ABSTRACT

ARORA, PUNEET. Netset – A Software Framework for Automation of Network based Tests. (Under the direction of Dr. Injong Rhee).

Testing network protocols or networking devices for generating performance benchmarks is integral to computer networking research. The general pattern in conducting such tests is configuring a desired topology over a network test-bed, deploying test software and tools onto network nodes, setting up load traffic on the network, executing the deployed software, gathering data and generating reports for analysis and comparison. A typical example of such a process is performance testing of congestion control protocols. Congestion control protocols are designed to satisfy a complex set of goals and their performance is sensitive to network topology, network delays, and router queue sizes and policies. It is thus non-trivial to construct testing procedures for congestion control protocols.

While there exist network emulation test-bed services like Emulab, WanInLab, Deter etc. they limit themselves to merely providing a set of nodes which the developer can use for testing. They provide no means of setting up the test environment, generating test traffic, gathering data and generating reports. As a result developers tend to write their own automation procedures to carry out these steps. Such tendency restricts portability, repeatability and comparison of test procedures and their results. This lack of a generally accepted practice or of a testing tool affects many parts of the networking research community including researchers, students, standardization bodies and developers.

To address this issue, we develop a tool for network based testing called Netset. The tool implements a general model which accommodates a wide range of testing processes and provides programming interface to develop specialized tools for a particular category like congestion control protocols. We then apply this tool to two testing scenarios and compare its benefits versus using a manual approach in the same scenarios.

Netset – A Software Framework for Automation of Network based Tests

by
Puneet Arora

A thesis submitted to the Graduate Faculty of
North Carolina State University
In partial fulfillment of the
Requirements for the degree of
Master of Science

Computer Networking

Raleigh, North Carolina

2008

APPROVED BY:

Dr. Injong Rhee Committee Chair	Dr. Khaled Harfoush
Dr. Laurie Williams	

DEDICATION

To my parents …

BIOGRAPHY

Puneet Arora was born on March 2, 1983 in Mumbai, India. He graduated with a Bachelor of Engineering degree in Electronics and Telecommunications from Dwarkadas J. Sanghvi College of Engineering, Mumbai University, India in July 2005. After completing his under-graduate program, he worked for nine months with Wcities Content Solutions India Pvt. Ltd. as a Software Engineer before arriving at North Carolina State University to pursue a Master of Science in Computer Networking. During the MS program he worked with Tekelec at Morrisville, NC as a Test Automation Engineer – Summer Co-op for two summers.

ACKNOWLEDGEMENTS

I would like to extend my sincere gratitude to my advisor Dr. Injong Rhee for giving me an opportunity to work with him to pursue this thesis. The work could not have been successfully completed without his support and motivation. I also thank Dr. Khaled Harfoush and Dr. Laurie Williams for participating in my thesis committee.

I thank my friend and colleague Sangtae Ha for educating me about his TCP testing work, sharing his expert knowledge with me and keeping me motivated by reviewing my work regularly.

I thank John Bass, Technical Director of the Centennial Networking Lab at NCSU, for supporting my work, helping me interface it with NCSU's Virtual Computing Lab infrastructure and for sharing his networking testing experience with me. I would also like to thank Tim Brady, Manager of Test Tools at Tekelec, Morrisville, under whom I worked during my two summer internships. The summer work was related to network test automation and the experience proved instrumental while developing Netset.

I thank my friends Santosh, Deepthi and Archana for their interest in my work, which served as a motivating factor. Their review and feedback was quite useful while writing this document.

I thank my family and friends in Mumbai for their unwavering love and support, without which I could not have been successful.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

One of the prominent topics in computer networking research is of developing new protocols. As the Internet evolves in its size, capacity, and characteristics, many new types of protocols at network layer, transport layer etc. will be developed as the current ones are unlikely to continue to satisfy the distinct requirements of the growing and diversifying networks of today. Our system and process of inventing, endorsing and adopting protocols is however significantly stymied, we believe, mainly because of lack of standard performance evaluation procedures and tools for evaluating the performance of new protocols.

Testing a protocol for better performance against existing ones or against other contenders is integral to computer networking research. The general pattern in conducting such tests is configuring a desired topology over a network test-bed, deploying test software and tools onto network nodes, setting up load traffic on the network, executing the deployed software, gathering data and generating reports for analysis and comparison. A typical example of such a process is performance testing of new transport protocols or that of an existing one like TCP with new congestion control algorithms. Transport and Network protocols are designed to satisfy a complex set of goals and their performance is sensitive to network topology, network delays, and router queue sizes and policies. It is thus non-trivial to construct testing procedures for these protocols. While protocols are the key areas that require network based tests, there are other areas as well, where a network based test environment is required. These include performance testing of network devices like routers, switches etc., network application servers, telecommunication application servers etc.

When testing in a network environment there are many aspects that affect the test results. These include network topology, lossy links, delays, buffer sizes, existence and kind of background traffic etc. Any network based test should involve setting these parameters to different values to investigate how the results change with their variations, which in turn shows the merits or demerits of the protocol or device under test. Adjusting the values of these network parameters involves significant effort and is a task regularly undertaken by researchers who actively test protocols or network devices. The lack of a standard automation tool to automate this process results in duplication of effort by the researchers.

There exist several projects and services, such as Emulab, WanInLab, PlanetLab and WhyNet, which build laboratory network test-beds for Internet research using local infrastructure or overlay over the Internet. While these test-beds provide physical environments for network testing, they do not create the operational environments that include testing scenarios, background traffic generation, performance metric and test reporting. This lack of a generally accepted practice or of a testing tool affects many parts of the networking research community including researchers, students, standardization bodies and developers.

As a solution to this problem, we develop a network test automation tool called Netset and discuss the approach and its architecture in this thesis. Netset provides a framework to integrate various network testing resources under one roof. It implements a general model, which accommodates a wide range of testing processes. It provides its own test-bed configuration module and can also interface with existing test-bed services. Further it provides a mechanism to specify tests in a portable form that includes topology specification, environment and traffic specification and report formats. These portable tests can be shared amongst researchers to reduce duplication of effort and increase efficiency and repeatability. It also makes it convenient for central authorities to publish reference test cases and for third-party reviewers to compare test results from different researchers.

We apply netset to two testing scenarios and compare its benefits versus a manual approach in the same scenarios. We conclude with a basic implementation of Netset and list possible future work that can greatly enhance the usefulness of netset.

Chapter 2

Context

This chapter discusses the nature and scope of Network Based Tests. It describes various testing scenarios, test parameters, workloads and performance metrics. It also discusses related work done towards testing on network test-beds.

2.1 Network Based Tests

A few examples of test processes that require a network test-bed environment are testing of congestion control protocols (or altogether new transport protocols), testing of routing protocols, testing of network interface driver software, testing of application protocols (like network file-systems) and testing of new network and transport protocols. For conducting these tests, network environments are defined to include network parameters that define network topology, workload generation that determines the patterns of background traffic, testing scenarios that guide the timing, placement and number of background flows, and network performance measurement that involves tracing protocol-related variables (metrics) and monitoring network health. We briefly discuss these components of a test network environment.

2.1.1 Network Parameters

The performance of network protocols and devices depends on various network parameters that include network topology, network delays, router buffer sizes, the amount of congestion, traffic mix and flow arrival and departure rates (or variance in available bandwidth). A test automation tool must provide convenient ways for users to control these parameters for their testing. Table 2.1 summarizes these parameters of interest in network testing, their values, and techniques to control them. The number-listed text that follows briefly describes each of these parameters.

Table 2.1: Summary of parameters of interest in network testing

Network Parameter	**Typical Values / Range of Values**	**Mechanisms to control its value**
Topology	Dumbbell, Parking Lot, Ring etc.	By switch configuration in local test beds of by using third-party arbitrary topology provider services like Emulab [EMULAB]
Delays	Various distributions	By setting up Dummynet [Ri97] delay queues for each class of delays divided into a fixed granularity of delays
Router buffer sizes	Small (e.g. 1MB) to BDP [1] of flows with the longest RTT[2]	By resizing Dummynet buffers
Congestion	Low to high	By controlling the number of medium or long term flows
Traffic Mix	The ratio among various application flows (e.g. HTTP, VoIP, FTP, P2P)	By controlling the mix of UDP and TCP flows with various lifetime and transmission rates

[1]BDP: Bandwidth Delay Product, [2]RTT: Round Trip Time

Table 2.1 Continued

Network Parameter	**Typical Values / Range of Values**	**Mechanisms to control its value**
Available Bandwidth Variance	Low to high	By controlling the parameters of Pareto distribution for short and medium flows, and controlling the range of flow sizes
BDP	Small (a few 10s) to very large (several 100,000s)	By controlling the bandwidth and delays of Dummynet links
Packet loss rate	Low to high	By controlling random loss rates at the routers

1. *Network topology* governs the paths that flows travel to compete with each other. Network topologies can influence the performance of protocols by introducing varying numbers of bottlenecks and also varying different routing paths for forward and backward paths. There could be many diverse topologies that could bring a new insight into protocol behaviors. The network test-bed must be flexible enough to handle a diverse set of topologies. Topology configuration can be automated by automating configuration of switches in the local test-bed or by using third party arbitrary topology provider services like Emulab. Typically, topologies used in network testing include, but are not limited to, dumbbell, parking lots, ring, etc.

2. The *delays* of cross-traffic flows emulate the effect of statistical multiplexing at routers adding randomness and dynamics essential for realistic network environments. This is because most flows (e.g. TCP) maintain a closed-loop with their receivers and they control their rates at a timescale of RTT. From Internet measurement studies, it is shown that the network delays of flows have various

distributions [AKSJ03], including exponential distribution. As the Internet becomes more diverse, the distribution changes from exponential distribution to more heavy-tailed distributions. Thus, a comprehensive test of protocols must include diverse distributions of RTT.

3. The *buffers* at bottleneck routers are used to dampen the effect of rate variations of traffic. The size of these buffers plays a key role in determining the stability of the network and also network flows [AKM04]. If the buffer size is too small, the network tends to be less stable as it becomes unable to accommodate rate variations present in traffic. If it is too large, it tends to increase network delays. In a typical test configuration, it is common to fix the buffer size to the estimated value of the maximum bandwidth-delay product (BDP). However, there is an increasing trend for use of smaller buffers than the BDP. Thus, when testing performance of a transport protocol for example, it is desirable to maintain good stability under various sizes of router buffers. The size of buffers can be controlled easily by adjusting the amount of buffer memory at routers.

4. A transport protocol under test must handle various levels of *congestion* as a network path may undergo a diverse degree of congestion. Under any network condition, a protocol must strive to achieve high utilization and at the same time, share the path bandwidth fairly with competing flows. Adjusting the arrival rates of background flows can control the degree of congestion. The variability of congestion over time is an important factor in determining the behaviors of protocols as it could affect the stability, fairness and responsiveness of protocols. Congestion variability can be controlled by adjusting the parameter values used in generating cross-traffic.

5. As the networks with high *bandwidth-delay products* become more abundant, the scalability of a transport protocol in utilizing the BDP becomes important. The

range of BDP typically being considered is in the range of a few ten packets to a few hundred thousand packets for a packet size of 1000 bytes (for instance, 10 Gbps network with 100ms delay results in 125,000 packets). We can control the BDP of links by controlling the bandwidth and delays of Dummynet links.

6. *Packet losses* are an important factor to the performance of protocols since most protocols use packet losses as congestion signals. However, in some mediums such as satellite and wireless links, packet losses are not necessarily caused by congestion, but by signal losses such as fading and interference. In order to test protocols under these environments, routers can randomly drop packets at a rate determined by testing scenarios.

2.1.2 Workload Generation

Workload and traffic generation has recently become an active area of research. Most of the existing work is related to creating traffic for a particular type of Internet flow [TTCP, IPerf, NetPerf, Thrulay] or simply providing a vessel for traffic generation [TG, MGEN, NetSpec, Rude] in which a user can specify a network model and a tool is provided to generate traffic based on the model. These tools are useful if we have a realistic network model.

In general, two types of network models exist for traffic generation: packet-level models and source-level models. Packet-level models [DITG, PsnTraf, SSTraf] characterize the aggregate traffic on a link as some stochastic process typically parameterized by measures of packet size, packet arrival rate, and packet inter-arrival time. While packet-level models have their use, a disadvantage, in particular for congestion control research, is that the models inherently are network-dependent. They include the effects of congestion on the sources that were generating the traffic and, hence, can only accurately reproduce the network conditions from which they were derived.

In contrast, source-level models characterize the end-systems' use of the network and, thus, are network- independent (to the extent that an implicit axiom that sources do not adapt their behavior based on perceived network conditions holds). Because they are network-independent, source-level models are generally preferred over packet-level models [FP03] because they can be applied in environments that are different from those from which they are derived.

Most source-level models are application-specific models, e.g., models of web traffic, file transfer, etc. The model typically consists of a state machine (with non-deterministic transitions) that emulates user (or server) behavior and drives the use of the application-level protocol. Popular source-level traffic generators include [TrafGen, GenSyn, Surge, FGHW99].

For a network test automation tool to be complete, it should provide a framework to implement these models or interface with their existing implementations to set up flexible test scenarios.

2.1.3 Performance Metrics and Testing Scenarios

Performance metrics are values of interest that are observed in or calculated from the results of a test. For example, when testing a new transport protocol (or a new congestion control algorithm applied to an existing transport protocol like TCP), there are a number of desirable properties to be observed that characterize the performance of the protocol. These properties include, but are not limited to, protocol stability, fairness, convergence and scalability. Testing scenario specifications involve the placement and timing of flows of the protocols being investigated, in the network environment being setup. These scenarios are designed to highlight the specific properties of protocols and to be more conducive to measure these performance characteristics. Below we discuss the typical performance metrics for transport protocols and their corresponding scenarios that are commonly used in practice. Floyd [Flo06] also gives a nice discussion on these performance metrics.

TCP-friendliness: As new protocols other than TCP are introduced, their behaviors tend to deviate from those of TCP (Reno style). Because a majority of flows in the Internet are TCP, the 'friendliness' of these new protocols to competing TCP flows becomes important. The TCP-friendliness of a protocol is measured by the throughput ratio (a also called throughput fairness index) between flows of that protocol and competing TCP flows.

Convergence speed: This metric is the time taken by two or more competing flows of the same protocol to enter an equilibrium state. Since the equilibrium state is hard to define, the point in time where all flows converge to an approximately equal transmission rate is typically used in practice [LLS06]. An experiment to measure convergence speed must be conducted under various types of background traffic because protocols may exhibit substantially different convergence times even under a small amount of randomness in the timings of packet losses. Various types of background traffic can be obtained by varying its flow size variance and arrival rates, which tend to govern the rate variance of their generated traffic. We also need to try many possible combinations of network parameter values discussed in Section 2.1.1.

Intra-protocol fairness: This property of a protocol determines how equitably competing flows of the same protocol share the bottleneck bandwidth. We measure the long-term and short-term throughput ratio, or Jain's fairness index [JFI], of throughput that each flow obtains in a given time scale. For realistic measurement, this property should be tested with inclusion of background traffic and many different combinations of network parameter values discussed in Section 2.1.1.

RTT-fairness: This metric pertains to the bandwidth share of competing flows of the same protocol, but with different RTTs. In the Internet, flows may traverse through many diverse paths experiencing different round trip delays with other flows on a bottleneck link. Then unless care is taken, flows with short RTTs may take much more bandwidth than those with large RTTs. Although it is rather controversial whether flows with different RTTs should share the same bandwidth since the flows with larger RTTs tend to use more network resources, it is clearly accepted that short-RTT flows must not completely starve long-RTT flows. The testing scenarios for this metric are very similar to those of convergence speed except that the protocol flows may have different RTTs.

Efficiency (or scalability): The efficiency of a protocol is a measure of its ability to fully utilize the available bandwidth of the bottleneck link(s). One of the ways to measure this is to run the protocol under various random loss rates. We can introduce the losses at the bottleneck links by artificially dropping packets at an input rate.

Stability and responsiveness to changes: A protocol's stability means its ability to adapt to the changes of network environments to reach equilibrium even after significant perturbation in the network conditions. Its responsiveness is a measure of time it takes to return to equilibrium after a change. Testing scenarios for stability and responsiveness must include a high level of fluctuations (or changes) in network environments. These can be caused by introducing large variance in available bandwidth which can be controlled through a statistical distribution of flow sizes, flow arrival rates and flow delays of the background traffic.

Other metrics and scenarios: There are other metrics such as deployability, robustness to failure and other user-defined metrics for which it is hard to construct a meaningful scenario in advance. A test automation tool must thus provide a convenient way to construct new, arbitrary testing scenarios.

2.1.4 Performance Monitoring, Representation and Data Management

Performance monitoring, representation and data management deals with extracting values of interest (like the metrics described in section 2.1.3) from test results (which are log files in most cases), performing calculations on them if needed, representing them as convenient entities like tables and graphs and finally generating reports for analysis. To get results earlier during a test execution, it may be preferable to see the values of interest in real-time while a test is executing. A test automation tool should provide convenient methods for these operations and should also provide a mechanism for storage and organization of test results for future reference.

In some cases, a value obtained from one stage of the test may be used as input to another stage. A test automation tool should thus have provisions to connect data from one test stage to another.

2.2 Related Work

Floyd and Kohler [FK02a] propose to build a rich understanding of the range of realistic models, and of the likely relevance of different model parameters to network performance for various Internet research areas including congestion control, AQM and router functions. IETF Transport Modeling Research Group (TMRG) [TMR] is also investigating performance metrics and testing scenarios that can be used for simulators and emulation test-beds. While these projects build a foundation for understanding network models and provide recommendations for test scenarios and metrics, they are not readily available in a more accessible form for integrated use with other aspects of network based testing.

There are several projects and services, such as Emulab [EMULAB], WanInLab [WIL], PlanetLab [PLANET] and WHYNET [WHYNET] which build laboratory network test-beds for Internet research using local infrastructure or overlay over the Internet. These projects have established procedures that allow public access of their test-beds. While these test-beds provide physical environments for network testing, they do not create the operational environments that include testing scenarios, background traffic generation, performance metric and test reporting customizable to the protocol or device under test. There is a lack of an automation standard or tool that augments them with provisions to simplify and automate creation and execution of complete tests.

There exist many studies [ZCD97, TGJS02, FK02, DPMC04, TrafGen, YSVS00, SHJO01, LAJS03, HJS04, HAR] on network modeling for simulation, as well as measurement-based traffic workload generation. Most of these have also been implemented for use in tests. However, there is a need for an automation tool that will leverage these studies by integrating their tools and models for the purpose of building a more standardized benchmark suite in which network based tests can be conducted.

The community tool *Network Simulator* (NS-2) is widely accepted by the networking research community to run network test simulations. Henderson et al. [NS3] propose to extend NS-2 to improve its scalability and functionality for large simulation involving diverse types of networks. While NS-2 provides an integrated system to simulate network tests, there is a growing need for a tool that provides an integrated environment for running tests on real test-beds.

2.3 Motivation and Need

We have seen that an evolving Internet and its protocol suites necessitate significant efforts in the direction of developing new testing suites to evaluate and measure their performance. The testing framework and test-beds of today do not necessarily meet an entire set of criteria for the research community in terms of the ease of test case generation, portability of test cases, gathering data and representing the data in relevant formats. So we can see that testing a protocol requires more than just networking hardware test-beds such as Emulab [EMULAB] and WanInLAB [WIL]. It involves "software" components such as testing scenarios, realistic workloads, monitoring, performance metrics and their representation. These test scenarios, workloads and performance monitoring metrics along with the underlying hardware components constitute what we call a testing environment. A typical performance test of a protocol involves many steps: (1) Evaluators design a network topology where the protocol is tested on, (2) Conceptualize a real world workload for each node of the underlying network topology developed in step 1, (3) develop a test scenario which determines the arrival and departure of various competing or overlapping flows for the protocols being tested, (4) Provide the testing framework a specific set of protocol related parameters of interest to be collected and monitor the behavior of network devices and network's at various checkpoints, such as bottleneck routers during the experiment, (5) run the experiment and gather the data, (6) represent the measured data into performance graphs of the metrics, (7) Save the test case scenario for reuse or modification It is non-trivial to construct these testing procedures and environments and furthermore, prove such environments for realism of representing the target network environments where the tested protocol(s) are designed to run. The lack of an automation tool for this or at least a generally accepted practice has affected many parts of the Internet community including researchers, students, standardization bodies, developers and Internet users.

Newly developed Protocols need extensive measurement and analysis of their behavior in emulated but realistic scenarios. It is necessary to evaluate these new protocols against the ones that currently exist. To prove the correct functioning of a protocol (and publish it), researchers need to establish that their protocol performs better than or at least as well as existing protocols in many “common” test environments. The problem however is that there are no such “common” cases that everyone has agreed to use, which makes it all the more difficult for researchers to evaluate their protocol against a benchmark. Thus, researchers tend to rely on their ingenuity in creating testing environments and often spend significant effort in validating and motivating their own testing environments. Moreover, third-party reviewers of the work find it difficult to validate these “new-each-time” testing environments just based on the description in the published papers. Much of it is duplication of effort and more important, makes the testing results from different researchers almost incomparable. This also confuses the target users of the protocols because the results from different researchers can be conflicting. Moreover, researchers tend to blame each other’s test environments for any discrepancy in performance.

A generally accepted benchmark testing and evaluation tool can ease this situation. Our discipline can use a more objective evaluation of new protocols through standardizing a benchmark tool that makes testing convenient, test environments portable and the results verifiable and reproducible. At this point in our discussion, let us state that in network based tests, especially those for new protocols in which a large number of people or a community are involved, there are two problems – (1) Existence of benchmark test cases that the majority agree upon mutually (2) A standard framework or software to implement test cases. Promoting relevant discussions in the community and helping the parties involved to reach a conclusion, which may be subject to its own dedicated research, can solve problem 1. Problem 2 has a technical solution and can greatly accelerate the solution to problem 1. To solve problem 2, a software framework can be developed that incorporates network topology configuration, parameters and metrics described in the previous section in such a manner that is general enough to fit in any test case (or most test cases) that may be proposed by the community as a solution to problem 1. Below, we illustrate this discussion.

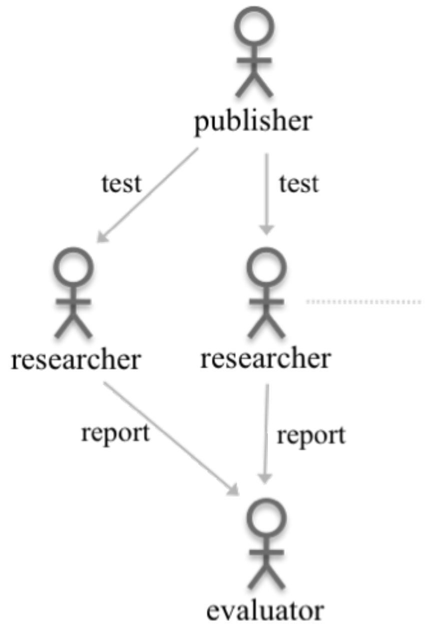
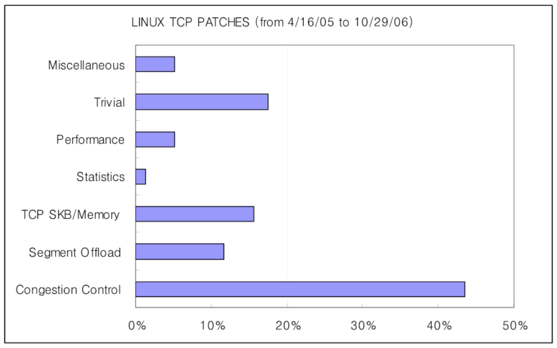


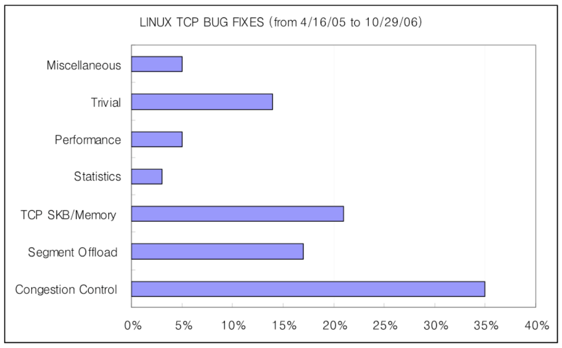
Fig 2.1: User roles in Network Testing

Fig 2.1 shows the preferred user roles in network testing, and their interaction. There should be a publishing entity, which may be a central authority like an IETF group that publishes benchmark test cases. The researchers test their protocols under these test cases and submit their results to an evaluator, which may be the same as the publishing entity. Here, an example of the 'protocol' that we are talking about could be a new Transport protocol or a new congestion control algorithm for TCP. The evaluator's job is to compare test results with those submitted by other researchers or that of existing protocols and conclude if one of the new protocols should replace an existing one. In fact, this is how it is currently done in the networking research community, though loosely. The test cases are published in the form of descriptive text in research papers and each researcher uses his ingenuity to implement them. As stated earlier, this approach results in duplication of work between the researchers, makes test specification interpretation more prone to errors, makes the results less comparable or repeatable and slows down the research and development process. In this scenario, if a test automation tool can be used to conveniently and completely define test cases in a portable form, then the publisher can publish the test case in the format specified by the tool. The researchers can 'plug-on' their protocol implementation into the given test case and execute the test to generate results and reports in a standard form, hence making it more efficient to compare results from different researchers.

Let us also look at some statistical evidence that testifies the need for better efforts in testing network protocols. From April 16, 2005 to October 29, 2006, 154 Linux patches have been made to the TCP stack of Linux versions 2.6.12 to 2.6.19 – almost 1 patch per every 2 to 3 days. Many of these versions are currently in use by millions of users around the world. Out of 154 patches, 77 patches (50%) are bug fixes, 58 patches (38%) are related to enhancements and cleanups, and 19 patches (12%) are related to new additions. Fig 2.2 (a) shows the breakups of patches in various categories. Fig 2.2 (b) examines only bug fix patches of Linux TCP stack. It shows that 35% of the total bug fixes made during that time period are directly related to congestion control, which happens to be a major research topic. Note that congestion control bugs are hard to catch because their operations and idiosyncratic behaviors can reveal themselves only under some specific operating conditions. Many of these bugs could have been avoided if thorough testing with a good testing suite had been performed before the release. Fig 2.3 shows the performance of H-TCP, one of advanced TCP stacks implemented in Linux 2.6.13. It shows the result of a test run involving two H-TCP flows with different RTTs. This version of H-TCP [HTCP] contains a bug (a simple misplacement of a parenthesis). The figure reveals a problem with H-TCP as long RTT flow uses more bandwidth. But this bug even escaped the scrutiny and testing by the authors. It was introduced in August 29, 2005 and fixed on March 20, 2006. A similar instance was reported with BIC-TCP [BIC] which until recently has been the default TCP stack of Linux. These instances illustrate how a benchmark testing suite could benefit the community in discovering these bugs before the release of the operation system. Note that since Linux is frequently downloaded by millions of users as the operating system for their servers and desktops, it would have a significant impact on the user community.

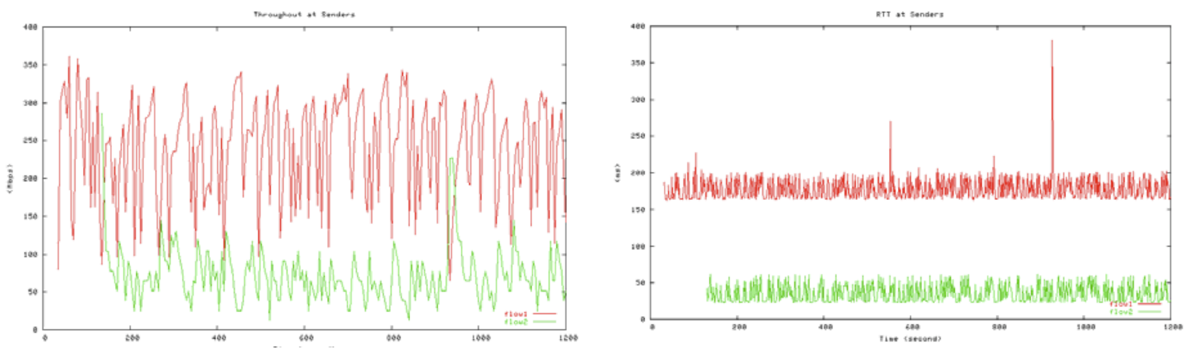


(a)



(b)

Fig 2.2 (a) Linux TCP Patches & (b) TCP Bug Fixes



(a) (b)

Fig 2.3: (a) Throughputs & (b) RTTs of two H-TCP flows

With this motivation, we proceed to discuss a possible solution in the following chapter.

Chapter 3

Netset – A Solution

The previous chapter discussed in detail the nature and scope of network based tests and pointed out the need to build an automation tool to carry out those tests. It also discussed the requirements, features and objectives of such a tool. Taking those points into consideration, in this chapter we discuss an approach to build an automation tool for network based tests called Netset.

3.1 Test Requirements and Objectives

We begin by summarizing the requirements and objectives that Netset must address –

1. In a test definition, hereafter simply referred to as *test*, the user should be able to specify a network topology (that includes nodes, links and Local Area Networks (LANs), network parameters), traffic information, flow timings, metrics to be gathered from the test, and report formats. Lets call these as the test components.
2. The set from which test components are chosen should not be hard-limited in the system. For example, a test can have any arbitrary topology and not limited to

choice from a preset list. Similarly, the types of traffic flows that can be defined in a test should not be from a finite set. There should be a provision for the user to conveniently add new types of traffic flows that can be used in a test.

3. The test should be portable, such that it could be saved as a file and transferred to another user who can use it. Further there should be a provision to change values of parameters used in the test.
4. The system should provide convenient integration with existing test-beds like Emulab.
5. The system should provide convenient integration with independent tools used in network testing like iperf and other traffic generators, so that it can leverage on existing work done for network testing.
6. The system should provide a way to capture produced data and store it for use in report generation.
7. The system should have a provision to specify report formats. Further it should have a way to manage reports for future reference and comparison.

The broad objective of the tool is to bring all network testing resources under one umbrella for integrated use.

Netset divides the network testing processes into two phases – the Network Layout phase and the Test Automation Phase. In the Network Layout phase, the system provisions the nodes that will be used for testing and connects them in way such that the topology specified by the test is achieved. In the Test Automation phase, the system carries out the rest of the test like starting the traffic and test flows, gathering data from the flows and generating reports.

3.2 Network Layout

The key actions in this phase are to provision nodes and to connect them as per the specified topology. For this purpose, it should be possible to use third party test-bed services like Emulab. We achieve this by defining a system entity called the Provisioning Module and defining a standard Application Programming Interface (API) using which the system communicates with it. This API includes functions to communicate the list of nodes and information about their connectivity to the test-bed service. By writing a separate provisioning module for each third party test-bed service like Emulab, a number of test-beds can be supported. Depending on the services provided by the test-bed, not all 'features' may be available. Some functions in the API are marked as mandatory while others are optional. As long as a test-bed provides features required by the mandatory API, it can be used with Netset.

Netset also features a built-in provisioning module that can work directly with a local test-bed infrastructure to implement a given topology on it. Here, one may ask that if Emulab can also be installed locally for use with a local infrastructure, why does Netset provide a provisioning module to use with the local infrastructure directly. Instead a provisioning module for Emulab can be used and in turn Emulab can access the test-bed and configure it. However, there are certain things that Emulab lacks, to overcome which we provide a 'direct-access' provisioning module with Netset. We first discuss how the Netset's native provisioning module works and then point out why we didn't rely on the use of Emulab alone.

An arbitrary topology can be configured on a network infrastructure provided the infrastructure contains layer-2 switches (L2) that support Virtual LANs (VLANs). More can be achieved if the infrastructure also contains layer-1 switches (L1). An L1 provides software configurable electrical (physical) connectivity between any two interfaces on it. An arbitrary network topology can be specified in a test in the form of a set of datalinks (connections[1]) between pairs of interfaces, sets of interfaces that belong to the same LAN and optionally the Internet Protocol (IP) address assigned to each interface. This topology can be realized on a test-bed by appropriate configuration of the L1s and L2s in it. Netset includes two algorithms to achieve this layout automation – one for automatic implementation of datalinks and another for automatic setup of LAN groups and trunk lines. These algorithms read the user's topology specification and calculate required settings for the L1s and L2s in the test-bed. They need to know the test-bed layout including L1s and L2s to work. The core connectivity information and list of nodes in the test-bed are thus stored in the system for use by these algorithms.

[1] We use the word 'datalink' instead of 'connection' as 'connection' is used for a different purpose in the context of Netset's layout automation algorithms. See 'Datalink Automation Datalinks'

3.2.1 Datalink Automation Algorithm

This algorithm deals with automatic configuration of L1s in a test-bed so as to achieve a datalink between two interfaces. We define a datalink as a physical path between two interfaces none of which belong to an L1. A connection however is defined as a physical path between two interfaces, one, both or none of which can belong to an L1. It is thus possible for a connection to also be a datalink.

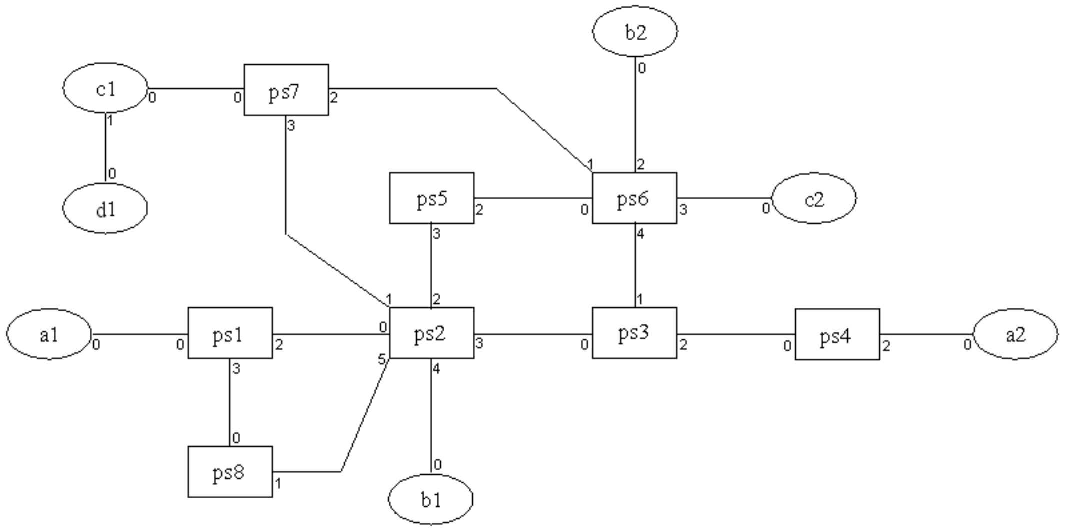


Fig 3.1: Illustration of a test-bed contain L1s

Fig 3.1 shows a test-bed that contains L1s. The rectangles represent L1s where as the ovals can be L2s or end nodes. The node interfaces are numbered for identification. Here, the edge between a1/0 (i.e. interface 0 on node a1) and ps1/0 is a connection, but the edge between c1/1 and d1/0 is both a connection and a datalink. If we were to manually implement a physical path between c1/0 and c2/0, we would need to set L1 internal connections 0-2 on ps7 and 1-3 on ps6. We would then get c1/0 – c2/0 which is a datalink containing three connections. In the above test-bed, let's say that we need to implement the datalinks a1/0 – a2/0. b1/0 – b2/0 and c1/0 – c2/0 as a part of our test topology. We would specify these in our test and the system would appropriately configure the L1s ps7 and ps6 by using the datalinks automation algorithm. The algorithm begins by finding all possible physical paths between the specified pair of interfaces. There may be more than one possible physical path for a given pair. (Observe that between c1/0 – c2/0 there is at least one more possible path through ps7, ps2, ps5 and ps6). The system chooses the shortest path and marks the connections on that path as used. It then moves to the next datalink in the list and follows the same procedure. No two datalinks can share the same connection. Hence if the chosen shortest path for a datalink has a connection that has already been marked as used by another datalink, then the system chooses the second shortest-path. Incase all possible paths for a datalink have a conflict, the system picks up the shortest-path again and sees what datalink is blocking it and goes back to re-route that datalink. Proceeding this way the algorithm converges to end with a single chosen path for each datalink. The algorithm converges even if there is no possible path to implement a datalink, in which case it returns a blank for that datalink. The system should detect this and raise a layout exception. Fig 3.2 highlights the paths chosen by the algorithm in the above-mentioned case. A datalink can be specified between two interfaces only if they are connected to L1 interfaces in the test-bed.

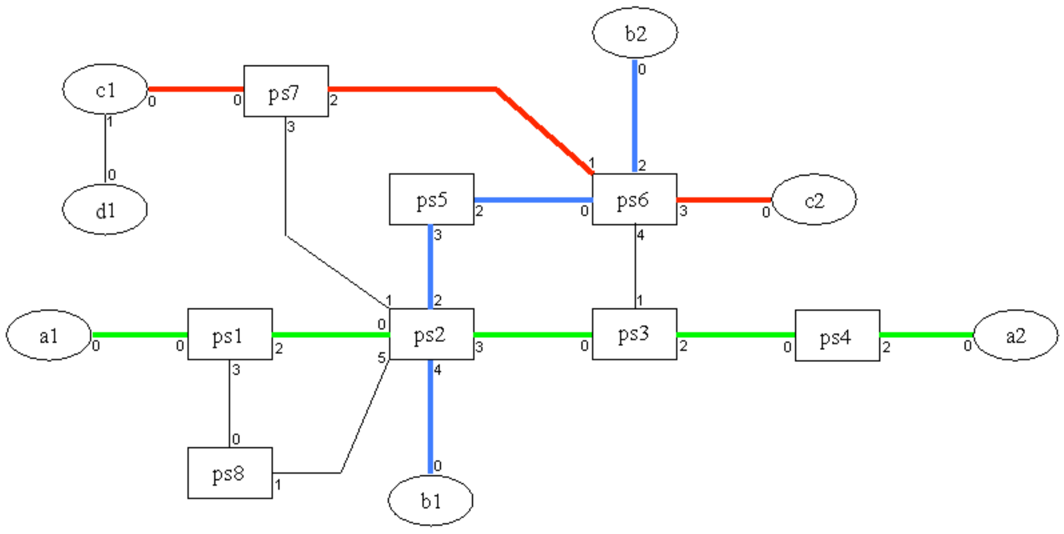


Fig 3.2: Datalinks implemented on a test-bed containing L1s

3.2.2 LAN Automation Algorithm

Datalinks alone cannot be used to specify arbitrary topologies. To specify arbitrary topologies, the user should be able to pickup a group of interfaces and assign them to the same LAN. If a group has only two interfaces, then putting them on the same LAN is similar to (but not the same as) forming a datalink between them. This is done instead of specifying a datalink when there are no L1 switches in the infrastructure. While specifying the LAN for an interface, the L1s in the infrastructure, if any, become transparent, as they have already been used to form datalinks in a prior step. Fig 3.3 shows a test-bed that contains L2s.

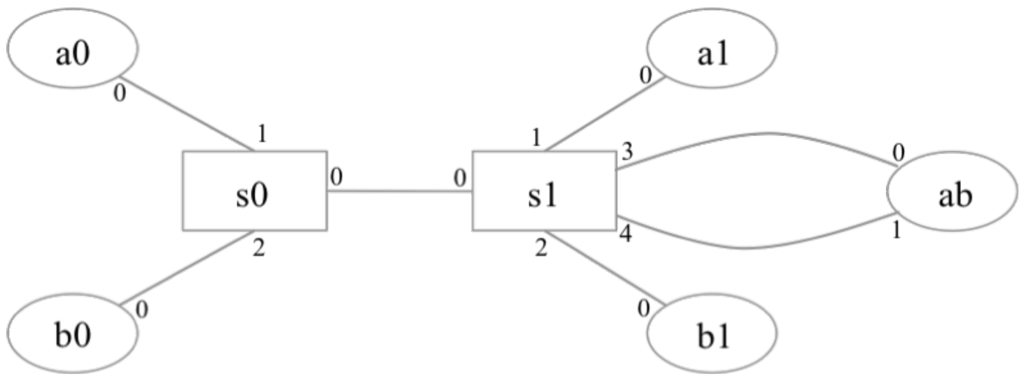


Fig 3.3: Test-bed containing L2s

In Fig 3.3, the rectangles represent L2s while the ovals represent end nodes. The edges are datalinks and may have been implemented in a prior step by configuring L1s, hence L1s are transparent in this diagram. Let's say in our test, we specify to put interfaces a0/0, a1/0 and ab/0 on one LAN and interfaces b0/0, b1/0 and ab1/ on another. This is achieved by configuring interfaces s0/1, s1/1 and s1/3 to be on one VLAN and interfaces s0/2, s1/2 and s1/4 to be on another and by trunking both those VLANs on the datalink s0/0 – s1/0 using the LAN automation algorithm. The algorithm begins by marking each L2 interface that connects to a node interface, with the VLAN that it should be on. If a VLAN is distributed across multiple L2s then the system finds available paths (if needed, passing through other L2s) that connect any two free interfaces on those L2s, selects the shortest path and marks the interfaces on the path with the VLAN. In the end, the interfaces marked with multiple VLANs are configured as trunks. An interface can be specified to be on a particular LAN only if it connects to an L2 switch in the test-bed after implementing the datalinks through L1s.

If Emulab is being used, then the above two algorithms are not used and Netset simple passes the topology information to Emulab using the node provisioning API. Another example of a third-party test-bed service is VCL [VCL]. VCL can return requested nodes but cannot implement the topology if one is specified. This is an example of certain test-beds being restricted in the services they provide, as we discussed earlier.

3.3 Test Automation

The key actions in this phase are environment and traffic setup, data collection and report generation. We observe that all these actions boil down to execution of commands on the nodes under test. We wish to have an extensible system, so we keep Netset at a lower level of semantics. It does not understand what traffic means. It instead understands commands which when executed on a node result in traffic at a higher meaning. For each command, we specify the command line to be executed and the target node on which the command should be executed.

The test can include specification of traffic timings – the start times and durations of traffic flows. To achieve this, there should be a provision to have command launches arbitrarily placed in time. Further there should be a way to group commands together as some may be related to environment setup like configuring loss rates or queue sizes on routers while others may be related to setting up the traffic flows. Typically the environment setup is a first step, which is isolated from the steps that follow to setup the traffic. To enable such grouping of commands, we introduce command sequences that contain commands. To place commands arbitrarily in time we may need to introduce sleep statements between commands. For sleep we need not execute a command on the node. The system itself can cause a sleep duration. To achieve this we define a special command called ‘Sleep’ that can exist within a command sequence but not necessarily have an associated command line to be executed on a target node. Similar, we see that there is a need for other special commands like ‘Kill’ and ‘Wait’ which can kill a command launched previously or cause the system to wait for it to end, respectively. For example, wait can be used between environment setup and traffic setup to wait for all environment setup commands to end before the traffic setup starts. Kill can be used in the final phase to cleanup any running processes.

Another objective is to have a provision to change the parameters passed to these commands, which in turn affect the traffic they generate. This is achieved by ‘parameterizing’ the command line associated with the command. This means variables can be used in the command line. Before launching the command the system will substitute them with real values. The user can provide these values at runtime or the system can provide a way to store sets of such values used for test execution.

We are also interested in collecting the data generated by these commands as that contains our metrics of interest. Netset cannot directly understand the output of every command that may be executed through it, as it could be any arbitrary command put in by the test author. Instead we can have a component in Netset that can read data in a particular form and store it. For every process whose output needs to be monitored for data, we can have a secondary command that launches an ‘output parser’, which is specific to it, which reads its output, parses it and regenerates it in a form that Netset understands. Typically, we identify three kinds of values that a process may generate – current, aggregate and message. A current value is a value produced by the process at regular intervals as it executes. An aggregate value is a value that is generated in the end when the process is terminating. A message is not a data value but some information that should be displayed to the user about the process. We design the Netset’s standard data format keeping these values in mind.

The parameters associated with a command may not be simple values but could also be statistical distribution tables as in the case of some traffic generators. Such parameters can only be specified as files. Further if the user wants to add his own protocol implementation to the test, he does so by including a file in the test. Hence the test should have a provision to associate files with nodes. These files need to be deployed to the nodes at the time of test execution.

So far we have talked about deploying files to nodes and executing commands on them for setting up environment and traffic flows. A node may have one of the many different operating systems available. Further there could be many ways for remotely launching commands or uploading files, only one of which may be implemented in the node. Hence we define something called Communication Modules, which provide a standard API to the system to transfer files and remotely launch, monitor and kill processes. One method of communicating with a node running Linux for example is by using SSH. SFTP could be used to transfer files and the SSH protocol has provisions to launch processes and monitor them by pulling down their standard output and error streams. However, it provides no means for getting the process id of the process just launched, which is used for identifying it when it needs to be terminated by a kill special-command. To overcome this, the command can be executed though the 'sh' shell with the 'echo $$; exec ' prefix which will cause the first line in its output to contain the process id.

When a command is launched on a node, its output is fetched and stored in a log file locally so that it could be passed to the corresponding output parser. The output parser gives back the extracted data to the system, which the system stores along with timestamps. Since a test should have provisions to define report formats, we let the user associate HTML templates with a test, which serve as report formats. These reports may include macros that can refer to values obtained from a test or specify axis and value ranges for a graph to be included in the report

We store the history of every execution of a test along with the associated log files and reports that were generated. This chronological store serves as a way to manage test execution results and reports.

To make the test portable, we can export its information into a structured file, an XML file for example. The XML files along with other files associated with the test can be compressed into a single file, which serves as a portable form of the test. When a test is import into the system from a file, its nodes need to be re-mapped to the test-bed in use, as test-bed information cannot be exported with the test due to its localized nature.

Such an approach provides a basic but robust and extensible method for network test automation. Since it is defined not in terms of network test semantics but in terms of commands and command sequences, it stays very general in nature and offers a good scope of extension.

Chapter 4

System Architecture

This chapter discusses Netset's system architecture and implementation based on the concepts discussed in the previous chapter. In the later part, it also discusses a typical usage scenario.

The current system is implemented using a web-based user interface. The system is programmed mainly in Python and makes use of Django [DJANGO] – A Python based Web Development Framework. It uses MySQL [MYSQL] as the backend Database Management System. It also uses JavaScript, Dynamic HTML techniques, Asynchronous JavaScript and XML (AJAX) and jQuey [jQ] – A JavaScript library, in its interactive user interface.

Fig 4.1 describes the Netset system architecture. Each rectangular block represents a core-programming component of the system. The cylindrical blocks represent data stores. The arrows denote paths along which the data flows. A shaded arrow indicates data flow in a standardized data format or through a standardized data exchanges API. Below, we describe each block in detail.

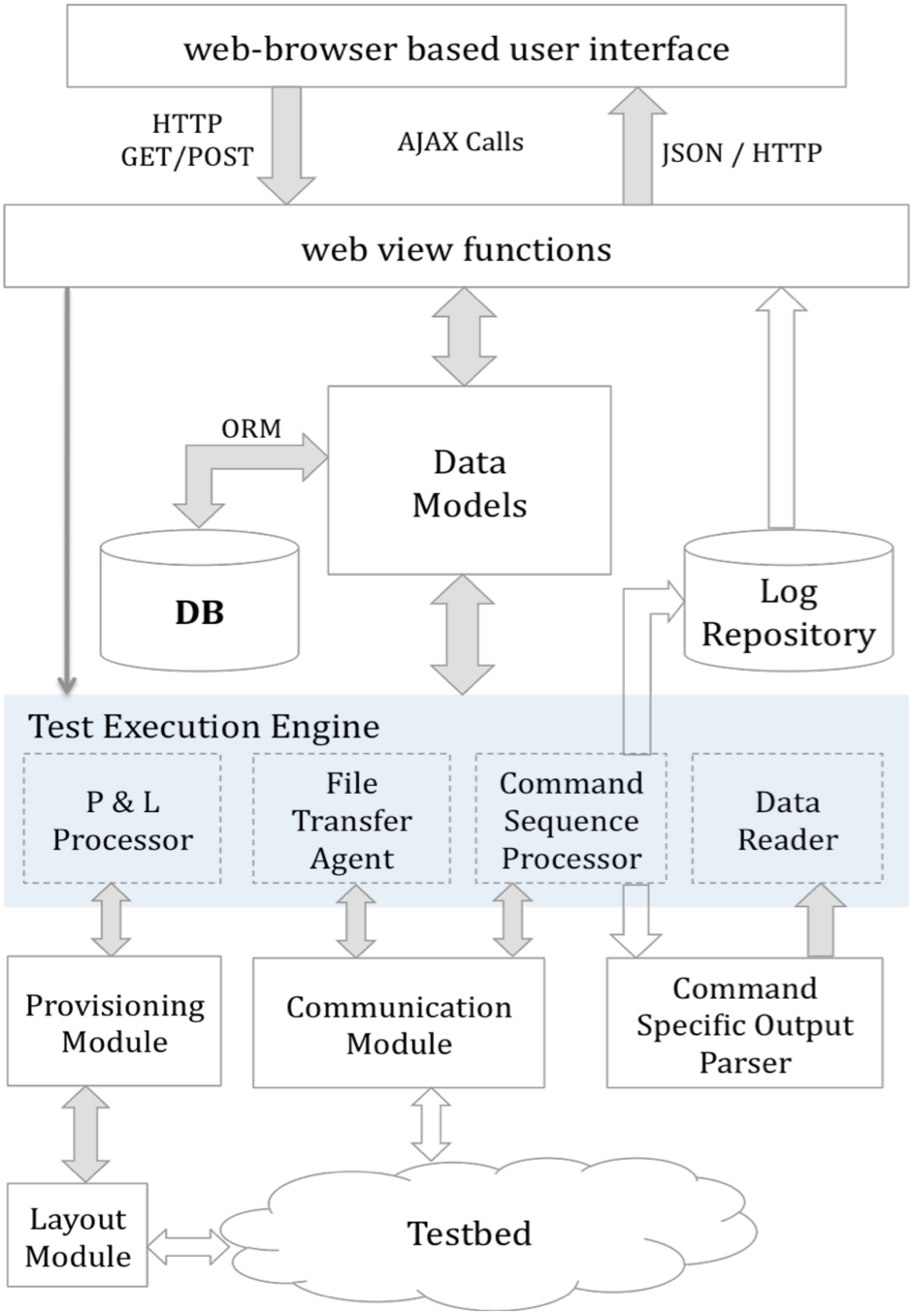


Fig 4.1: Netset's System Architecture

4.1 Data Models

Netset data models implement the data structures and database access for the various entities that it defines. Each model is a Python class that is a descendent of Django's Model class that provides Object Relational Mapping (ORM) services. The ORM-enabled Model class is a part of the Django package and provides an abstraction layer between the client application and the database, enabling the use of different databases without modification in the client application. As of Django version 1.0, Django's Model classes supports PostgreSQL, MySQL, Oracle and SQLite database management systems. Other blocks of the system communicate information using these models.

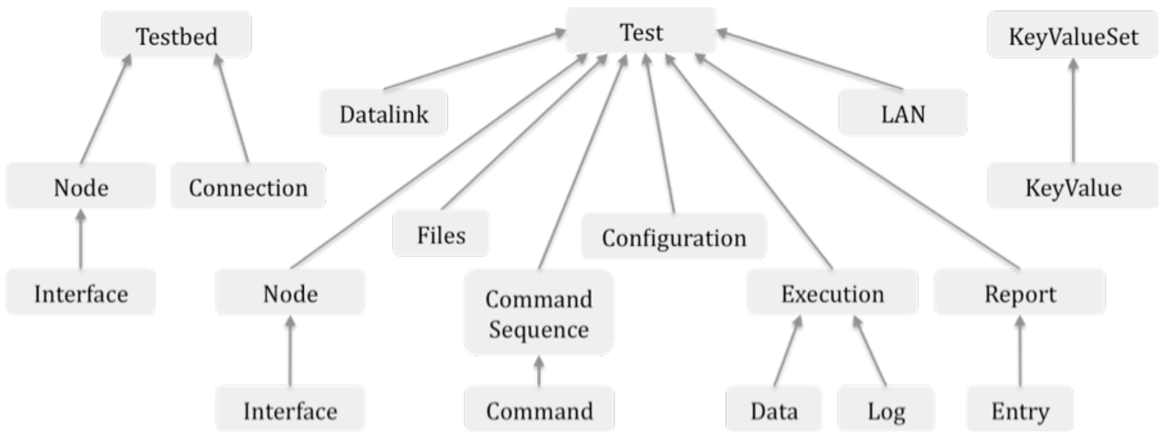


Fig 4.2: Netset's Data Models

Fig 4.2 shows the hierarchy of Netset's data models. The *Test* and *Testbed* are the two main top-level models. The *KeyValueSet* model is an accessory. It is a collection of objects of type *KeyValue* model that represents a key-value pair – a name associated with some data. It is used to associate property-value and parameter-value pairs with other models.

The Test model is used to represent a test scenario specified by the user. Within a test, there are *nodes*, *datalinks* and *LANs* that represent the topology information. Each node can have one or more network *interfaces*. A datalink specifies a pair of interfaces between which a physical path should be established at the time of test execution. The system achieves this in the layout phase of test execution by appropriately configuring the Layer-1 switches in the test-bed. Each LAN contains a list of interfaces that should be on the same Layer-2 network at the time of test-execution. The system achieves this in the layout phase of test execution by appropriately configuring Virtual LANs on the Layer-2 switches in the test-bed. By configuring the Layer-1 and Layer-2 switches in this manner, the system accomplishes the topology requirement of the test.

In addition to topology information, a test also contains *files* and *command sequences*. Each file entry specifies a file to be uploaded to the node and the directory path on the node that is should be uploaded to. Each command sequence is a list of *commands*. The commands setup the network environment – flows, kernel parameters, routing parameters etc. – for the test. They also gather data from the processes under test. Each command entry can be of *Execute*, *Sequence*, *Sleep*, *Wait* or *Kill* type. Execute type is used to specify a *command line* to be executed on a particular node. Sequence type is used to specify another command sequence that should be executed in parallel from that point on. Sleep is used to cause the command sequence to wait for a specified duration. Kill is used to terminate another command of Execute type, in the same or a different command sequence. Wait is used to cause the command sequence to wait for another command of Execute type that may be in the same or a different command sequence, to terminate. The command line of an Execute type command can be parameterized – parts of it can be replaced with variables having some default values that can also be overridden to ‘configure’ the test differently. A *test configuration* comes into play for this purpose. A test configuration has a name and an associated KeyValueSet. The keys in this KeyValueSet correspond to command parameter names in the test and their associated values are used to override the default parameter values. The user specifies a test configuration at the launch of the test. Incase no test configuration is specified, the test is executed with the default parameter values. A test configuration's KeyValueSet can also hold a key that can be marked to disable a particular command in that configuration.

Every test has an associated *report*, consisting of *report entries*. The report specifies a document template used to generate a test execution report. Each report entry specifies a piece of text, a table of values or a graph to be included in that report. Note that a report only specifies a document format and entries to be included in it and is not an instance of an actual report generated from a test execution. A test when executed produces an *execution* that stores its start and stop timestamps with the overall outcome – success or failure. Each execution contains an *execution log* and *execution data*. The log stores timestamps and information about state transitions that the test and each of its command sequences and commands went through during the execution. The execution data contains the set of values generated during the test execution. This is used to build the test reports.

The Testbed model is used to represent local test-beds on which a test can be run. It consists of *nodes* and *connections*. A test-bed node corresponds to a physical node. It has one or more network interfaces. Note that this is different from a test node, which is just a part of the definition of the test. A connection in the test-bed represents an existing real physical path between two test-bed node interfaces. To execute a test, each of its nodes should be mapped to a real test-bed node and the system uses the test-bed's connection information to implement the test's required topology. This makes the test entity portable and disconnected from any real test-bed. It may be exported from one Netset installation and imported into another that uses a different test-bed. As long as a test can be fit onto a test-bed i.e. the test-bed infrastructure is able to implement the required test topology and execute it. A test can be exported to or imported from an XML file that has the same tags as the test model names and in the same hierarchy.

4.2 Test Execution Engine

The test execution engine is one of the main components of the Netset system. It's a multi-threaded program written in Python. It is launched with the id of the test to be executed, as a parameter. It can also take a test configuration id as an optional parameter. It can be launched from the command line, or as more commonly done, through the system's web interface. Upon launch, it carries out the different phases of test execution. These include the node provisioning and layout phase, the file transfer phase, the configuration phase, the test phase and the cleanup phase.

The node provisioning and layout phase is carried out by the Provisioning and Layout (P&L) Processor component of the test execution engine. Provisioning means acquiring a test-bed node for each test node and layout means arranging the acquired nodes in the topology specified by the test. A *Provisioning Module* is used for this purpose. The provisioning module is test-bed specific.

Upon successful node provisioning and layout, the file transfer phase is carried out by the *File Transfer Agent* component of the test execution engine. In this phase the files required by the commands are transferred to the nodes. These may include source code files, binaries, data tables, kernel modules etc.

After the required files have been transferred, *the Command Sequence Processor* component of the test execution engine carries out the configuration, test and cleanup phases by executing the *config*, *main* and *cleanup* command sequences respectively. These three phases of execution are similar in the sense that they all involve execution of a command sequence. The purpose of the config command sequence is to execute initialization commands for the test like installing kernel modules, configuring router delay and loss characteristics etc. The purpose of the main command sequence is to launch the core test processes and that of cleanup command sequence it to execute clean-up commands like unloading kernel modules, deleting temporary files etc. Command sequences can contain five different types of command entries – Execute, Sequence, Kill, Wait and Sleep. There can be a set of options associated with every command entry. For an Execute type command entry the command sequence processor processes the command line to replace the parameters with actual values from the configuration in use. It then executes it on the associated node. A *tracker thread* is then started to track its termination. If the option to 'monitor' the command is set, the tracker thread also reads its raw output and dumps it to the *Log Repository*. Further if an output parser is specified for the command, it is launched and the process's output stream is connected to it. The output parser extracts data from the process's output stream and generates a new output in a standardized format recognized by Netset. To read this output, a *Data Reader* thread is started for each output parser process. An output parser is specific to the command launched. Alternately, an option can also be specified to directly connect the data reader to the original process if it directly outputs in Netset's format. The data reader interprets the output in Netset's format and puts the values read into the database using the execution data model. If neither an output parser nor the option to directly read the original process's output for data is specified, then a data reader thread is not started and only raw logs are available, provided that at least the 'monitor' option was set. When a launched process terminates, the tracker thread reads its return value to check if it completed successfully and reports this to the database from where a user can read it.

Sequence type commands cause the command sequence processor to launch the specified sequence in a separate thread that runs in parallel with the launching sequence's thread. A Wait type command causes the command sequence to wait for the specified Execute type command or command sequence to terminate. This is achieved by waiting for the command's tracker thread or the command sequence's thread to end. When an Execute type command is launched the command sequence processor stores it process id. When a Kill type command is issued, the command sequence processor uses this information to terminate the target command. A sleep type command simply causes the command sequence to wait for the specified duration.

4.3 Provisioning Modules

A provisioning module is used to acquire test-bed nodes for test nodes and to build the topology specified by the test. It provides a standardized programming interface to the test execution engine. Netset provides a built-in provisioning module which can be used with local test-beds that contain Layer-1 and Layer-2 switches. The provisioning first maps each test node to a test bed node. The mapping information is read from one of the properties of the test node (stored in an associated KeyValueSet). Thus, if the built-in module is being used, the user can (should) choose which test-bed node should be mapped to each test node by setting its properties. The provisioning process simply associates three pieces of information with the test node – the control address, the control username and the control password. After node provisioning, the provisioning module configures the switches to achieve the required topology.

Provisioning modules are not built into the test execution so that additional provisioning modules can be written to interface Netset with third party test-bed services like Emulab. Netset also provides an additional provisioning module to provision nodes from North Carolina State University's VCL service. VCL however does not support an arbitrary topology implementation, so the node provisioning is limited to just acquiring the nodes.

4.4 Layout Modules

When Netset's built-in provisioning module is used, the process of implementing the required topology for a test involves configuration of Layer-1 and Layer-2 switches in the local test-bed. There are many different vendors for these devices and diverse models are available with different control APIs. Hence the provisioning module cannot directly talk to these devices. Instead it defines a set of standard interface functions that are implemented by the *Layout Modules*. Examples of these functions are those used to create datalinks or Virtual LANS or attached a Virtual LAN to an interface etc. Layout modules are like drivers in an operating system. On one side a layout module provides a standard set of functions to the test execution engine and on the other side it implements the device-specific API to configure it. A layout module is thus device specific. Netset defines the standard set of functions that should be implemented by layout modules. Currently the system supports only those layout modules that are written in Python. Future versions can include support for other programming languages by including Python wrapper modules for them.

4.5 Communication Modules

The test execution engine needs to transfer files to the nodes under test and launch processes on them. It also needs to monitor these launched processes and pull down their output and error streams for logging. Different nodes may be running different operating systems and providing different services. Some may provide shell access through SSH while others may expose a web-service API for access. The test execution cannot directly deal with such diverse access possibilities. It instead defines a standard set of *communication* functions that are implemented by *communication modules*. Examples of these functions are those used to send files, launch a remote process, kill a remote process etc. Netset provides a built-in communication module called the *SSH_sh* communication module. This module works with nodes that support Secure Shell Access (SSH) and have the *sh* shell installed. This is true for most of the nodes that run a POSIX compliant operating system. Windows based nodes can also be used with this module by installing Cygwin [Cygwin].

4.6 Output Parsers

One of the important services offered by Netset's test execution engine is automatic collection and storage of data generated by the test processes. This data is extracted from the output generated by the test processes, associated with a timestamp and stored in the database for future reference. The test execution engine allows launching of arbitrary commands through its command sequences and it cannot understand the output of every possible command. Thus command specific *output parsers* are written for each command. An output parser for a particular command parses its output to extract values of interest and outputs them in a standard form that the test execution engine can understand. The test process's output stream is connected to the output parsers input stream and the output parser's output stream is connected to back to the data reader in the test execution engine. Since the output parser only has to work with an input stream and an output stream it may be written in any programming language as long as it can run on the Netset host machine. Netset provides output parsers for commands like *iperf* that are more commonly used in network testing.

4.7 Web View Functions

The web view functions provide a web interface to Netset. They are called 'view' functions, as they are indeed what are called the *view functions* in the Django Framework's Model-Template-View (MTV) pattern. These view functions are accessed through web URLs defined as per the Django's URL settings. They provide the backend for Create-Read-Update-Delete (CRUD) operations on the Netset models. They also provide special interfaces like the *Execution Console* to start and view a test execution or to generate reports from data generated by a test execution. In classic web interfaces, the HTML pages are generated on the server and sent to the client when a page is requested. The current system however uses a more recent trend in which a DHTML based document is sent to the client ones and all further transactions take place through AJAX calls, much like in Google's Gmail service. Hence, the web view functions in our case only generate JavaScript Object Notation (JSON) [JSON] responses and not full HTML documents. The JSON object returned to the client has a property called *retval*, which should hold a zero if the operation was successful. If its non-zero, it indicates an error in which case the object has another property called *errorMessage* which contains a descriptive message for the situation. In addition to retval the returned object has another property that is an object containing the information requested, generally referred to as the *answer*. The specific name of this object is however operation specific. Fig 4.3 depicts this transaction between the DHTML User Interface and a web view function.

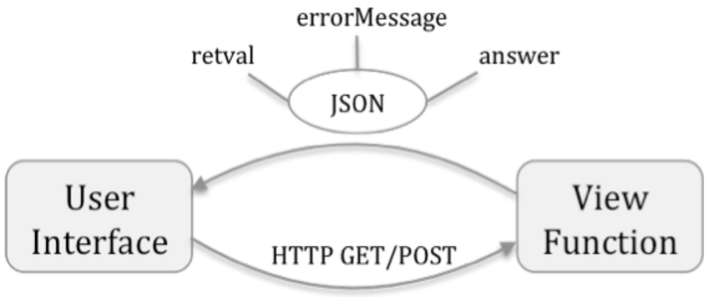


Fig 4.3: AJAX call to a web view function

4.8 User Interface

This refers to the HTML file that is sent to the client (browser) when web access to Netset is first requested. This file contains the entire user interface programmed in JavaScript. It uses the jQuery JavaScript library for generating AJAX calls, to achieve cross-browser compatibility and to reduce lines of code. The user interface communicates with the Netset backend using AJAX calls to its web view functions that reply in JSON.

4.9 A Typical Usage Scenario

In a typical usage scenario, the user logs on to the system using a username and a password, creates a new test or opens an existing one, creates or chooses an existing test configuration, maps the test to a test-bed, executes the test and views the generated reports. Table 4.1 enlists in detail the steps taken by a user to create a test from scratch and executing it.

Table 4.1: Steps in a typical usage scenario

Step	Description
1	Log on to the system
2	Create a new test by specifying a unique name and an optional description. Open the newly created test
3	Add nodes by specifying a unique name and an optional description for each. For each node, add network interfaces by specifying a unique name and an optional description for each.
4	Specify the test topology by specifying datalinks between interface pairs and grouping interfaces in LANs.
5	Add files to be deployed on each node.
6	Create the required *main* command sequence and the optional *config* and *cleanup* command sequences. Create custom command sequences if needed.
7	Add commands to each command sequence. While adding an Execute type command, specify a parameterized command line and associate default parameter values with the command.

Table 4.1 Continued

Step	**Description**
8	Create zero or more report templates for the test.
9	Create zero of more test configurations that specify different values for command parameters or enable or disable a command.
10	Associate test-bed mapping criteria to the test nodes by setting their properties. (Examples: In case of local test-beds, Setting the *tb_node* property does this. In case of VCL, the *vcl_image* property is set to the id of VCL image to be used on the provisioned node.)
11	Execute the test
12	View Reports

Typically, the same user may not carry out all the steps specified in Table 4.1 for test creation. These steps may be divided between multiple users. A user may select an existing test stored in the system and directly begin at step 9 or in some cases at step 10. The user can also import a test provided by another user in the form of a portable XML file and again directly begin at step 9 or 10. A user may also log on to the system just to view and compare reports of past test executions, in which case only step 12 is performed.

In case the test nodes are configured to use a local test-bed, then before step 10 can be performed, it must be ensured that information about the local test-bed exists in the system. This is done by performing steps similar to 2, 3 and 4 but for creating test-beds instead of tests. In this case step 4 will differ in the sense that *connections* are created instead of datalinks and LANs.

Ideally, the above-mentioned steps should be divided amongst four kinds of users – test publishers, researchers, reviewers and system administrators. A test publisher would create the test by specifying the necessary topology and environment in the form of command sequences. A researcher would execute the test with his/her protocol software and a reviewer would review and compare reports for the test scenario performed by different researchers using their respective protocol implementations. A system administrator's role would be to maintain the test-bed and its information in the system installation, and communicate it to the system users.

Chapter 5

Case Studies

In this chapter, we look at two real-world test scenarios – one used for testing a TCP congestion control protocol and another used for testing the driver software for a network interface card. We evaluate the current testing approach of each, identify its limitations and describe how they can benefit from Netset.

5.1 Dumbbell Topology and Dummynet Routers for testing CUBIC

In this section we review the test process carried out by Sangtae Ha, here after referred to as the researcher, who is the maintainer of CUBIC TCP and a PhD candidate at the North Carolina State University. CUBIC TCP is the default TCP implementation in current Linux Distributions, as of November 2008. The researcher's job is to monitor feedback about CUBIC from the networking research community, make changes to the CUBIC implementation and test it for performance. The performance test of CUBIC requires a network test-bed. Fig 5.1 shows the researcher's test setup.

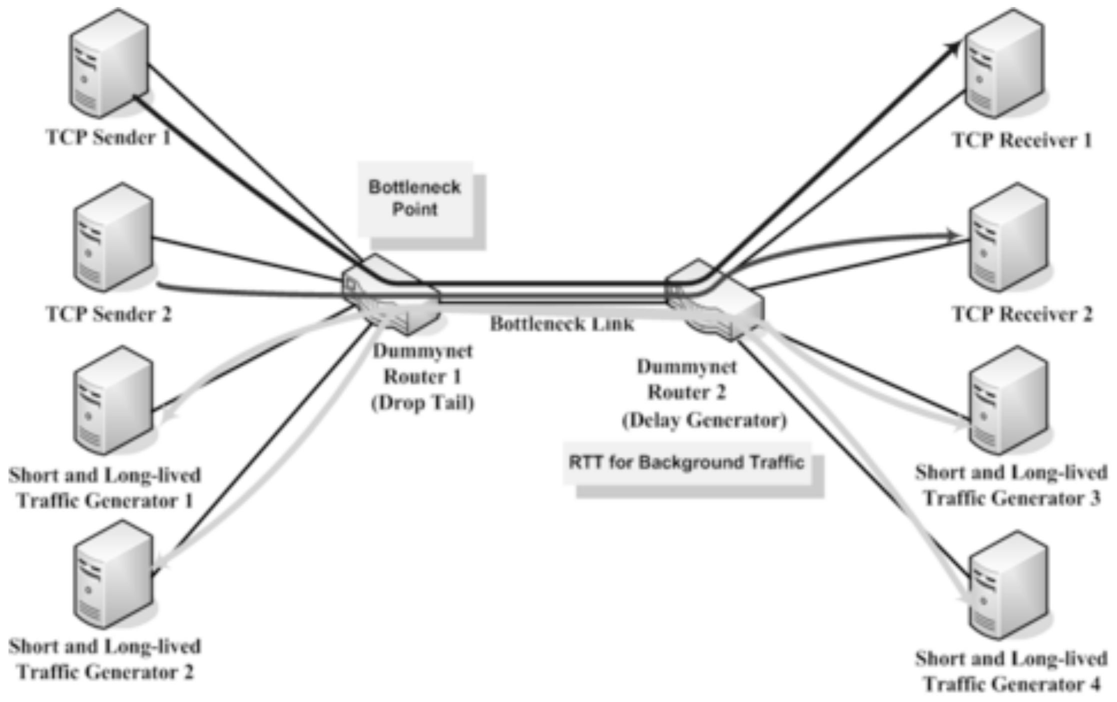

TCP Sender 1
Bottleneck Point
TCP Sender 2
Short and Long-lived Traffic Generator 1
Short and Long-lived Traffic Generator 2
Dummynet Router 1 (Drop Tail)
Bottleneck Link
Dummynet Router 2 (Delay Generator)
RTT for Background Traffic
TCP Receiver 1
TCP Receiver 2
Short and Long-lived Traffic Generator 3
Short and Long-lived Traffic Generator 4

Fig 5.1: Test setup for performance testing of CUBIC
(*Courtsey: [NRL]*)

The test-bed consists of ten nodes arranged in a dumbbell topology as shown in Fig 5.1. Each black line between any two nodes is a single link and each such link is a separate LAN. The two nodes in the middle act as routers. Their buffer size and delay characteristics are set using Dummynet. Also, the link between them is used to emulate a bottleneck link by bandwidth limiting the interface it connects to on the left router (Dummynet Router 1). The right router (Dummynet Router 2) is configured to introduce delay in the packets it forwards. Both routers use the drop tail policy for buffer overflows. On the left there are two nodes where test flows originate (TCP Sender 1 and TCP Sender 2). These flows respectively terminate at two nodes on the right (TCP Receiver 1 and TCP Receiver 2). There are four more nodes, two on each side, that generate background traffic. The network topology causes all test and background traffic flows to pass through the two routers and the bottleneck point. This test has been described in [HaTrf06]. The objective of this test is to obtain flow throughputs, RTT-ratios, TCP-friendliness, link utilization, congestion window characteristics, router queue characteristics, CPU utilization at the nodes etc. under various types of traffic-mixes and generate reports showing the results. The traffic mix is varied by varying the distribution parameters given to the traffic generator program that runs on the traffic generator nodes.

The researcher uses a scripted automation approach to conduct this test. The scripts are written in *tcsh* and are stored and executed from a separate control machine not shown in the diagram above. The script that executes the entire test is called *plan_one*. *plan_one* internally calls other scripts specifically written for this test, which in turn may call other script. The system consists of around 30 such *tcsh* scripts, all of which are specific to the test. Examples of these scripts are initialization scripts, clean up scripts, traffic generator launcher scripts, utility scripts etc. These scripts use positional command line parameters. The researcher used his ingenuity to decide the scripts' names and the parameter positions. To understand the parameter positions, one has to look into the script's code, as not every script is supposed to be called directly by the user and hence not documented. Some scripts show signs of assumptions that are specific to this test. This makes it difficult and discouraging for another researcher to review or modify the test setup. This is also evident from our experience as a third-party while trying to evaluate the setup, which required a lot of interaction with the researcher. Below we discuss the steps that *plan_one* takes while running, point out their drawbacks and describe how using Netset eliminates those drawbacks.

1. The script begins by reading the positional command line parameters into variables. One has to see this section of the code to know what parameters are to be passed and in what order. Since the script is like a personal tool, it is not properly documented to begin with. *plan_one* takes up to 22 such parameters, making the command line very complex. Another researcher who writes a similar script for his test may use slightly different parameters in a different order,

resulting in similar work being repeated and moreover in an incompatible manner. Using named arguments of type "--key=value" can help, however *tcsh* doesn't include a library to parse that and it is not fair to expect the researcher to write that code when the main objective of the script is something else. Using Netset, parameters associated with a command can be named causing a 'self-documenting' effect, which causes them to show in a test-configuration where the user can provide their values.

2. Next, the script reads the 'test-bed' configuration by including another script that has the configuration information (IP addresses, port numbers etc.) stored. Here, the test-bed information is directly tied in to the test making it non-portable. Also, the number of nodes is hard-coded into the script requiring a modification of the entire script if the topology or number of nodes were to change. When using Netset, the test-bed is defined separately and the nodes included in a test are mapped to the test-bed at the time of test execution. This makes it convenient to add nodes or change the topology. It also makes the test portable. Netset also includes automatic topology implementation by configuration of switches, which is done manually in the test we are looking at.

3. This test supports 5 traffic mixes – None and Type1 through Type5, which are hard-coded into *plan_one*. One of the parameters to *plan_one* specifies the traffic mix to use and the corresponding traffic parameters, which are hard-coded, are then loaded for use. Adding a new traffic mix thus requires editing the script. Using Netset this could be done by specifying traffic parameters in test configurations, which can be chosen at the time of execution. Adding a traffic mix to the test would then just mean creating a new configuration.

4. Next, the script copies required files onto the nodes. These files are stored in folders that have the same name as the nodes they are to be deployed onto. An

intelligent approach, but not necessarily the only one as another researcher doing similar work may come up with a different approach. Also the path these files are to be deployed to is hard coded in *plan_one*. *plan_one* deploys these files by using *scp*, which has the same problems as *ssh*, as described in the next point. With Netset, files to be deployed to a node are simply associated with it under the context of a test along with a target path. The test automation engine takes care of the rest.

5. The script then initializes the nodes, which includes configuring the Dummynet routers, setting interface bit rate for the bottleneck point, starting traffic servers etc. This requires executing commands on the nodes. The script has hard coded *ssh* command calls for this. If the target node used a method other than *ssh* for access to it, this part of the script would need to be changed. Further, calling *ssh* requires interaction with the terminal and not with ssh's input/ouput streams alone as it throws a password prompt directly to the terminal. This would require an *expect* script to be written. The *tcsh* script would call the *expect* script which would then call *ssh*. To avoid this complicated approach the researcher has set 'public keys' on the node machines manually so that *ssh* commands connect without throwing a password prompt. This is another point that makes the test non-portable. Netset makes this transparent in communication to the nodes as it defines a separate communication module which could be based on *ssh* or something else. The researcher then simply needs to choose the command and the node to run it on. If Netset is not used, then the researcher also needs to redirect the command's output to log files and track and download them. Netset makes this transparent and automatically creates and monitors log files for commands that are marked for monitoring.

6. The next step taken by the script is to launch the test and traffic flows. This requires launching commands for which the script has similar drawbacks as stated

in the previous point. Also, the script uses a series of sleep statements to distribute the flows in time. Using Netset, this can be done by appropriately arranging the commands in command sequences.

7. The running commands generate logs, which contain the data required to generate reports. *plan_one* extracts this data but does not store it to disk. It directly processes it to generate reports. The reports are HTML documents and their markup (presentation) is hard-coded into *plan_one*. A change in report format, either presentational or one to include an extra graph or value thus requires a change in the script. While using Netset, reports are defined as HTML templates with placeholders for values and graphs. Netset permanently stores the data extracted from the logs along with the necessary timestamps. It can thus even generate new reports, which were not defined at the time of execution of the test.

8. At any point during the execution of *plan_one* if one of the command fails, the test execution can no longer be considered valid and all other commands need to terminated before the test can be re-run. *plan_one* does not automate this and it needs to be taken care of manually. Netset on the other hand tracks every command launched and upon failure of one can go back and terminate all others to return the nodes to a fresh state.

In conclusion, this test for CUBIC TCP has a strong personal element of the researcher that affects it portability. By making the test script more general and adding the missing features the problems described can be taken care off. However, it is then seen that the testing process has a pattern and it would be redundant and inefficient for every researcher to write detailed scripts. Instead a general script shared by all would be more advantageous. But such a general script is exactly what Netset provides along with a graphical user interface, which makes test definitions even simpler. In development of something like CUBIC TCP where a research community is involved, using a framework like Netset would definitely ease communication of test scenarios and reports amongst researchers. The results for the CUBIC TCP test can be seen on the NRL [NRL] website.

5.2 Chelsio Linux 10Gbps Ethernet Performance Test

While Netset can serve as a useful tool for computer networking researchers, it can also be used for benchmarking network devices or associated software. Here we look at a case where performance of a network interface card (Chelsio S310E-SR 10Gbps NIC) is benchmarked with two different driver software – one provided by Redhat and one by Chelsio. This test is designed by John Bass, here after referred to as the user, who is the technical director of VCL at North Carolina State University. Fig 5.2 shows the test setup used.

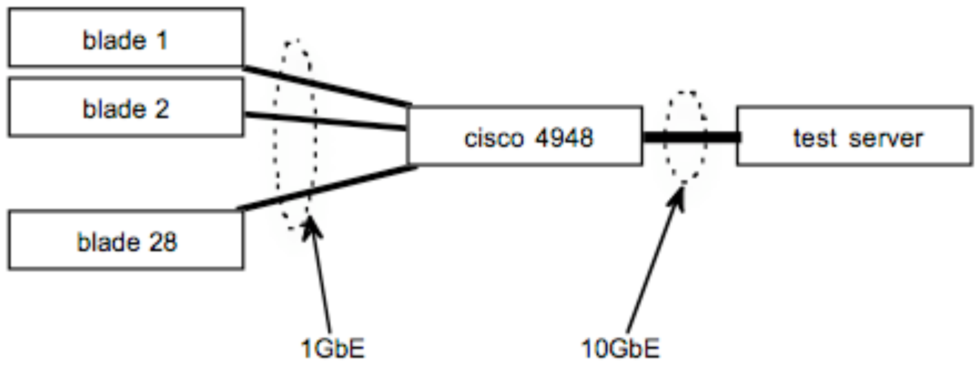


Fig 5.2: Test setup for Chelsio S310E-SR 10Gbps NIC
(Courtesy: Chelsio Linux 10Gbps Ethernet Performance, John Bas)

The test setup involves 30 nodes – Blades 1 through 28 and a test server, all connected to a Cisco switch. Each blade is connected through a 1Gb Ethernet link while the test server that has the Chelsio NIC installed is connected through the same NIC to the switch using a 10Gb Ethernet link. The objective is to start TCP flows from each blade to the server. Since there are 28 blades, each connected at 1Gbps, the 28 flows overwhelm the 10Gb link to the server, hence driving it to its maximum capacity. Iperf [IPerf] is used to generate the TCP flows. These TCP flows are balanced i.e. have equal transfer in both directions. Iperf server is run on the server, whereas iperf client is run on each blade for about 700 seconds. While these flows run, statistics like transmission bandwidth, CPU utilization and memory usage are gathered every five seconds from the server. The test is run twice – once with a software driver from Redhat and another time with a software driver from Chelsio. This is relatively a much simpler test as Compared to CUBIC TCP. A shell script is used to first launch the iperf flows. A logger script is run on the server with its output redirected to a file, which is later processed manually to extract data of interest and plot the results.

Here, using Netset as an automation tool can eliminate writing the scripts. The commands can simply be defined in the main command sequence using the user interface. Netset can then automate management of the logs and data extraction. A report format can be defined for the test, into which Netset can automatically substitute values to generate reports.

Chapter 6

Conclusion and Future work

In this thesis, we looked at and documented the nature and scope of network based tests, which are primarily integral to testing network protocols. We identified the problems that arise due to unavailability of a standard tool for creating, sharing and executing network based tests and managing and evaluating their results. As a solution to the problem, we successfully designed and developed an extensible tool called Netset that automates all aspects of network based tests – from topology definition to report generation.

Netset provides a robust base system to automate network testing. We believe a lot of useful development can be undertaken to extend Netset in terms of its user interface, features, accessory plug-ins, support for more test-beds, integration of more discrete testing tools etc. Below we discuss a few developments that can be undertaken in near future.

Action Library: A list of pre-stored commands can be added to the Netset user-interface from which a user can choose while adding commands to a command sequence. So if the user wants to add for example the *iperf* command in a command sequence, instead of typing it by hand, he can choose it from this list called the *action library*, which will auto populate the command sequence with the required iperf command line. A user can then better define the test in network terms instead of commands.

Declarative Configuration: A feature can be added to Netset to allow the user to associate key-value pairs with a test, node or interface that will result in a command being executed at the time of test. For example user can 'declare' the bit rate for an interface using a key-value pair instead of specifying a command to do so.

Real time commands and statistics: It can be interesting and useful to intervene in a running test. This means launching commands while a test is already in progress or triggering a process to return some value, which is not otherwise being monitored. Netset currently doesn't support this but recommends this feature as a future development. Integration with an SNMP client sounds like a promising approach to achieve this.

Comprehensive Test Configurations: Currently test configuration can override command parameter values and enable or disable commands. Their use can be made better if they can override everything about a test like modifying a test topology or adding new commands and command sequences, just like a class–derives class relationship in object oriented programming.

Promotion in the community: The true usefulness of Netset will be seen only after it has been adapted by a wide user-base. Steps should be taken to promote Netset as a conventional community tool.

Integration of more discrete test tools, communication and layout modules: Netset currently doesn't support all the discrete tools available for testing, which include traffic generators, network delay simulators etc. It also supports a limited number of communication and layout modules. Integrating more test tools and modules into Netset will serve as a good means of extension.

Integration with NS-2: It will be useful to be able to export Netset files to NS-2 or import test scenarios from NS-2 into Netset.

REFERENCES

[AKM04]	G. Appenzeller, I. Keslassy, and N. Mckeown. "Sizing router buffers." In the *Proceedings of ACM SIGCOMM*, August 2004.
[AKSJ03]	J. Aikat, J. Kaur, F. D. Smith, and K. Jeffay. "Variability in TCP round-trip times." In the *Proceedings of Internet Measurement Conference*, 2003.
[BIC]	L. Xu, K. Harfoush, and I. Rhee. "Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks." In the *Proceedings of IEEE INFOCOM*, March 2004.
[Cygwin]	Cygwin Project: http://www.cygwin.com/
[DITG]	S. Avallone, A. Pescape and G. Ventre. "Distributed Internet Traffic Generator (D-ITG): Analysis and Experimentation over Heterogeneous Network." In *ICNP 2003 poster Proceedings, International Conference on Network Protocols* 2003 – Atlanta, Georgia (USA), November 2003.
[DJANGO]	Django Project: http://www.djangoproject.com/
[DPMC04]	C. Dovrolis, R. Prasad, M. Murray, and K. Claffy. "Bandwidth Estimation: Metrics, Measurement Techniques, and Tools." *IEEE Networks*, Vol. 17 No. 6, Nov-Dec 2003, pp 27-35, April 2004.
[EMULAB]	Emulab Network Emulation Testbed: http://www.emulab.net/
[FGHW99]	A. Feldmann, A. C. Gilbert, P. Huang, and W. Willinger. "Dynamics of IP Traffic: A Study of the Role of Variability and the Impact of Control." In *the Proceedings of SIGCOMM'99*, Boston, MA, Sep 1999.
[FK02]	S. Floyd, and E. Kohler. "Internet Research Needs Better Models." *Hotnets-I*, October 2002.
[FK02a]	S. Floyd, and E. Kohler. "NSF Project ANI/STI: Measurements, Models, and Simulation Scenarios for Internet Research."
[Flo06]	Metrics for the Evaluation of Congestion Control Mechanisms. draft-irtf tmrg-metrics-04.txt, August 2006.

[FP03] S. Floyd and V. Paxson. "Difficulties in Simulating the Internet." *IEEE/ACM Transactions on Networking*, Vol.9, No.4, pp. 392-403, August 2001.

[GenSyn] P. E. Heegaard. "GenSyn - a Java Based Generator of Synthetic Internet Traffic Linking User Behaviour Models to Real Network Protocols." Presentation at *ITC Specialist Seminar on IP Traffic Measurement, Modeling and Management*, Sep 18-20, 2000, Monterey, CA (USA).

[HaTrf06] S. Ha, L. Le, I. Rhee, L. Xu. "Impact of background traffic on performance of high-speed TCP." *Computer Networks: The International Journal of Computer and Telecommunications Networking, Volume 15, Issue 4, Aug. 2007*

[HJS04] F. Hernandez-Campos, K. Jeffay, and F. D. Smith. "How Real Can Synthetic Network Traffic Be?" Submitted for publication, February 2004.

[HTCP] D. Leith and R. Shorten. "HTCP protocol for high-speed long distance networks." *International Workshop on Protocols for Fast Long-Distance Networks (PFLDNet)*, February 2004.

[IPerf] Iperf Project: http://dast.nlanr.net/Projects/Iperf/

[JFI] Jain, R., Chiu, D.M., and Hawe, W. "A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Systems." DEC Research Report TR-301, 1984

[jQ] jQuery Project: http://jquery.com/

[JSON] JavaScript Object Notation: http://www.json.org/

[LAJS03] L. Le, J. Aikat, K. Jeffay, and F. D. Smith. "The Effects of Active Queue Management on Web Performance." In *the Proceeding of ACM SIGCOMM* 2003, pp. 265-276, Karlsruhe, Germany, August 2003.

[LLS06] Y. Li, D. Leith, and R. N. Shorten. "Experimental evaluation of TCP protocols for high-speed networks. To appear in *IEEE/ACM transactions on Networking*, 2006.

[MGEN] Mgen Project: http://mgen.pf.itd.nrl.navy.mil/

[MYSQL] MySql Project: http://www.mysql.com/

[NetPerf] Netperf Project: http://www.netperf.org/netperf/NetperfPage.html

[NetSpec] NetSpec project: http://www.ittc.ku.edu/netspec/

[NRL] Networking Research Lab at North Carolina State University. http://netsrv.csc.ncsu.edu/

[NS3] NS-3 Project: http://www.nsnam.org/

[PLANET] PlanetLab Project: http://www.planet-lab.org/

[PsnTraf] Poisson Traffic Generator: http://www.spin.rice.edu/Software/poisson_gen/

[Ri97] L. Rizzo. “Dummynet: A simple Approach to the Evaluation of Network Protocols.” *ACM Computer Communication Review*, January 1997.

[Rude] Rude Project: http://rude.sourceforge.net/

[SHJO01] F. D. Smith, F. Hernandez Campos, K. Jeffay, and D. Ott. “What TCP/IP Protocol Headers Can Tell Us About the Web.” In *the Proceeding of ACM SIGMETRICS* 2001/*Performance* 2001, pages 245-256, Cambridge, MA, June 2001.

[SSTraf] Self-Similar Traffic Generator: http://wwwcsif.cs.ucdavis.edu/~kramer/code/trf_gen1.html

[Surge] P. Barford, and M. Crovella. “Generating Representative Web Workloads for Network and Server Performance Evaluation.” In the *Proceedings of the ACM SIGMETRICS*, pp. 151-160, Madison WI, November 1998.

[TG] TG Project: http://www.caip.rutgers.edu/~arni/linux/tg1.html

[TGJS02] H. Tangmunarunkit, R. Govindan, S. Jamin, and S. Shenker. “Network Topology Generators: Degree based vs. Structural.” In the *Proceeding of ACM SIGCOMM*, 2002.

[Thrulay] Thrulay Project: http://sourceforge.net/projects/thrulay/

[TMR] The Transport Modeling Research Group (TMRG): http://www.icir.org/tmrg/

[TrafGen] R. Chinchilla, J. Hoag, D. Koonce, H. Kruse, S. Ostermann, and Y. Wang.

"Characterization of Internet Traffic and User Classification: Foundations for the Next Generation of Network Emulation." In *the Proceedings of the 10th International Conference on Telecommunication System, Modeling and Analysis* (ICTSM10), 2002.

[TTCP] TTCP Project: http://www.pcausa.com/Utilities/pcattcp.htm

[VCL] Virtual Computer Lab: http://vcl.ncsu.edu/

[WHYNET] WHYNET Project: http://whynet.ucla.edu/

[WIL] WAN in Lab Project: http://wil.cs.caltech.edu/

[YSVS00] M. Yuksel, B. Sikdar, K. S. Vastola, B. Szymanski. "Workload Generation for NS Simulations of Wide Area Networks and the Internet." In *the Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS), part of Western Multi-Conference (WMC)*, pages 93-98, San Diego, CA, 2000.

[ZCD97] E. W. Zegura, K. Calvert, and M. J. Donahoo. "A Quantitative Comparison of Graph-based Models for Internet Topology." *IEEE/ACM Transactions on Networking*, December 1997