perform the appropriate control logic (like connecting to the appropriate host, invoking the daemons, passing the parameter-value pairs, etc.). By separating these two operations, we have made it more modular and simple.

- The NetSpec user script is given as input to the NetSpec user interface which then passes it to the controller for parsing and performing the various control operations. The user interface was responsible for initiating the various phases in which the daemon execution takes place (explained in detail in later Chapters). The controller was designed more as a daemon, but is more powerful as it could invoke daemons specified in the script. It also had the ability to invoke itself, just like other daemons. This results in a multi-level control hierarchy. However, a vast majority of the experiments which were run using NetSpec utilized the single-level control hierarchy. Hence, we decided to continue with the single-level control hierarchy only due to its simplicity and widespread use. This also ensures that the entire control functionality is present in the controller and not distributed to the NetSpec user interface as it was in NetSpec version 5.0.

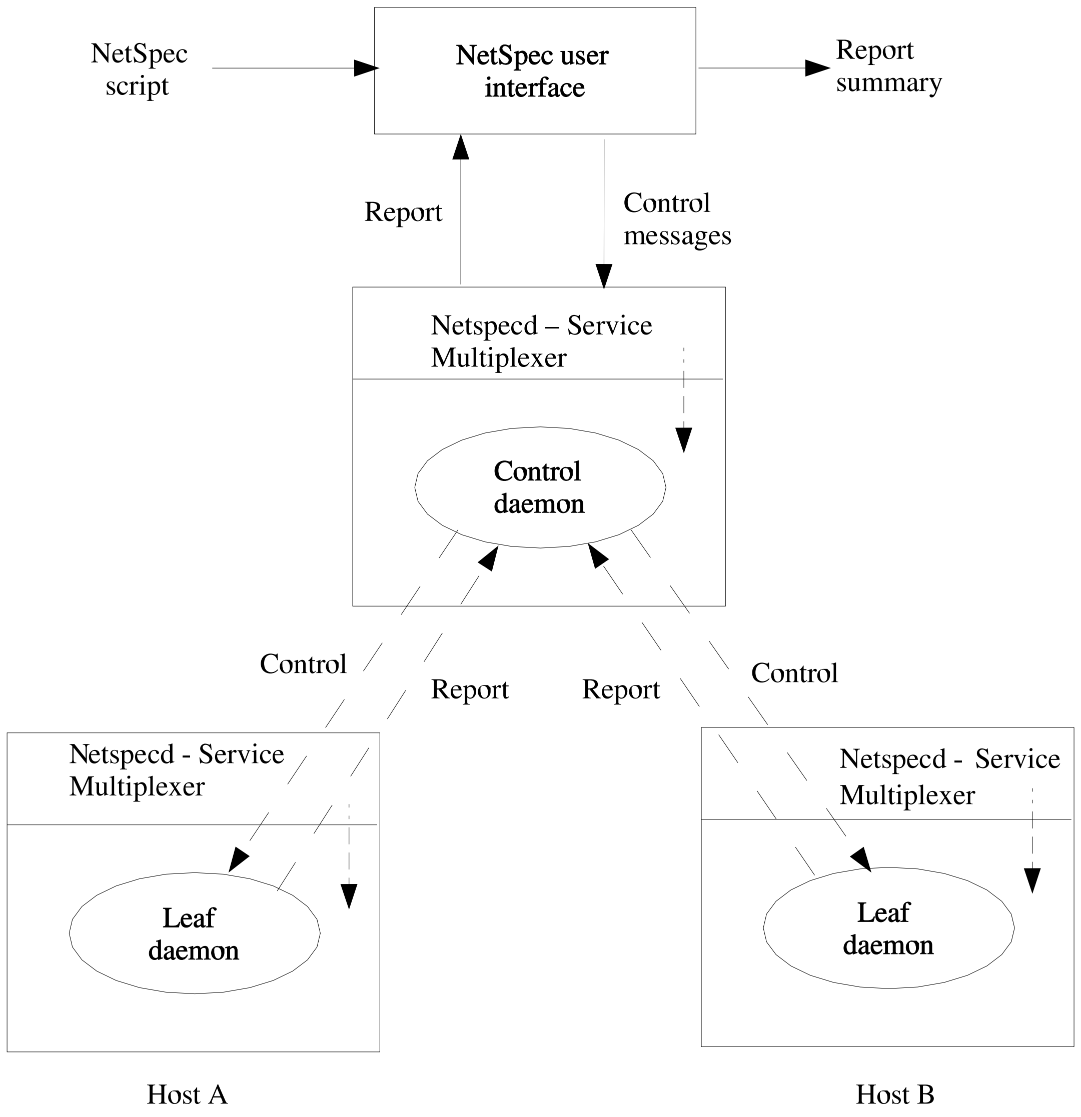


Figure 1.1: NetSpec 5.0 Architecture

Chapter 2

Related Work

2.1 Network Testing Tools

There are various testing tools which are freely available. Some of the notable tools are described below:

- **TTCP** [7]: Test TCP (TTCP) is a command-line sockets-based benchmarking tool for measuring TCP and UDP performance between two systems. It can also be used as a network pipe to transfer data between two systems. It was originally developed for the BSD [1] operating system in 1984. The original TTCP sources are in the public domain, and copies are available from many anonymous FTP sites.

- **Iperf** [9]: Iperf is a tool to measure maximum TCP bandwidth, allowing the tuning of various parameters and UDP characteristics. Iperf reports bandwidth, delay jitter and datagram loss. Though there are many network tools that measure network performance, such as ttcp, most are old and have confusing options. Iperf can be considered to be a modern alternative for measuring TCP and UDP bandwidth performance.

- **Netperf** [2]: Netperf is a benchmark that can be used to measure the performance of many different types of networking. It provides tests for both unidirectional throughput and end-to-end latency.

"TTCP provides a means to measure TCP throughput through an IP path. Testing involves starting the receiver on one end and the transmitter on the other end. The transmitter sends a user-specified number of TCP packets to the receiving side. At the end of the test, the two sides display the number of bytes transmitted and the time elasped for the packets to pass from one end to the other." [8]

NetSpec provides a scripting language to specify daemons and daemon data. It is more flexible than TTCP in that we could run more complex experiments by having scripts as input rather than command-line options. If we wanted to measure the throughput between 4 pairs of hosts using TTCP, it would involve opening 8 xterms and starting TTCP on each of those hosts by specifying the appropriate command line options.

The NetSpec centralized controller ensures that we do not have to open xterms on various hosts if we need to conduct an experiment involving daemons on those hosts. NetSpec helps the user set up a network from a central location using the information provided in the script and run the experiment as specified in the script. TTCP is designed specifically for measuring TCP/UDP throughput, determining the actual bit rate of a particular WAN or modem connection and also testing the connection speed between any two devices with IP connectivity between them. NetSpec is not limited to any specific testing methodology. It need not be used for network testing and experimentation alone. By designing daemons, we can use the NetSpec centralized controller to spawn daemons across various machines. Thus, NetSpec provides capabilities to setup a network and perform computations which require setting up processes across various hosts.

Chapter 3

Background Information

In this section, we explain NetSpec phases and how the script writer can schedule the various phases of daemons specified in the NetSpec script and invoked using the controller.

There are two mechanisms currently provided by NetSpec to schedule the phases of daemons participating in an experiment.

- NetSpec Classic eight phase
- NetSpec Variable phase

We discuss NetSpec Classic 8 phase mechanism in this section. NetSpec Variable phase mechanism is described in Chapter 4 (NetSpec 6.0 Architecture).

3.1 NetSpec Classic 8 Phase Behavior

"Daemons accomplish their tasks in phases. Their execution is controlled by the control daemon using a command-control interface. The daemons and the controller use a text-based protocol, called *Remote Control and Information Protocol (RCIP)*, to communicate.

All the daemons participating in the experiment are executed in eight phases, irrespective of the daemon types. The controller passes the commands shown in Table 3.1 to the slave daemons starting with the *setup* command. A phase's successful execution

Command	Action
Setup	Allocate Resources
Open	Establish necessary socket sonnections
Run	Execute the desired functions
Finish	Finish the execution
Close	Close all the socket connections
TearDown	Release all the acquired resources
Report	Prepare and send the report
Kill	Terminate the execution

Table 3.1: The 8 Phase Classic Execution Model

is notified through an acknowledgement process, called *rcipacknowledge*, to the control daemon. After receiving a successful acknowledgement for a phase from all participating daemons, the control daemon moves to the next command in Table 3.1 [6]. An experiment can be marked by the presence of many slave daemons. In an execution phase, it may be desirable to accomplish all participating daemon functions either in serial or in parallel. This can be achieved by specifying proper execution constructs in the user scripts." [6]

Execution Constructs

The control daemon accepts the following three types of execution constructs:

1. Serial

2. Parallel

3. Cluster

These execution constructs apply only to the *open*, *run* and *close* phases of the daemons participating in the experiment. All the other phases, like *setup* and *kill*, take place in serial. For two daemons A and B, the *setup* phase for daemon A (or B) is completed before the *setup* phase for daemon B (or A) is started.

When two daemons, Daemon A and Daemon B, are inside a serial, parallel, or cluster construct the following behavior is observed:

- **Serial**: The *open*, *run* and *close* phases with respect to one daemon (say daemon A) are finished before daemon B is executed. So, daemons A and B are executed serially.

- **Parallel**: The run phases of both daemons are executed in parallel. A phase represents sending the command (in this case the *run* command) to the daemon and waiting for an acknowledgement from the daemon. In this construct, the *run* command is sent to all the daemons and this initiates the *run* phase in all the daemons simultaneously.

- **Cluster**: In this construct, the *open*, *run* and *close* phases of both the daemons are done in parallel. The *open* phase is first executed in parallel by sending the *open* phase command to all the daemons. After receiving the acknowledgements from the respective daemons, the *run* and *close* phases are executed in a similar manner.

Figure 3.1 [6] shows the various messages passed between the controller and daemons for each of the three execution constructs and illustrates the working of each of these constructs. The labeled arrows show the messages which are passed between the controller and the daemons. Consider the serial construct in figure 3.1, the controller passes the message *OPEN* to daemon A causing it to initiate its *OPEN* phase. After an acknowledgement for the phase, indicated by *ACK*, is obtained from the daemon we initiate the *RUN* phase for daemon A by sending the *RUN* command to the daemon. After the *OPEN*, *RUN* and *CLOSE* phases for daemon A are completed, the controller initiates the similar phases for daemon B. Thus, daemons A and B are executed *serially* or one after the other.

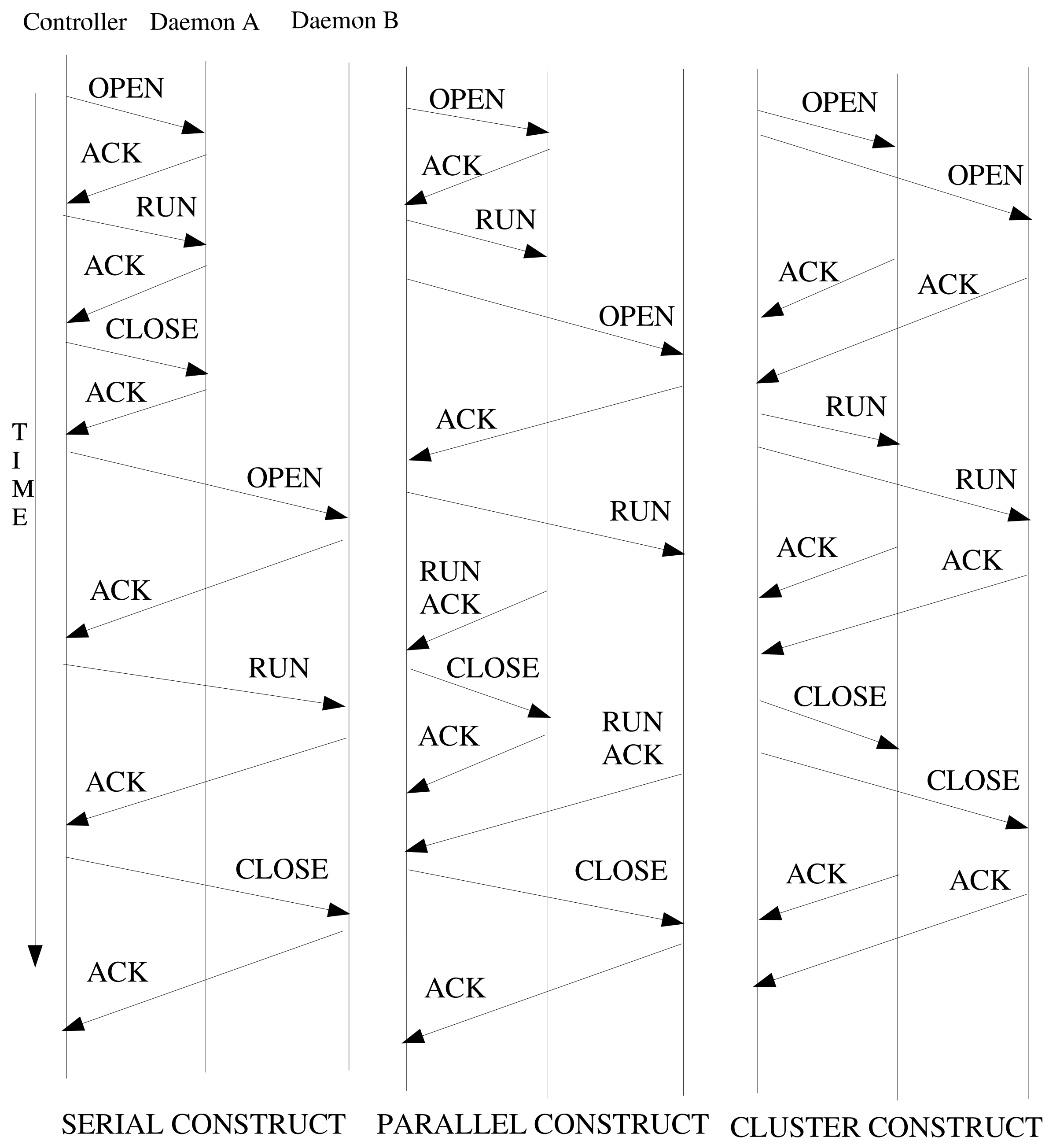


Figure 3.1: Execution Construct

Chapter 4

NetSpec 6.0

In this chapter, we discuss the NetSpec 6.0 architecture. We also describe Variable Phase NetSpec, which is another mechanism (along with the Classic 8-phase mechanism explained in the Background section) provided by NetSpec to schedule the phases of daemons specified in the NetSpec script. Newer features incorporated into the NetSpec script are also mentioned.

4.1 NetSpec 6.0 Architecture

Figure 4.1 shows the new NetSpec architecture. The user interface is no longer present in the new architecture. The user script is now given directly to the controller, which parses the script and fills data structures, and is also responsible for showing the results of the experiment to the user. This functionality was provided by the user interface in NetSpec 5.0.

The input to the NetSpec 6.0 controller is the script file. The script file is parsed and appropriate data structures are then filled. These data structures are used by the controller to connect to the daemons specified in the script, pass the parameter-value pairs to them, and display the results of the experiment to the user.

Features of the NetSpec 6.0 Architecture

- The user interface portion in NetSpec 5.0 has been removed. This has simplified

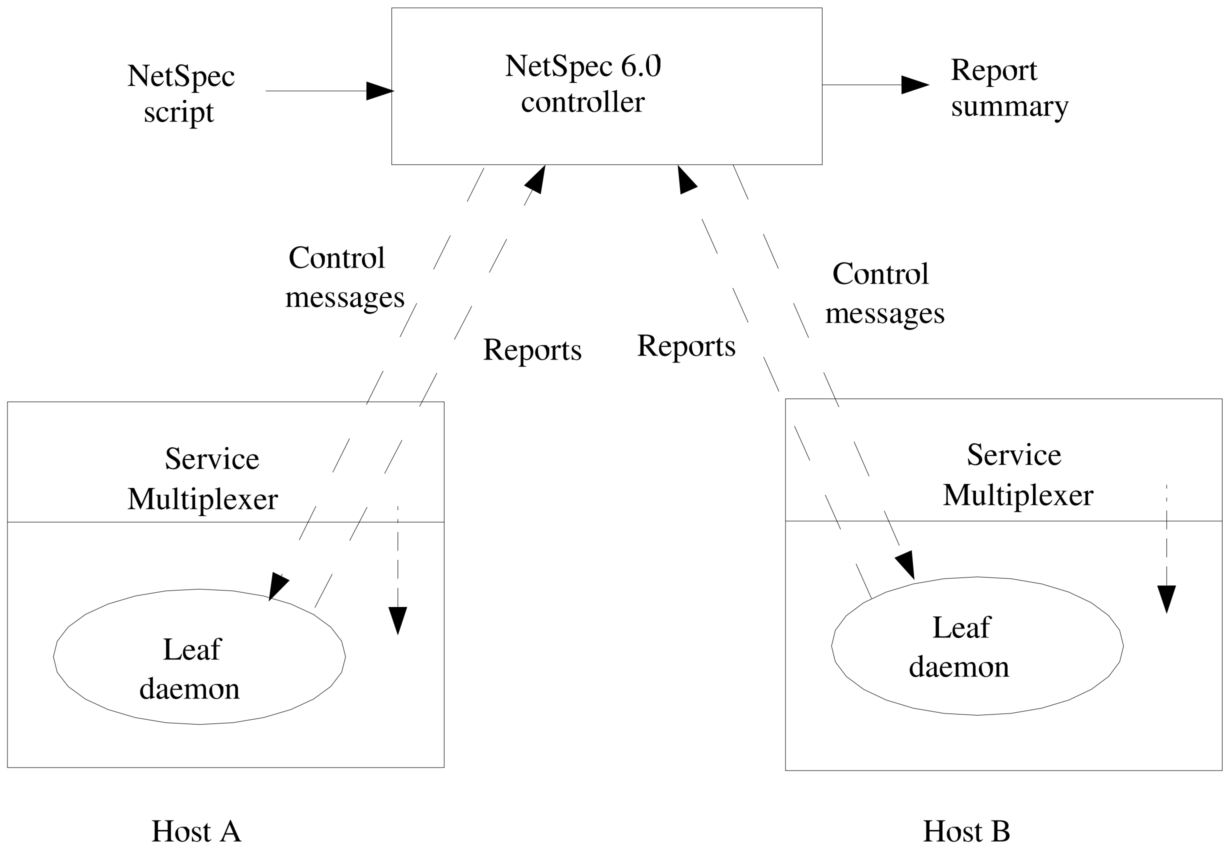


Figure 4.1: NetSpec 6.0 Architecture

the NetSpec architecture.

- Separating the parsing and control functionalities have made the control design simple and modular.

- In NetSpec 5.0, we could have a multi-level control hierarchy. The NetSpec controller was designed more as a daemon than a controller, and it had the ability to spawn itself. It was found that majority of the experiments run using NetSpec used the single-level control hierarchy. Hence, the multi-level control hierarchy is not supported in the present controller. A multi-level control hierarchy would have been a nice feature but, as we could not find suitable scenarios where it could be utilized, it was felt that it was not necessary to introduce the feature and make the architecture more complex.

4.2 NetSpec Variable Phase

The variable phase feature was introduced in version 5.0 of NetSpec and it addressed the following shortcomings present in version 4.0

- Each daemon participating in an experiment was limited to eight phases.
- Only the *open*, *run* and *close* phases of a daemon could be run in *serial*, *parallel* or *cluster* mode as discussed in Chapter 3.
- All the daemons had to go through their execution in eight phases. This resulted in daemons doing nothing in certain phases, e.g., the NetSpec *system command* daemon. The *system command* daemon provides the ability to construct a desired command line on the target system and then invoke it using the *system* system call, thus resulting in the command to be executed on the target system. The *open* and the *close* phases do not have any meaning for the *system* daemon. Only the *run* phase is important, where the *system* system call is invoked. Thus, the *open* phase was implemented for the *system* daemon with it doing nothing in that phase except for sending an acknowledgement to the controller indicating that the phase was over.

These limitations have been removed with the introduction of the NetSpec variable phase feature. This feature allows a NetSpec user to choose the number of execution phases for each daemon. It also provides a mechanism to specify how phases of various daemons has to be scheduled. Using schedule information provided in the NetSpec script, a NetSpec experiment involving multiple daemons can be divided into different slots. Each slot contains phases of different daemons which will be executed concurrently. The slots themselves are executed in serial. Hence, phases of daemons which should be executed *serially* will be placed in different slots. The slot-based scheduling mechanism is explained in further sections.

Several changes have been made to the Variable phase feature in NetSpec 6.0. We will discuss features of the Variable phase mechanism as implemented in NetSpec 5.0, followed by the modifications made in NetSpec 6.0.

NetSpec 5.0 Variable phase features

- Execution constructs passed from the command line were used to tell the controller how to schedule phases of daemons in a particular slot. For example, *serial* would mean that the phases of daemons in a particular slot have to be executed one after the other and *parallel* would mean that the phases in a slot should be executed concurrently.
- Schedule information was passed to the controller through a separate file, apart from the NetSpec input script file.
- Two different parsers were used to parse the schedule information (in the schedule file) and daemon information (present in the input script file). The schedule parser was invoked using the *system* system call.

Improvements to Variable phase features in NetSpec 6.0

- In NetSpec 6.0, two-levels of schedule information was provided. In the first level, we have the execution constructs *serial* and *parallel*. These tell us how the daemons need to be scheduled. Consider a NetSpec script specifying 4 daemons. The *serial* construct would mean that the 4 daemons have to be executed one after the other. The *parallel* construct would mean that the 4 daemons have to be executed concurrently. The constructs *serial* and *parallel* do not tell us how the phases of the daemons will be scheduled. On the second level, we have the slot-based scheduling mechanism, which provides us with such information. The two-level approach to expressing schedule information has greatly increased the simplicity and power of NetSpec. Information for scheduling individual daemons and the phases of various daemons that were tied together previously have been separated.

- The schedule file and the script file have been combined to form a single input script file. The scheduling details and the daemon details can now be found from a single source.

- A single parser is now used to parse the schedule and the daemon information.

Advantages of Variable phase feature

- Daemons are no longer restricted to execute in eight phases. Daemon designers can now design daemons, that execute in one or more phases, as deemed fit by the designer.

- NetSpec users can now schedule the phases of daemons in any order as specified in the schedule information. Using the classic 8-phase model, we could not schedule the *run* phase of a daemon before the *open* phase of another. Only after the *open* phase of all daemons is complete, we move onto the *run* phase. The variable phase feature would allow us to do such scheduling. Thi concept is explained further in the following sections.

- Only the *open*, *run* and *close* phases are subject to the various execution constructs. When you used the *parallel* execution construct, only the *open*, *run* and *close* phases were executed in parallel, whereas the remaining five phases were executed in *serial*.

A sample NetSpec script with the slot-phase association (schedule information) and the daemon information is shown in Script 4.1.

The first block in Script 4.1 represents the schedule information. We now describe, the meaning of the schedule portion of the script. The *daemon_name = dummy* construct tells us that the scheduling information of the *dummy* daemon is being specified. Note that the scheduling information is for a *daemon type* rather than for a daemon on a specific host. In the above script, we have provided schedule information for the

Script 4.1 NetSpec Script Showing Slot-phase Association

```
map defmap {
  daemon = new_daemon (daemon_name = dummy;
           setupCommand = 1;
           runCommand = 2;
           finishCommand = 3;);
}

serial {
   dummy testbed13 {
     varA = 3;
     varB = 4;
     varC = 5;
     varD = 6;
   }

  dummy testbed14 {
     varA = 1;
     varB = 2;
     varC = 3;
     varD = 4;
  }
}
```

dummy daemon and we have specified two *dummy* daemons in the daemon portion of the script. The scheduling information applies to both these daemons. Also, look at how the *serial* execution construct block encompasses the specification of the two *dummy* daemons. While the schedule information provided in the first block would tell us how we would schedule the phases of various daemons, the *serail* construct tells us that the *dummy* daemon invoked on testbed13 will be scheduled first followed by the *dummy* daemon on testbed14.

The *setupCommand = 1* line tells us that the *setup* command/phase for the *dummy* daemon should be executed in slot #1. The NetSpec controller parses the schedule information and constructs commands which are then passed to the daemons. When the controller wants to initiate the *setup* phase for the daemon, it passes the command string constructed to the daemon. The *rcips* parser on the daemon side is responsible for handling the commands from the controller and initiates the appropriate phase desired by the controller. The command string passed to the daemon is of the following format:

daemon type:Command/phase which the daemon needs to execute

Example: *dummy:setupCommand*

- *dummy*: Refers to the daemon name
- *setupCommand*: Indicates the *setup* phase for the dummy daemon.

The schedule portion of the script in NetSpec 5.0 is shown below:

```
map defmap {
  daemon = new_daemon (daemon_name = dummy;
           phaseACommand = 1;
           phaseBCommand = 2;
           phaseCCommand = 3;);
}
```

Hence, in NetSpec 5.0 the command string passed to the daemon had the following format:
daemon type:phase (phase number in alphabet)Command

Example: *dummy:phaseACommand*

- *dummy*: Refers to the daemon name
- *phaseACommand*: phase# in alphabet = A. Indicates the command corresponding to phaseA. On the daemon side, the command corresponding to the alphabet parsed ("A" in this case) is invoked. It could be the *setup*, *open* or any other command as decided by the daemon designer.

We felt that the string *phaseACommand* does not give a clear idea about the exact phase in which we expect the daemon to execute. Such a term is extremely ambiguous and we decided to mention the exact phase/command that we plan to execute in a particular slot. This makes it easier to understand the schedule information. However, future daemon designers can use the following format for the command string:

daemon type:string representing the phase which needs to be executed

The NetSpec controller calls the *phaseControlCommand ()* function of the daemon which takes the command string defined above as an argument. The *phaseControlCommand ()* for each daemon decides whether the command is intended for that particular daemon or not. Based on this decision it either ignores the command by simply acknowledging the command, or else it performs the function corresponding to that command.

A typical implementation (in pseudocode) of the *phaseControlCommand ()* is shown in Program 4.1. The variable *phaseStr* in the example refers to the command string of the format: *dummy:setupCommand*.

4.2.1 Scheduling Daemons

In this section, we will explain in detail the slot-based execution construct provided by NetSpec Variable phase feature to schedule daemons participating in an experiment.

There are two execution constructs supported by NetSpec Variable phase feature.

- **Serial**: All daemons specified in the daemon portion of the script will be executed one after the other.
- **Parallel**: All daemons specified in the script will be executed concurrently.

In both the cases, scheduling information provided in the script would tell us how the phases of different daemons will be executed.

Let us take a sample NetSpec variable phase script involving three daemons (daemonA, daemonB and daemonC for simplicity's sake). Each daemon is run on 2 machines. Using the following scenario, we explain how scheduling is performed. The schedule portion of the script is shown in Script 4.2.

Program 4.1 Pseudocode for the phaseControlCommand () function

```
void phaseControlCommand (char *phaseStr) {
                ....
                ....

    check to see if the command string is meant for this
    particular daemon type

    extract the command to be executed

    if phaseString is ‘‘setupCommand’’ then
      setupCommand ();
    else if phaseString is ‘‘openCommand’’ then
      openCommand ();
    else if phaseString is ‘‘runCommand’’ then
      runCommand ();
    else if phaseString is ‘‘closeCommand’’ then
      closeCommand ();
    else if phaseString is ‘‘finishCommand’’ then
      finishCommand ();
    else
      send an ACK to the controller;
  } /* end of phaseControlCommand function */
}
```

Script 4.2 NetSpec Script Schedule Information

```
map defmap {
  daemon = new_daemon (daemon_name = daemonA;
             setupCommand=1;
             runCommand=3;
             finishCommand=4;);
}

map defmap {
  daemon = new_daemon (daemon_name = daemonB;
             setupCommand=1;
             runCommand=3;
             finishCommand=4;);
}

map defmap {
  daemon = new_daemon (daemon_name = daemonC;
             setupCommand=2;
             openCommand=3;
             runCommand=4;
             finishCommand=6;);
}
```

Each block in Script 4.2 represents scheduling information for the respective daemon type mentioned in the *daemon_name=daemon* portion in the block.

From the schedule portion of the script, it is evident that we would like to schedule the *setup* phase of daemonA in Slot 1, the *run* phase in Slot 3 and the *finish* phase in Slot 4. For daemonB, we would like to schedule the *setup* phase in Slot 1, the *run* phase in Slot 3 and the *finish* phase in Slot 4.

Consolidating the information obtained from the script would give us the following phase table or schedule:

SLOT 1: daemonA:setupCommand and daemonB:setupCommand

SLOT 2: daemonC:setupCommand

SLOT 3: daemonA:runCommand, daemonB:runCommand and daemonC:openCommand

SLOT 4: daemonA:finishCommand and daemonC:runCommand

SLOT 5: daemonB:finishCommand

SLOT 6: daemonC:finishCommand

This clearly shows the phases of daemons that execute in a particular slot. In Slot 1, the *setup* phases of daemonA and daemonB are executed concurrently. In Slot 2, only the *setup* phase of daemonC are executed. Thus, the scheduling information tells us that we would like the *setup* phases of daemonA and daemonB to happen in parallel, whereas the *setup* phase of daemonC would happen in the next slot and hence will be executed after the *setup* phases of daemonA and daemonB are completed. In a particular slot, phases of different daemons are executed concurrently. Individual slots are executed one after the other. Only after all the phases delegated to Slot 1 are complete, we will move to Slot 2 and so on.

Let us now look at the daemon portion of the NetSpec script shown in Script 4.3

Script 4.3 NetSpec Script Daemon Information

```
serial {   # can be either serial or parallel
  daemonA testbed1 {
    parameter = value;  # specify parameter value pairs
                        # associated with daemonA
  }
  daemonA testbed2 {
    parameter = value;  # specify parameter value pairs
                        # associated with daemonA
  }
  daemonB testbed3 {
    parameter = value;  # specify parameter value pairs
                        # associated with daemonB
  }
  daemonB testbed4 {
    parameter = value;  # specify parameter value pairs
                        # associated with daemonB
  }
  daemonC testbed5 {
    parameter = value;  # specify parameter value pairs
                        # associated with daemonC
  }
  daemonC testbed6 {
    parameter = value;  # specify parameter value pairs
                        # associated with daemonC
  }
}
```

4.2.1.1 Serial Execution Construct

The execution behavior that results from using the *serial* construct is clearly illustrated in Table 4.1. It is evident from Table 4.1 that daemons are executed one after the other using the *serial* construct. Daemon phase execution follows the slot-based schedule information provided in the script. The table only shows the execution of the first 4 daemons.

daemonA on testbed1	daemonA on testbed2	daemonB on testbed3	daemonB on testbed4	daemonC on testbed5	daemonC on testbed6
setup					
run					
finish					
	setup				
	run				
	finish				
		setup			
		run			
		finish			
			setup		
			run		
			finish		

Table 4.1: Variable Phase: Serial Construct

4.2.1.2 Parallel Execution Construct

The execution behavior that results from using the *parallel* construct, is clearly illustrated in Table 4.2. It is evident from Table 4.2 that phases of all daemons are executed in parallel in a given slot. It should be remembered that all slots execute in serial.

The columns in a particular row represent execution that would take place in parallel and individual rows are executed serially. Each element in the table represents the phase that will be executed on the daemon-machine pair mentioned in the corresponding column. An empty cell means that no phase is being executed on the daemon-machine pair.

daemonA on testbed1	daemonA on testbed2	daemonB on testbed3	daemonB on testbed4	daemonC on testbed5	daemonC on testbed6
setup	setup	setup	setup	setup	setup
finish	finish			run	run
		finish	finish		
				finish	finish

Table 4.2: Variable Phase: Parallel Construct

4.3 New NetSpec Script Language Features

Besides changes to the NetSpec controller design, several changes were made to the NetSpec scripting language. However, we ensured that any changes to the scripting language do not cause any backward compatibility problems with older NetSpec scripts. These features are enumerated below:

- **Schedule Information**

  Daemons invoked by the controller execute in phases (explained in detail in the Background section). NetSpec provides a means by which we can schedule the various phases of daemons participating in an experiment using the slot-based execution construct. In NetSpec version 5.0 this scheduling information is present in a separate file and was passed to the NetSpec user interface through a command line argument. This information was then parsed by the user interface and then appropriate commands were passed by the interface to the controller daemon to initiate the various phases as described in the schedule file.

  In NetSpec 6.0, the schedule information can now be specified along with the script file having the daemon information (which specifies the peers, daemons, and parameter-value pairs). By tying these two pieces of information together, we have now made the NetSpec script the one and the only source for all information regarding the schedule and daemons. If the script writer does not specify the schedule information, the daemons are scheduled in eight phases (also known as Classic NetSpec Behavior and explained in detail in the Background section). It also makes sense to specify both the schedule and daemon information in a single file, as they are interdependent. The schedule information parser was also integrated with the NetSpec control parser, so, a single parser parses the entire script.

- **Transferring Files to/from Daemons**

  In certain cases, we wanted to download one or more files specified in the script file to the daemon. The daemon uses the files as input for its processing tasks. Capabilities for getting files back from the daemon were also needed. This capability was necessary, as the daemon could send back the results of its processing through a file to the controller. A daemon options section has now been added where we can specify the file we want to send to or receive from the daemon along with the peer name in which we want to have the daemon spawned. This feature increases the power and functionality that NetSpec provides.

  In the daemon definition of the script after the address/name but before the daemon's specific parameter values section {}, there is an optional *general daemon parameterValues* section delimited by *()*. General daemon parameterValues are parsed and processed by the *rcipsparse* parser immediately before the daemon specific parameterValues are parsed and processed by the daemon's own parser. As the name suggests, the *rcipsparse* parser is responsible for handling control requests that originate from the controller. The controller communicates with the individual daemons using the *RCIP* text protocol and this parser is responsible for parsing the text messages and initiates the appropriate phases. This parser is shared by all the daemons and each daemon has its own parser, which it uses to parse daemon specific parameter value pairs.

  The daemon specific code never sees the general daemon parameter values.

  Currently the valid parameterValue of the general daemon parameter values is the *to_file* and the *from_file* parameter. The value of the *to_file* parameter is two filenames in strings separated by a *colon(:)*. The first file name is the name of a file on the controller host. The second file name where the file should be put on the remote host. The file is transferred from the controller host to the daemon host as part of the daemon initialization. The *from_file* also follows a similar format, except that the file name before the colon represents a file on the daemon/remote host and the one after the colon represents where the file should be put on the controller host. The file is transferred from the daemon host to the controller host after the controller is done executing the phases of daemons as specified in the schedule portion of the script.

The *to_file* and the *from_file* parameter value can appear any number of times and hence we can send/receive many files to/from the daemon as shown in Script 4.4

Script 4.4 Script Options for Transferring Files to/from Daemons

```
parallel {
    nssyscmd NodeB (to_file="echo.txt":"/tmp/echo.txt",
            from_file="test.txt":"/tmp/scriptTest.txt") {
           cmd="rm -f /tmp/return.txt";
           cmd="cat /proc/stat > /tmp/return.txt";
           cmd="cat /tmp/echo.txt >> /tmp/return.txt";
           cmd="sleep 10";
           cmd="cat /proc/stat >> /tmp/return.txt";
           cmd="rm -f /tmp/echo.txt";
           filename="/tmp/return.txt";
    }
}
```

In Script 4.4 the *echo.txt* file in the current directory (the current directory when *netspec* was run) is transfered to the filename */tmp/echo.txt* on the daemon host during daemon initialization. This happens before any of the *cmd* or *filename* daemon specific parameterValues are processed by the daemon.

- **Daemon and Host Identifier**
  Provision for uniquely identifying a daemon/host pair has been provided. A script may specify a number of daemons of the same type (ex. NetSpec traffic daemons) running on the same machine. However, the controller does not have any means to distinguish between the different daemons. Therefore, an optional daemon identifier can now be provided which enables the controller to make such distinctions.

Script 4.5 Script Options for Identifying Daemon/Host Pair

```
netTraffic testbed61 daemon_instance1 {
   ...
}

netTraffic testbed61 daemon_instance2 {
   ...
}
```

- **Comments**

  Comments can now be added to NetSpec scripts. Comments start with # and extend to the end of the line.

4.4 Miscellaneous Features

While redesigning the NetSpec controller, several error-handling and debugging features were added to the source code. Some such features have been enumerated below:

- Line and column number information are printed when the controller encounters errors while parsing the NetSpec script. This leads the user to the location where the error was detected and results in faster error resolution. The parser has been redesigned so that it parses the values in a bottom-up manner. This means that values which are parsed (representing the tokens or the leaves) are passed up successively across various levels and then combined in data structures until the entire script is parsed.
- The source code has been made modular with the addition of functions which perform a single/unique task.

Chapter 5

Implementation

Figure 5.1 shows the design of the NetSpec controller. We will use this figure to describe the implementation details of the various components of the controller.

As can be seen from the NetSpec controller block in Figure 5.1, the three major components are — (a) the parser, which is responsible for parsing both the schedule and daemon information present in the script, (b) the data structures which act as a glue between the parser and the control logic. Parsed values are stored in data structures which are subsequently used by the control logic and (c) the control logic, which is responsible for using the data structures to contact the appropriate peers, invoke the necessary daemons and pass the parameter-value pairs to it. The daemons and the controller use a text-based protocol called *Remote Control and Information Protocol (RCIP)* to communicate among them. The controller connects to the daemons using TCP-based sockets. The report generated by the daemons is then passed over to the controller, which displays it to the user running the experiment.

The *main ()* function of the NetSpec controller has been provided below to illustrate the various components involved.

Note that the main () function has not been reproduced exactly. The comments in the original program are lengthy. Only the essence of the function has been shown.

We will now discuss the working of the parser (along with the data structures which the parser fills up) and the control logic.

Program 5.1 NetSpec controller main() function

```
int main (int argc, char *argv[])
{
  /* Process the command line arguments */
  processCommandLineArguments (argc, argv);

  /* Parse the script file */
  daemonParserparse ();

  /* Set up OS signals */
  setupOSSignals ();

  /* Print input NetSpec script. Only used for debugging purpose. */
  /* Control logic will not be executed if option is set*/
  if (prettyPrintingOption == TRUE) {
...
      prettyPrinterSchedule ();   /* Print schedule portion of script */
      prettyPrintingDaemons ();  /* Print daemon portion of script */
  }
  else {
    /* Invoke NetSpec controller */
    netspecCentralizedController ();
  }
  return 0;
}
```

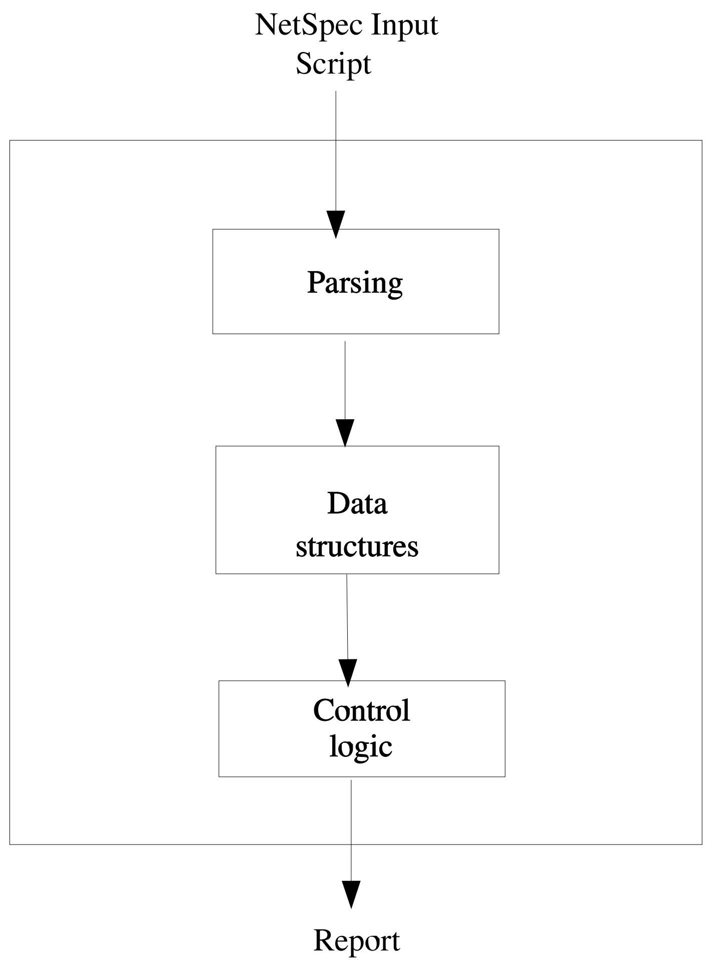


Figure 5.1: NetSpec 6.0 Controller Design

5.1 Parser

The control parser has been designed using standardized tools present on the Linux platform Lex [4] and Yacc [5]. Lex is a tool for generating scanners: programs with recognized lexical patterns in text. YACC is a tool for generating parsers. More information on these two tools can be found in the *man* pages.

The NetSpec script has two major pieces of information — schedule information, which tells the controller how it should execute the various phases of the daemons; and daemon information, which tells the controller the various daemons it needs to invoke on the specified hosts. However, the schedule information is optional. If the schedule information is not present, the controller executes the daemons in 8-phases as discussed previously in the NetSpec 8 phase classic behavior.

5.1.1 Schedule Information

Schedule information is composed of one or more schedule blocks, wherein each blocks represents the schedule for a particular daemon type. A schedule block is shown in Script 5.1.

Script 5.1 NetSpec Script Schedule Block

```
map defmap {
  daemon = new_daemon(daemon_name=daemonA;
               setupCommand = 1;
               openCommand  = 2;
               runCommand   = 3;
               finishCommand = 4;);
}
```

The above block represents the scheduling information for a daemon of type *daemonA*.

Our schedule block will be represented in the parser as follows:

```
scheduleBlock:
MAP IDENTIFIER '{' newDaemonInfo '}'
;
```

Words in caps refer to tokens (*MAP* refers to the word *map* in the schedule block shown above. *IDENTIFIER* refers to *defmap* above). *newDaemonInfo* is the nonterminal (as it can be parsed further, unlike tokens which indicate the end of parsing a particular term). It represents the information present in between the braces {, } in the schedule block shown above.

This information is then stored in the following schedule data structures,

```
struct daemon_phase_list {
  char *daemon;
  char *phaseCommand;
  struct daemon_phase_list *next;
};
```

The structure *daemon_phase_list* stores the names of the daemon type and the phases will need to be executed in a particular slot.

```
struct slot_info {
  int slotId;
  struct daemon_phase_list *assoc;
  struct slot_info *next;
  struct slot_info *prev;
};
```

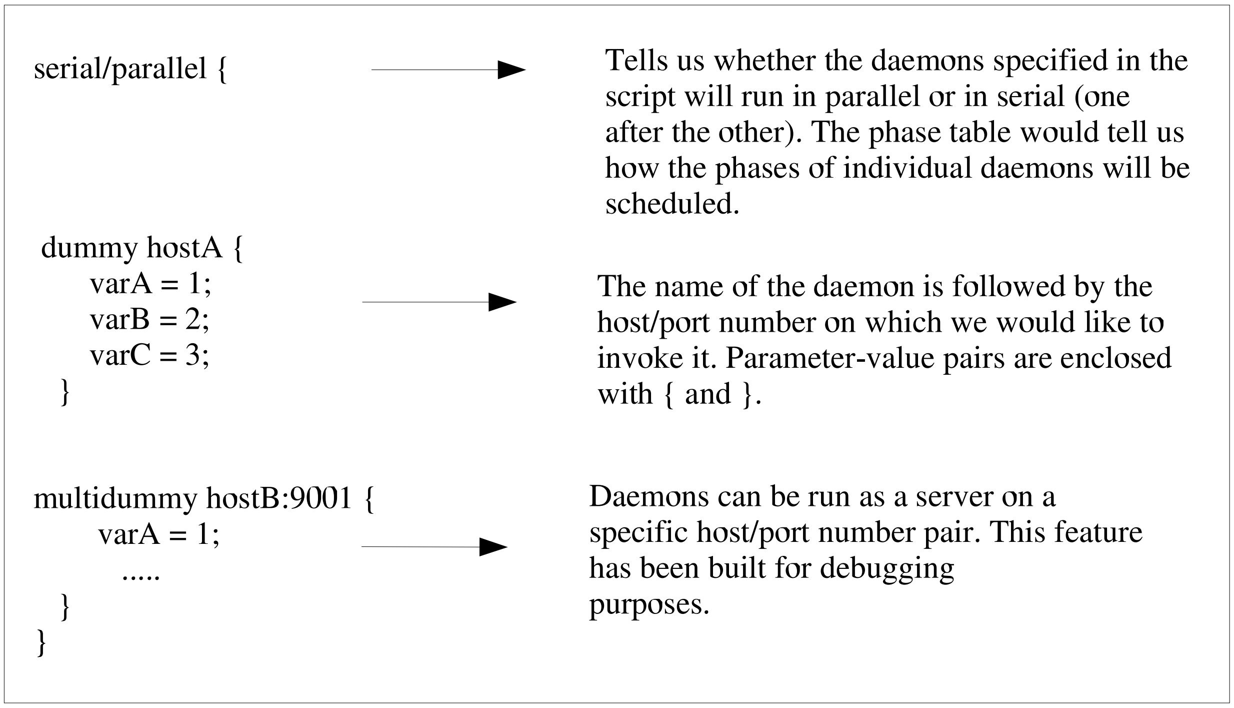
The structure *slot_info* maintains a doubly-linked list of all the slots through which a daemon execution proceeds. From the schedule block shown above, when we say, *setupCommand = 1*, we mean that the *setup command* needs to be executed in *slot 1*. Therefore, there are in total 4 slots and in each slot, we could schedule phases of various daemon types. As can be seen, the structure *slot_info* has a pointer to the list which has information about daemons and phases which need to be run in that particular slot.

5.1.2 Daemon Information

Daemon information consists of two major components — specifying the execution construct (*cluster/parallel/serial* for Classic 8 phase NetSpec behavior (or) *parallel/serial* for Variable Phase NetSpec) and specifying the daemons which have to be invoked on the specified hosts along with the data in the form of parameter-value pairs which needs to be passed to the daemon.

Figure 5.2 shows a sample script and explains what each portion of the script means.

Figure 5.3 shows the data structure used to store the parsed daemon values and explains each element in the structure.



```
serial/parallel {

 dummy hostA {
      varA = 1;
      varB = 2;
      varC = 3;
   }

multidummy hostB:9001 {
       varA = 1;
          .....
   }
}
```

Tells us whether the daemons specified in the script will run in parallel or in serial (one after the other). The phase table would tell us how the phases of individual daemons will be scheduled.

The name of the daemon is followed by the host/port number on which we would like to invoke it. Parameter-value pairs are enclosed with { and }.

Daemons can be run as a server on a specific host/port number pair. This feature has been built for debugging purposes.

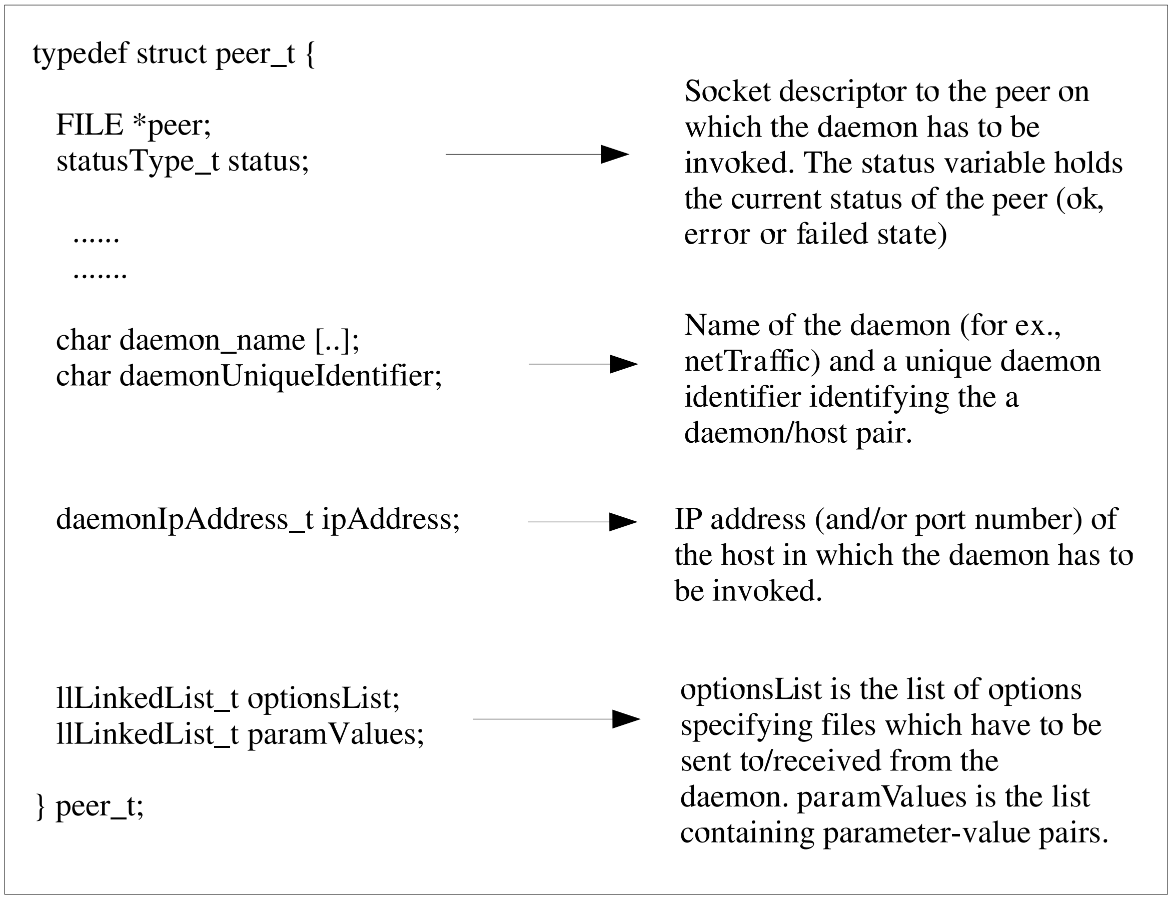
Figure 5.2: NetSpec Sample Script

5.2 Control Logic

The various control operations undertaken by the controller are enumerated below:

5.2.1 Connecting to the Peer

The linked list containing the list of daemons obtained from the daemon portion of the script is taken as input and we begin the control actions by connecting to each one of the specified peers (or hosts) and if the connection is successful, we pass the parameter-value pairs to the respective peers. The connections are made using TCP-based sockets (so we have a reliable connection which takes care of any errors which might occur during transmission). While sending the parameter-value pairs to the peer, we perform proper error checking to ensure that the peer has not crashed once the connection has been established. If the peer crashes after the connection has been established,



```
typedef struct peer_t {

  FILE *peer;
  statusType_t status;

    ......
    .......

  char daemon_name [..];
  char daemonUniqueIdentifier;

  daemonIpAddress_t ipAddress;

  llLinkedList_t optionsList;
  llLinkedList_t paramValues;

} peer_t;
```

Socket descriptor to the peer on which the daemon has to be invoked. The status variable holds the current status of the peer (ok, error or failed state)

Name of the daemon (for ex., netTraffic) and a unique daemon identifier identifying the a daemon/host pair.

IP address (and/or port number) of the host in which the daemon has to be invoked.

optionsList is the list of options specifying files which have to be sent to/received from the daemon. paramValues is the list containing parameter-value pairs.

Figure 5.3: NetSpec Data Structure

proper error-checking measures have been built-in to ensure that further parameter-value pairs are not sent to it, by setting the status flag for the peer appropriately.

5.2.2 Execute the Phases

Based on whether the schedule information is present or not, we either execute NetSpec Variable Phase or the Classic 8 Phase. We present pseudocode for each of these cases. Both mechanisms have been explained in Chapters 3 and 4.

5.2.2.1 Classic 8 Phase NetSpec

Classic eight phase NetSpec provides three execution constructs — serial, parallel and cluster. Only the open, run and close phases are subject to the execution constructs. The remaining 5 phases are executed serially. The pseudocode for each of the execution constructs *serial*, *parallel* and *cluster* is presented below:

Program 5.2 Classic 8-Phase: Serial construct

```
for (all the daemons specified in the script)
{
  check the status of the peer on which the
  daemon has to be invoked;

  if the peer is ok {
    send the string 'openCommand' to the peer using the
    socket connection established before;
    /* This prompts the daemon to execute it's open phase */
  }

  wait for reply from this peer;
  /* Getting a reply would mean that the open
     phase finished successfully */

  check the status of the peer;
  if the peer is ok {
    send the string 'runCommand' to the peer
  }

  wait for reply from this peer;

  check the status of the peer;
  if the peer is ok {
    send the string 'closeCommand' to the peer;
  }
  wait for reply from this peer;
} /* end of for loop */
```

5.2.2.2 Variable Phase NetSpec

Variable phase NetSpec was explained in previous sections. Here, we present the pseudocode for the execution constructs *Serial* and *Parallel* supported by Variable Phase

Program 5.3 Classic 8-Phase: Parallel construct

```
for (all the daemons specified in the script) {
  check the status of the peer on which
  the daemon has to be invoked;

  if the peer is ok {
    send the string ’openCommand’ to the peer
    using the socket connection initially established;
    /* The string ’openCommand’ is a directive to the daemon to
       start it’s open phase*/
  }

  wait for reply from this peer about the
  result of the open phase;

  re-check the status of this peer;
  if the peer is ok {
    send the string ’runCommand’ to the peer;
  }
} /*end of for loop */

for (all the daemons specified in the script) {
  wait for the reply from the peer about
  the result of the open phase;

  re-check the status of this peer;
  if the peer is ok {
    send the string ’closeComand’ to the peer;
  }
  wait for the reply from the peer about the
  result of the close phase;
}
```

NetSpec. Note that schedule information is provided in two-levels. On the first level we have the *serial* and *parallel* execution constructs tell us how individual daemons will be scheduled and on the second level we use the slot-based mechanism to specify how phases of daemons participating in the experiment will be scheduled.

Program 5.5 shows the pseudo-code for the *serial* construct and example 5.6 is the pseudo-code for the *parallel* construct.

Program 5.4 Classic 8-Phase: Cluster construct

```
for (all the daemons specified in the script) {
  check the status of the peer on which
  the daemon has to be invoked;

  if this peer is ok {
    send the string ’openCommand’ to this peer;
    /* The string ’openCommand’ is a directive
       to the daemon to start it’s open phase*/
  }
} /*end of for loop*/

for (all the daemons specified in the script) {
  wait for reply for ’openCommand’ from peer;
}

for (all the daemons specified in the script) {
  check the status of the peer on which
  the daemon has to be invoked;

  if this peer is ok {
    send the string ’runCommand’ to this peer;
  }
} /*end of for loop*/

for (all the daemons specified in the script) {
  wait for reply for ’runCommand’ from peer;
}

for (all the daemons specified in the script) {
  check the status of the peer on which the
  daemon has to be invoked;

  if this peer is ok {
    send the string ’closeCommand’ to this peer;
  }
} /*end of for loop*/

for (all the daemons specified in the script) {
     wait for reply for ’closeCommand’ from peer;
}
```

Program 5.5 Variable Phase: Serial construct

```
for (all the daemons which need to be invoked) {
  for (all the slots present in the phase table) {
    for (all the phases of daemons present in the particular slot) {
      /* sending the command to the peer indicates initiating
       * a particular phase
       */
      send the command string to the appropriate peer;
      wait for reply;
    }
  }
}
```

Program 5.6 Variable Phase: Parallel construct

```
for (all the slots present in the phase table) {
  for (all the phases of daemons present in a particular slot) {
    for (all the daemons which need to be invoked) {
      /* sending the command to the peer indicates initiating
       * a particular phase
       */
      send the command string to the appropriate peer;
    }
    wait for reply from all the peers to which the command was sent;
  }
}
```

Chapter 6

Testing

The new NetSpec controller was tested for correctness as well as robustness. The newer design should not only work correctly, but should also be able to run scripts which invoke many daemons. In this chapter, we discuss testing measures which were employed to test the features discussed in the previous sections.

6.1 Testing Correctness Feature

6.1.1 Pretty Printer Option

This option was employed to ensure that data structures are filled correctly by the parser. Before the data structures can be used by the NetSpec controller to perform the necessary control operations, the pretty printer options use the data structures filled by the parser to reconstruct the NetSpec input script. The output script from the pretty printer can then be compared with the input NetSpec script given to the parser using the GNU [3] *diff* tool. If there are any differences between these two files, then the data structures were not filled correctly by the parser. This test enables us to find any errors in parsing before testing the correctness of the control logic. One of the advantages of this option is that if we have a script which is formatted badly, we can turn on the pretty printer option and the output script so obtained is formatted neatly with proper indentation, etc., hence the name *pretty printer*.

6.1.2 Testing Control Logic

Even though the NetSpec controller design and architecture underwent a major overhaul, its interface to the daemons did not change. This made it easier to test the correctness of the control logic. The *Remote Control and Information Protocol (RCIP)* was used to communicate between the controller and the daemons in the newer and older designs of NetSpec. Thus there was not a great deal of change on the daemon's side due to the changes in the controller design. The daemons still execute in a phase-based manner with the controller being responsible for initiating the various phases. The *dummy* and the *system command* daemons come in very handy here, as both daemons are simple and the results from these daemons are known in advance.

The *dummy* daemon was developed to illustrate the working of variable phase NetSpec. A sample *dummy* daemon is shown in Script 6.1. The *dummy* daemon takes the four variables and values as part of the parameter-value pairs. It adds two variables in each phase and their results in a subsequent phase. The *dummy* daemon adds the four variables passed to it and the result is then sent back to the controller.

Script 6.1 NetSpec Dummy Daemon

```
map defmap {
  daemon = new_daemon(daemon_name=dummy;
           phase1Command=1;
           phase2Command=2;
           emptyphaseCommand=3;
           phase3Command=4;);
}

serial {
  dummy testbed61:9001 {
    varA = 1;
    varB = 3;
    varC = 5;
    varD = 7;
  }
}
```

The *system command* daemon provides the ability to construct a desired command line on the target system and then invoke it using the *system* system call, which invokes it from within a program as if it had been typed on the command line. For example, if a user wants to run a set of commands in x different systems, it would be rather tedious to log into each system and run these commands. Instead the user can pass those commands, through the scripts to the system daemon. The *system command* daemon will then execute them from a centralized controller system. As the results of the command we would like to execute are known in advance, the *system command* daemon proves to be very useful. A system command daemon is shown in Script 6.2.

Script 6.2 NetSpec System Command Daemon

```
map defmap {
  daemon = new_daemon(daemon_name=nssyscmd;
           setupCommand=1;
           openCommand=2;
           runCommand=3;
           closeCommand=4;
           finishCommand=5;);
}

parallel {
  nssyscmd waldorf:8001 {
    cmd="rm -rf /tmp/route.txt";
    cmd="ls > /tmp/route.txt";
    filename="/tmp/route.txt";
  }

  nssyscmd testbed61:8001 {
    cmd="rm -rf /tmp/route.txt";
    cmd="ls > /tmp/route.txt";
    filename="/tmp/route.txt";
  }
}
```

6.2 Testing Robustness Features

The NetSpec framework was tested for robustness by executing scripts which involve many daemons invoked across various hosts. The controller was also tested for its ability to handle large amounts of report data sent to it from individual daemons. Tests involving 40 NetSpec traffic daemons on eight hosts were performed and report data generated exceeded 16 MB. Conducting these tests and obtaining the results showed us that the new controller design was indeed robust and could handle very high loads.

”NetSpec traffic daemon is the most commonly used daemon, among the family of NetSpec daemons. It is mainly used for traffic generation and throughput measurement. The traffic daemon can generate and sink a variety of TCP and UDP traffic types. After the experiment is complete traffic daemon generates report from the local statistics (socket properties and resource usage statistics) gathered from the kernel.” [6]. A portion of a script used in testing the NetSpec framework is shown in Script 6.3. The NetSpec service multiplexer, *netspecd*, can be started either in standalone or inetd mode on the hosts on which the daemons are to be invoked. The NetSpec controller then contacts the appropriate service multiplexer to invoke the various daemons in the experiment.

Script 6.3 NetSpec Traffic Daemon

```
netTraffic testbed28 daemon3 {
  type = burst (blocksize=9140, stamps=9000,
               period=14000, duration=120);
  protocol = udp (xmtbuf = 65536);
  own = testbed28:8001;
  peer = testbed33:8001:daemon4;
}

netTraffic testbed33 daemon4 {
  type = sink (blocksize=9140, stamps=9000,
              durationCorrection=60000, lingerCycles=2000,
              duration=120);
  protocol = udp (rcvbuf=65536);
  own = testbed33:8001;
  peer = testbed28:8001;
}
```

Chapter 7

Conclusions And Future Work

When we began redesigning the NetSpec controller, there were three goals that we set out to achive — simplicity, modularity and robustness. Separating the parsing and control logic functionalities was an important first step, which resulted in the design being simpler. It also helped in making the code modular. We also had to ensure that the new design be able to handle many daemons invoked across various hosts.

Redesigning the NetSpec controller not only helped us achieve the above-mentioned goals, but has also resulted in adding new capabilities to the controller like providing a mechanism for downloading files to the daemons, which would be used by the daemon for its processing needs. Options like *Pretty printer* (explained in Chapter 6) have been provided, which help in testing controller code better. Ability to provide comments in the script and print out line and column number information when the controller encounters errors while parsing the NetSpec script are some of the many changes which have been made to the controller.

Future Work

The NetSpec script consists of the scheduling information in which we specify how we wish to schedule the phases of various daemons along with the set of daemons which we plan to invoke. Let us define a semantic termed *Group* which represents a block having schedule and daemon information. We could have a script containing various *Groups* and we could specify how we plan to schedule the various *groups*. We could schedule *Groups* in parallel, serial or a combination of both. The *Group* semantic is powerful, as it will help us to group together daemons (scheduling them according to the schedule information provided) and also schedule the various groups, based on additional information provided.

Bibliography

[1] Inc. Berkeley Software Design. http://www.bsd.org.

[2] Hewlett-Packard Company. Netperf manual. http://www.netperf.org/netperf/NetperfPage.html.

[3] Free Software Foundation. http://www.gnu.org.

[4] Bell Laboratories. http://plan9.bell-labs.com/magic/man2html/1/lex.

[5] Bell Laboratories. http://plan9.bell-labs.com/magic/man2html/1/yacc.

[6] Karthikeyan Nathillvar, Dr. Douglas Niehaus, and Anupama Sundaresan. Netspec technical report. Technical report, ITTC, 2002.

[7] Inc. Netcordia. Test tcp whitepaper. http://www.netcordia.com/tools/tools/TTCP/ttcp.html.

[8] Cisco Systems. Using test tcp (ttcp) to test throughput. http://www.cisco.com/warp/public/471/ttcp.html.

[9] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf version 1.7.0. http://dast.nlanr.net/Projects/Iperf/.