Network Characterization Service (NCS)*

Guojun Jin George Yang Brian R. Crowley Deborah A. Agarwal

Distributed Systems Department
Lawrence Berkeley National Laboratory
1 Cyclotron Road, Berkeley, CA 94720
{g_jin, brcrowley, daagarwal}@lbl.gov

Abstract

Distributed applications require information to effectively utilize the network. Some of the information they require is the current and maximum bandwidth, current and minimum latency, bottlenecks, burst frequency, and congestion extent. This type of information allows applications to determine parameters like optimal TCP buffer size. In this paper, we present a cooperative information-gathering tool called the network characterization service (NCS). NCS runs in user space and is used to acquire network information. Its protocol is designed for scalable and distributed deployment, similar to DNS. Its algorithms provide efficient, speedy and accurate detection of bottlenecks, especially dynamic bottlenecks. On current and future networks, dynamic bottlenecks do and will affect network performance dramatically.

1. Introduction

Even though the network is increasing in speed, the utilization of high-speed networks is still not optimal. Common problems are congested links and an inability to use available bandwidth. The underlying problem is that it is difficult to know and control traffic on the network. Imagine this situation: two major roads cross at an intersection without a traffic light. There is no traffic report to let people know what the traffic is like around the intersection. Everyone thinks these two roads are the best way to go and decides to use them at the same time. Without a traffic light arbitrating the flow, a traffic jam naturally occurs. Without a traffic report no one knows about the traffic jam and people continue to drive along the two major roads. This intersection is similar to a congested link in the network. Another analogy is a road with three lanes that allows multiple vehicles to drive in parallel. If a transport dispatcher thinks it is a single lane road, he will send out one truck at a time. Traffic will be good, but throughput will be low. This case is similar to a network with an under-utilized throughput. These are problems that need to be solved.

In this paper we differentiate between static and dynamic bottlenecks. A *static bottleneck* is a network element (link, router, or switch) that has the minimum bandwidth along a path. A *dynamic bottleneck* is the network element with the minimum available bandwidth on a given path at a specific time. The dynamic bottleneck normally reflects *available bandwidth* of a path. The available bandwidth of a network element is its physical bandwidth minus the bandwidth of the network element currently used by other traffic. A key feature of available bandwidth is its dynamic nature. To measure available bandwidth at a specific point in time is not very meaningful due to its volatility. What we are actually measuring is the average usable bandwidth over a certain period of time. If an application is doing large data transfers, then we are usually interested in the available bandwidth over a longer period of time. But, if we are monitoring the network to do congestion control, we use a shorter time frame so that we can adjust the traffic flow in a timely manner.

This paper provides an overview of the design and implementation of the NCS daemon (NCSD) which is designed to measure network characteristics including dynamic bottlenecks. We discuss the features, capabilities and limits of NCS. We also introduce a desktop version of NCS — pipechar, and describe the differences between NCSD and pipechar. In Section 2 we present the related work, and in Section 3 we discuss the design goal and benefits of NCS. We describe the algorithms of NCS in Section 4; and the mechanisms for running NCS as a daemon and user-level program are described in Section 5. Results are presented in Section 6 which is followed by a summary of the paper.

* This work was supported by the Director, Office of Science. Office of Advanced Scientific Computing Research. Mathematical, Information, and Computational Sciences Division under U.S. Department of Energy Contract No. DE-AC03-76SF00098. This is report no. LBNL-47892.

2. Related Work

Several groups have designed mechanisms to improve performance on the current Internet. These mechanisms include Differentiated Services [DiffServ][RFC 2474/ 2475], RED [FJ93], SFB [FKSS01], AltQ [Cho98], Adaptive Gateways and others [RFC2990] [Jac88]. Most of these algorithms or mechanisms require knowledge about a number of network characteristics. Without this information, guaranteeing QoS on shared networks is almost impossible. The simple network management protocol (SNMP) [Bla95][Mil97] provides this information, but it requires router access privileges which are not usually available to ordinary users. Cprobe [CC96] is a nice tool that uses a high-resolution timer, but this feature is SGI-specific.

Current bandwidth measurement tools based on these technologies are aimed at measuring the static bottleneck and maximum bandwidth, which are not necessarily the most important factors for TCP performance tuning. For example, the Receiver Only Packet Pair (ROPP) [Pax97] tool with the compacted packet-pair feature can determine the maximum static bandwidth [LB99] at a bottleneck link. But, ROPP is not useful for TCP tuning because it filters out congestion information to find the maximum bandwidth at the bottleneck link. However, the available (dynamic) bandwidth on this link and/or the entire path may be smaller than its maximum static bandwidth. Using the maximum static bandwidth can result in setting of a TCP window, which is too large and reduces the TCP throughput [Tie01] [SM98].

Some other tools like pathrate [DMR01] and netest [Jin91] use packet trains, also known as 'Packet Bunch Modes' (PBM) [Pax97]. These tools that rely on packet trains can measure available bandwidth of a path fairly accurately, but they are not able to identify the network element causing the bottleneck. Information about the location of the dynamic bottleneck is critical to both congestion control and TCP tuning since it affects the round trip time calculation. But, measuring available bandwidth in a relatively short time frame is almost impossible in high-speed network. Also, pathrate and netest require a receiver at the remote end, which is not always possible.

There are several methods currently available for acquiring information about network characteristics such as bandwidth and latency with only ordinary user-level privileges that do not rely on things such as SNMP router queries. These methods are commonly based on sender-only, sender-receiver paired, and receiver only technologies [Pax97] and [LB99], as well as others [Sav99]. Current bandwidth measurement tools have other problems such as:

- lack of information about the location of a static bottleneck;
- large amount of time needed to acquire information;
- low accuracy of information;
- inability to measure high-speed network links;
- no easy method to share information;
- lack of APIs for local and remote access;
- difficult deployment including modifications to the operating system kernel;
- only available on a specific platform

3. Purpose of NCS

The primary design goal of NCS is to provide easily accessible information about the network and encourage applications to cooperate and optimize their traffic flow in the Internet. Tuning the TCP window size is a well-known technique to improve network bandwidth utilization, but getting reliable information, such as congestion location (distance measured by RTT), congestion pattern and frequency, destination distance (RTT), and bottleneck bandwidth, is relatively difficult. Applications need a way to determine an optimal buffer size, since setting an arbitrarily large TCP window not only reduces throughput, but also hurts the performance of the entire network [Tie01] [SM98]. The performance chart related to TCP windows resembles a Gaussian distribution. A too small or too large TCP window size will result in sub-optimal throughput.

Knowledge of the bandwidth and the distribution of the dynamic load is necessary to avoid congestion at a router. To accomplish this, NCS must identify bottlenecks on a specific path and determine if they are static or dynamic. If a bottleneck is determined to be dynamic, it needs to be monitored with a different methodology than if it is determined to be static. This kind of information will not only be useful to applications, but also to gateways using adaptive routing protocols.

High efficiency, minimal network traffic interference, and timely and accurate information are all important goals of NCS. It is designed to probe for available bandwidth for each segment in a path for a given time frame, and to cache acquired information for fast retrieval and exchange. Caching also helps with assembling and concatenating network segment information in order to reduce redundant probing.

NCS is designed to run as a daemon on each subnet, and NCS daemons on different subnets can cooperate to make measurements. The NCS daemon is also able to make measurements in single server mode. In this mode a single NCS daemon gathers network information autonomously.

NCS is primarily intended to solve problems on heterogeneous wide-area networks, not local-area networks. It is expected to be deployed one per domain or gateway to achieve monitoring capabilities, provide information services, and to perform troubleshooting.

4. Algorithms

4.1. Terminology

To aid in our explanation of the NCS algorithms, we need to first introduce some terminology.

Packet train — A packet train consists of a series of cars where each car contains one or more packets (MTUs). The car defines a measurement unit and cars in the train normally follow each other closely.

Hop-Differential (HD) — In a two-hop (with three nodes N_a, N_b, and N_c) path, the HD is the difference between the time to send a packet from N_a to N_b, and the time to send the same size packet from N_a to N_c.

Size-Differential (SD) — In a path with two end nodes N_a and N_b, the SD is the difference between the time to send a packet with size S from Na to N_b, and the time to send a packet with size $S + \Delta s$ from N_a to N_b.

4.2. Measurement methods

In order to build an algorithm to characterize the network correctly, we must first have a fundamental mathematical model. We then need to develop algorithms from this foundation. We present here our mathematical model for network status probing and algorithms for bandwidth computation using the "sender-only" method. In the "sender-only" method, transmission and reception are at the same location; there is no active receiver at the remote end to measure and feedback the probing request. This is a complicated method but very powerful since it requires access to only the sender of the measurement. We derive a number of algorithms to probe the network using the sender-only method.

single packet with size differential calculus at the same node (SPSD).

If we had a receiver running on a remote host, the receiver process could record the time between the arrival of the first bit and the last bit of a packet; we refer to this statistic as T_{ps} (traveling time per packet size). If we knew T_{ps}, the link bandwidth could be computed by dividing the packet size (S) by T_{ps}. Unfortunately, in the sender-only method, it is not so easy to measure the time, T_{ps} since we do not have a receiving process on the remote node. So, we send two packets with different sizes (X and Y) to a remote node and use the size-differential (SD) model to compute T_{Δ} (difference of traveling times between different sizes of packets). The traveling times for the packets are determined from the time to receive an acknowledgment for each packet. Figure 1 shows the transmission timeline for these packets.

$$T_{\Delta} = T_{py} - T_{px}$$

(T_{px} and T_{py} are the time between sending the packet and receiving the acknowledgment for X and Y respectively)

$$T_{px} = (T_{pd} + T_{psx}) + T_{qx} + T_{ack\_x}$$

$$T_{py} = (T_{pd} + T_{psy}) + T_{qy} + T_{ack\_y}$$

(T_q includes queuing time for both directions, T_{pd} is the propagation delay, T_{ps} is the time to send the packet, and T_{ack} is the time for the acknowledgment to travel back)

on an empty network (no cross traffic)

$$T_{ack\_y} = T_{ack\_x} \text{ and } T_{qx} = T_{qy} = 0$$

so,

$$T_{\Delta} = T_{psy} - T_{psx} \qquad \text{<}\mathbf{F\text{-}1}\text{>}$$

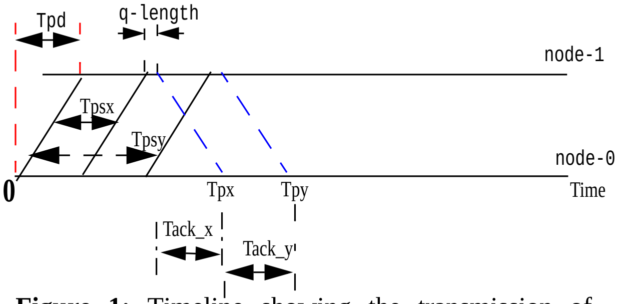

Figure 1: Timeline showing the transmission of packets X and Y and their acknowledgements.

Once we determine T_{Δ}, we can calculate the bandwidth by using Δsize ($\Delta s = S_y - S_X$) divided by T_{Δ}:

$$BW = (S_Y - S_X) \div T_{\Delta}$$

Therefore, using T_Δ and Δs to compute bandwidth is essentially equivalent to using T_{ps} and size S in an empty network.

The above calculations assumed that the round-trip time remains essentially constant for a given packet size. But, the maximum size of the packet is limited by the MTU standard of the network interface. For example, if the network interface is ethernet, the maximum Δs is less than 1472 Bytes. When a link bandwidth (BW) is OC-3 or better, the T_Δ will be smaller than $1472 \times 8 \div 155x10^6$ = 75.974 µs. A typical non-local round-trip time (RTT) is greater than 1ms and has a ±5% error rate, so the deviation of RTT (Δ_{RTT}), is greater than 50µs. Under these circumstance, the formula <F-1> becomes

$$\begin{aligned} T_\Delta &= Tpy - Tpx \\ &= (S_y \div BW + RTT_y) - (S_x \div BW + RTT_x) \\ &= T_{\Delta(\text{zero traffic})} \pm \Delta_{RTT} \end{aligned} \qquad \textbf{<F-1'>}$$

where

$$RTT = T_{sys} + T_{pd} + T_{ps} + T_q + T_{ack}$$

T_{sys} is the time (7-15 µs on idle, or tens µs on busy systems) to send a packet from user space to the edge of a network interface card (NIC) or the reverse.

The time difference between the largest packet and the smallest packet that can be transmitted from a source host to a middle router is inaccurate when Δ_{RTT} has a magnitude similar to $T_{\Delta(\text{zero traffic})}$, and thus dominates T_Δ. So, this algorithm is only good for probing networks with capacity up to 100Mb. To solve this problem, we need the ability to create a larger packet differential to get accurate results and probe faster networks. The solution to this problem is to use multiple packets.

multiple packets with size differential calculus at the same node (MPSD).

The MPSD algorithm uses multiple packet trains to measure a path hop-by-hop. The SD algorithm is used to determine the T_Δ within a packet train and the T_Δ between packet trains.

Dynamic bottlenecks (as opposed to static bottlenecks) are caused by cross traffic in the network. The MPSD method combined with statistical analysis allows us to analyze cross traffic. When packets travel together and pass a congested node without separation, they exhibit a bunching effect, and we can use this fact to detect dynamic bottlenecks. Dynamic bottlenecks can be recognized by queuing delays; when a packet train is sent on a network, the packets may become separated by queuing due to cross traffic on busy routers. We can use this separation to measure the queuing delays. The distribution of the queuing delays can be used to calculate available bandwidth on a particular node. Available bandwidth can also be calculated by using linear regression to find where the measured queuing delays converge and maximum physical bandwidth can be estimated by eliminating queuing delays using differential calculus.

We are also exploiting queuing in our algorithm to produce packet bunching. In an uncongested network a single stream will not create queuing if every upstream link on the path has the same speed. We need cross traffic to cause queuing at the router. By identifying a path that starts or crosses at a slow network interface, we can use multiple NCS instances to induce cross traffic and create packet bunching in the network. This can allow us to correctly probe nodes beyond the slow network interface. When NCS identifies a point in the network where it suspects a slow interface, it injects an additional traffic stream to provide cross traffic and create a dynamic bottleneck in the path. This is intended to ensure that the packets leaving the dynamic bottleneck will leave the interface with no space between them. This allows us to effectively speed up packet trains and probe links beyond a slow interface. Although we can use packet bunching to discover network characteristics, it is not intended to be used in normal conditions since it creates additional traffic.

single packet with the same size on hop differential calculus (SPHD).

This algorithm has been used in tools like pathchar[Jac97] and pchar[Mah99] to measure bandwidth for links. In this algorithm, a packet with size S_p is sent to two adjacent nodes. The bandwidth for the link between the two nodes is calculated using the following formula:

$$BW = S_p \div (T2 - T1 - T_{pd1\text{-}2}) \qquad \textbf{<F-2>}$$

where

T1 is the time to transmit packet P from a source to node 1

T2 is the time to transmit packet P from a source to a further node 2

$T_{pd1\text{-}2}$ is the propagation delay between the two different nodes. The propogation delay can be measured by using a packet of size S_p = 1 bit.

If we use a packet of size 1 bit to measure T_{pd1-2} then we must subtract that packet from the size Sp so, the more accurate <F-2> is

$$BW = (S_p - 1) \div (T2 - T1 - T_{pd1-2}) \qquad \textbf{<F-2'>}$$

Unfortunately, different routers have different ICMP response times. This makes it difficult for algorithms using hop differential measurements to be accurate. For instance, pathchar and pchar sometimes give negative results [Dow99]. We do not use the hop differential (HD) calculation algorithm in NCS for this reason.

4.3. Physical limitations

Network measurement tools often end up measuring end system characteristics instead of the network bandwidth. Before implementing a system based on our algorithms, we need to investigate the physical capabilities of the hardware and software so that we can correctly map the mathematical model to a physical model. The mathematical model describes an ideal state. The physical realm has many facts that are not ideal. Before trying to map our model to the physical world, we need to understand the characteristics of all the physical components, such as hardware, software, and operating systems. The major issue involved is accuracy of the timing measurements. The Δ_{RTT} in <F-1'> is affected by the clock resolution and other timing issues introduced by the hardware device drivers and operating system context switches.

Hardware capability

The most popular network interface adapters (NIC) are PCI bus-based. The typical hardware platform is x86 and the x86 based machines are equipped with one or more 32-bit/33MHz PCI buses for the Input/Output (I/O) subsystem. This I/O subsystem provides a maximum bandwidth of 132 MBps, which is equivalent to 1 Gbps or Gigabit speed. However, in practice the full 1 Gbps I/O throughput is not usually achievable on the PCI bus.

Now let's consider the throughput from the user space to the network wire (outgoing direction). The typical memory bandwidth provided by most current x86 motherboards, using 100MHz DIMM, is about 250∼300 MBps. When data travels from the user memory space to the kernel memory space (e.g., mbufs in BSD OS), it takes two memory cycles. Copying data from a mbuf to the NIC takes 3 memory cycles. A total of 5 memory cycles are needed to send user data to the NIC. The maximum throughput for this outgoing data is 400∼480 Mbps (50∼60 MBps) from user space; or 664∼800 Mbps from the kernel level. To measure a Gigabit network, we cannot stream data fast enough by simply sending the packets one-by-one from the user space.

In the incoming direction we have similar problems. When two 1500 byte packets arrive from a Gigabit Ethernet (GigE) network to the NIC, the device driver copies the first packet from the NIC buffer to a system buffer. This copy process takes almost the same time as for the second packet to arrive at the NIC. The device driver has to then finish processing the second packet before returning the CPU control to the operating system for the context switch. After this, the two packets are added to the incoming network queue — Q_{in} — for applications to read. When a context switch wakes up the user application waiting on this socket, the application sees both packets, reads the first packet, and starts timing the arrival of the second packet. Since the application can actually read the second packet immediately, the time required for processing the second packet is the time it takes to copy it from kernel space (Q_{in}) to the user space across the system memory bus. Thus, the timing measurement will reflect the memory bus speed rather than network bandwidth.

Device driver time delay

The device driver can also introduce skew in the measurements. In a well programmed x86 network device driver the receiving process consists of at least a dozen subroutine calls and at least 100 instructions, and takes ~10 µs of execution time on a 500 Mhz Pentium-II and a 750MHz AMD-K7 machines. If an pair of packets arrive back-to-back at the NIC from an OC-12 link, the time difference between the last bits of these two packets is 19.3µs. On a x86 machine with a 32-bit/33MHz PCI bus, copying a 1500-Byte packet from the NIC to a mbuf takes ~11 µs. Thus, without considering the execution time of any other overhead, the receiving process takes ~21µs to process a 1500-Byte packet at the NIC and put it into the incoming network queue. Then, the device driver has to finish the processing for the next packet before releasing the CPU. So, the user level application will see these two packets separated only by the time needed to copy the second packet from kernel memory to the user memory.

The skew caused by the device driver and the hardware (e.g. PCI bus) can introduce a chain reaction when many large packets arrive at a Gigabit NIC back-to-back. To avoid this chain reaction, the size of the timing packets has to be small. In the sender only method, the reply packets are ICMP and so their sizes are between 56 and 92 bytes; small enough to avoid the chain reaction. When NCS is used with a sender and a receiver (not described in this paper), we insert a 28-byte UDP packet between the cars for timing purposes. The extra time caused by the timing packets is subtracted out when calculating the time differential between cars.

Context switch effects

If the period measured by a timer spans an operating system context switch, then the measurement will include the time spent doing the context switch. This time should not have been included and will lead to a low estimate of the bandwidth. Generally, a process gets 10 ms of execution time between context switches. In many cases, the round trip times (RTT) between hosts on the Internet is steadily increasing. Ten years ago, the Internet connection from LBNL to ftp.freebsd.org (15~20 miles) was 7 hops and had a 50 ms RTT. Today, this connection has 15 hops and a 98~130 ms RTT, and more than 50% of the hops on the path have an RTT greater than 10 ms. When measuring long RTT path segments, a context switch is likely to occur and introduce significant error. To avoid this effect, we complete all non-timing related processes and then voluntarily context switch our process out for at least 20 to 25 context switch time slices to raise the process's priority. Then the process must be scheduled to run immediately after packets start arriving.

The above hardware and system issues render packet-pair technology unusable for measuring high-speed networks. This is the reason why packet-pair based tools, such as nettimer [LB01] and pathrate, usually either over or under estimate the bandwidth. Pathrate has identified this over/under estimate issue and the table II in their paper [DRM01] shows similar results — 26~28 Mbps — going between a 100Mbps and a 10Mbps host, and between two 100 Mbps hosts. Below is the output of nettimer-2.3.2, [LB99][LB01].

```
nettimer-static-2.3.2 output:
dpsslx04% nettimer --run_dpcap_server

dpsslx01% nettimer --dpcap_servers "localhost dpsslx04"
Cmds: (q)uit, any other key to update
FlowSource   FlowDest      FI TI Metr    Bandwidth (bps)
dpsslx01     solaris        0  0 SBPP       862.10
solaris       dpsslx01     -1  0 ROPP       416.24
dpsslx04     dpsslx01     -1  0 ROPP 1,200,000,000.00
dpsslx04     dpsslx01     -1  1 ROPP      1343.95
dpsslx04     dpsslx01      1  0 RBPP 1,200,000,000.00
dpsslx04     dpsslx01      1  1 SBPP      1,343.89
dpsslx01     dpsslx04     -1  1 ROPP       416.00
dpsslx01     dpsslx04      0  0 SBPP       416.05
dpsslx01     dpsslx04      0  1 RBPP  41,600,000.00
dpsslx01     dpsslx04      1  1 SBPP       416.04
solaris       dpsslx04     -1  1 ROPP   3,355,472.57
```

This data is the result of running nettimer measurements between two 500 MHz x86 hosts running the Linux operating system, with 1.2Gbps memory-copy bandwidth. The hosts were both on a Gigabit Ethernet network. The 1.2Gbps nettimer result reflects the system memory bandwidth. The 41.6 Mbps and lower reports are under estimates of the actual available bandwidth. The correct result should have been between 320~800 Mbps.

4.4. Algorithm description

SPSD and MPSD can be used to probe every node on a given path. The SPSD algorithm is a simple and fast algorithm, but it is only accurate enough for networks with a throughput less than 100Mbps. To measure high-speed network links, we use the multiple packets with a size differential algorithm (MPSD).

The model of the measurement packets in Figure 1 assumed that the packets were sent out at the same time, but this is not physically possible using a single network interface card. There are two possible methods: asynchronous and synchronous packet trains. In the asynchronous train approach, different length trains are sent at different times. Each train contains only one car and these trains travel over the network at different times. So, they might have experienced different network conditions and we cannot simply calculate the time differential between trains. Instead, we collect the times for running the same size trains. The results from individual trains are then processed using a convergence algorithm. We keep running packet trains until we reach convergence of the time differential calculations. The speed of convergence depends on the network status and train length. The better the network condition is, the faster the convergence occurs, and the smaller the number of trains that will be required. If long trains are used, convergence will normally be rapid since the long trains tend to average out network fluctuations. These long trains are good for tuning TCP buffer sizes since TCP needs a relatively stable buffer size. Short trains are normally used to determine congestion control information but short trains may take longer to converge.

The synchronous method sends one multiple-car train to a node to analyze the queuing delay, and measure the available bandwidth at a specific time frame. The cars are queued at the probing machine, and then flushed to the network. On an idle network, the physical implementation's timeline will be the same as the mathematical timeline in Figure 1. In the physical implementation, many components will introduce skew as described in section 4.3. Because an NCS timing packet is small, the effect introduced by the NIC device driver on network measurement is minimized. To determine the skew, the first timing chain is analyzed, but not used for the measurements. The measurement will then use timings from the rest cars.

There is a limitation on the train size (number of cars) that can be used in the synchronous method. If the head of the train returns to the sender before the rest of the train has been sent then there will be error in the measurement introduced by the extra loading on the network interface card and the system. So, the train size depends on the RTT and the link speed. That is, the train size (num_cars) has to be shorter than:

$$\text{max\_cars} < RTT \div T_{car}$$

T_{car} is the time for a car to travel through the slowest node along a path. On an OC-3 network with 0.35ms RTT, the maximum train size is:

$$35 \times 10^{-5} \div (MTU \div (155 \times 10^{6})) = 4.5 \text{ (MTUs)}$$

Unfortunately, a 4-car train may not be long enough to characterize a network feature due to the noise introduced by the hardware. So, a synchronous train may not be able to probe gateways correctly when the RTT is short. The train size for a synchronous train is determined by running an asynchronous train prior to running the synchronous train. The length of the synchronous train is then varied from hop to hop based on the RTT of the target hop and the path speed.

The synchronous approach analyzes the time variation between each car. It measures the queuing delay caused by cross traffic for each car, which is used to estimate the available bandwidth.

5. Building a NCS Infrastructure

5.1. A desktop version NCS — pipechar

Pipechar is a user level application that provides a command line interface to a subset of the capabilities available in the NCS sender-only mode. Pipechar runs as an application rather than a daemon. This offers users and network analyzers a convenient way to watch a network path from anywhere. For example, if you were having network problems when traveling in a desert with your wireless laptop computer, there might not be a NCS daemon monitoring the wireless network. With pipechar on your laptop, you could easily probe the suspected network path, analyze the problem and determine the node that is causing the problem.

Pipechar only contains NCS functionality useful for one-time probes; it is not designed to collect information over different times frames, or help determine variations in available bandwidth. Pipechar reports the available bandwidth measured during the time frame specified at the command line. Since pipechar will only measure values less than the maximum bandwidth of the underlying wire, it also uses simple heuristics to guess the likely underlying infrastructure and its actual maximum bandwidth--for example, if the value is more than the maximum bandwidth of a OC-12 network and there are some cars that arrived at the system memory speed, e.g., 150MBps, then you are likely dealing with a Gigabit network.

5.2. NCS as a daemon

Ideally, the core areas of the network should be monitored relatively continuously. To achieve this goal, NCS was extended from a local service to a global service daemon — Network Characterization Service Daemon (NCSD). The NCSD currently consists of the following four modules: core service, user API, daemon API and global service interface. The core service and user API are the primary modules in NCS, and the daemon API and global service interface are extended service modules for operation as a daemon.

The core module is responsible for detecting, acquiring, analyzing and caching information about network links on a given path. It provides mechanisms to reduce redundant probing. In the single server model, two re-probing periods are scheduled. One re-probe is scheduled at higher frequencies on links with high utilization to reflect bottleneck status changes. Another is scheduled at lower frequencies to reflect global status changes. To monitor global status changes, re-probing traverses the entire path node-by-node in a status confirmation process. Both re-probing processes introduce minimal traffic impact on the networks being monitored.

When the service is running in mutual service mode, it exchanges shared path information between NCS peers, and uses packets to exchange information to complete the bottleneck re-checking process. In the mutual server model, the monitoring process is primarily in passive mode using ROPP at the remote server. The active re-probing is only scheduled once a remote server sends feedback about abnormal packet-pair traffic.

When NCS runs on an adaptive gateway, the probing process will use a piggyback like mechanism that reduces the active probing to zero. This is achieved by inserting two small UDP packets into a congestion control queue — Q_{cc} — for a specific destination, one at the head of the Q_{cc}, and one at the tail of the Q_{cc}. These two UDP packets are targeted to the same destination as the queued packets with an unserviceable destination port, so we will get two ICMP packets back with time separation to compute current available bandwidth based on the size of queuing data between these UDP packets.

The user API defines a set of protocols and data structures that allow applications to query the network information from NCS. These protocol and data structures are described in ncs-api.h in the NCS distribution. A basic query API library for C is also available in the NCS distribution; the client program, nscC_example.c provides the mechanisms to use this API.

The daemon API is used by the NCSD services to exchange information on the common network segments (links). The goal is to reduce the network probing times and avoid unnecessary network traffic. It combines the user API with the neighbor NCS information exchange protocol (NIEP), which is used for gathering shared link status to form a new path database. For example, in Figure 2, if status for paths A-F (A-a-d-e-f-g-F) and B-C (B-b-c-e-C) has been initialized, then to get G-E status, only the link g-E needs to be probed in single service model, and no probing is needed in mutual service model.

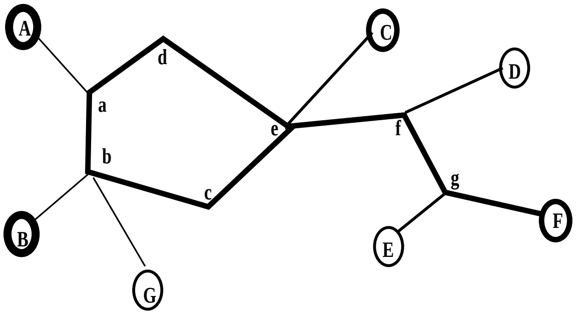

Figure 2: NCS information exchange

The NCS hierarchy is designed to operate in a manner similar to DNS. Every subnet might have a basic NCSD, and every area (e.g, a city) can use a level-1 NCSD to control up to 64K basic NCSDs in the area. A level-2 NCSD is deployed within a large region, such as the Bay Area or for a state. The level-2 NCS can control up to 64K level-1 NCSD, and level-3 NCS can control up to 64K level-2 NCSD, etc. The maximum number of levels that can be used in the NCS hierarchy is 4~8. The limitation in depth is created by the search algorithms used to assemble a new path from NCS caches. For example in Figure 2, in order not to probe the path G-E, we need to find whether any NCSDs have cached the link information for b-c, c-e, e-f and, f-g. The initial search inquiry is sent from a basic level NCSD to its leader (level-1 NCSD), and the leader starts the search algorithm to gather all required information and assemble a partial or completed new path characteristic database. This process involves LxMxNxP cache access. Where L is the number of NCS levels; M is the number of level-1 NCSDs invoked; N is the average number of basic NCSDs invoked at each level-1 daemon; and P is the average number of paths cached at each basic NCSD. In practice, L is usually small, so the linear search will normally be O(NxMxP). NCSD implements mechanisms to reduce the time of this search to a constant O(C), but these mechanisms will be described in a later paper.

These features, when combined with the appropriate choice of algorithms to collect the information, result in an easy to use service with a low impact on the network.

6. Experience with the NCS Implementation

In this section, we describe the NCS implementation and present some results from using pipechar on both normal and problematic networks. We compare the results with results from pchar and Web100 on the normal networks, and use pipechar to identify dynamic bottlenecks on the problem network. We also compare the processor usage and the amount of network traffic generated by the network measurements of the tools.

6.1. Implementation

NCS was originally developed for use with an LBNL implementation of adaptive gateways. It is written in C and uses an Auto-Configuration System (ACS) [Jin91A] that makes it relatively easy to compile. NCS has been compiled and tested on several common UNIX platforms, such as Solaris, Linux, FreeBSD, IRIX and Digital UNIX.

On FreeBSD, NCS makes use of some special features, such as the kernel timer and zero traffic probing. The FreeBSD operating system provides several advantages. First, since FreeBSD provides access to the kernel source, it is easier to obtain an accurate packet arrival time. Secondly, FreeBSD provides a user API to determine the NIC settings such as speed, duplex mode and connector type. Additionally, FreeBSD provides an easy to use development program interface, called KLD (dynamic kernel linker), which makes kernel related development much easier. Only one kernel file, /sys/net/if_ethersubr.c, needs to be modified to allow measurement of timing information for all incoming/outgoing packets and packet manipulation for zero-traffic probing. This patch code is available with the NCS distribution.

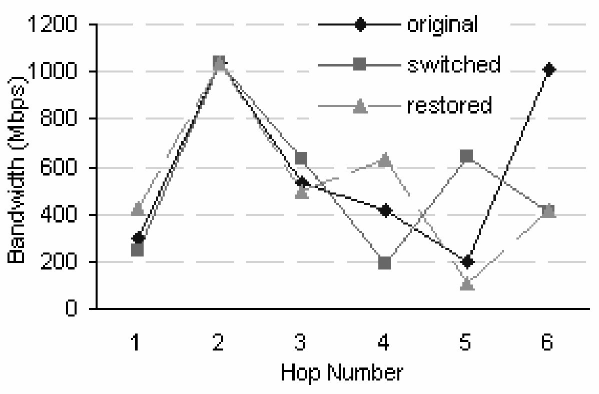

Figure 3: NCS Dynamic Bottleneck Detection

6.2. Measurement results

In Figure 3, a problem was reported from LBNL to goshen.mcs.anl.gov — GigE all the way, but the maximum throughput was less than 100Mbps. In this graph, line 1 shows how a dynamic bottleneck was detected by forcing packet bunching at router 5. When the Argonne personnel reset the routing path to check the correctness of detection of the dynamic bottleneck, they removed node 4, and put another router after 5 (to make the new path the same length as the old path), and we saw the bottleneck move from router 5 to router 4 (line 2 - switched), and we were still able to measure the rest of the links correctly. Argonne personnel confirmed that routers 4 and 5 had been switched. Line 3 shows the results after the Argonne personnel restored the original router configuration. The variation in measurements reflects changes in the dynamic bandwidth along the path.

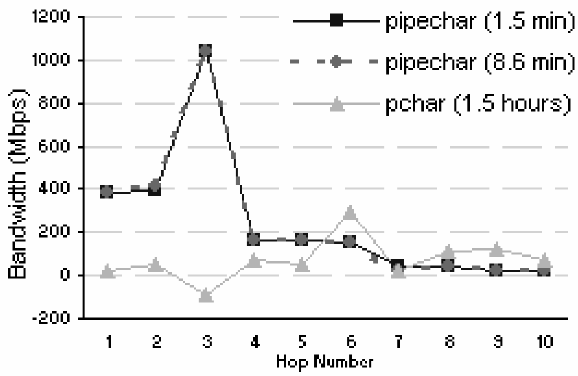

Figure 4: Comparison of pipechar and pchar

In Figure 4, we compare the amount of time required to run pipechar and pchar (execution times in parentheses) to probe every node on the path from LBNL to ANL (3 hops GigE, 3 hops OC-3, and 3 hops DS3) during normal traffic conditions. Pipechar measured correctly the available bandwidths of each of the links while pchar exhibited significant errors on the GigE and OC-3 links, and better results on the DS3 links. Pipechar took 1.5 to 8.6 minutes to generate the results while pchar took 1.5 hours to report on the same path.

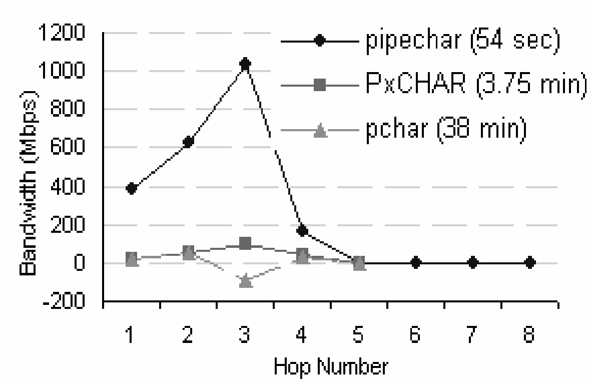

Figure 5: NCS, PxCHARand pchar results for a broken network path

When a problem of poor connectivity from LBNL to www.schwab.com was reported, we ran pchar, pipechar, and pipechar -PxCHAR (pipechar's modified fast pchar algorithm) on that path and got some surprising results. Figure 5 shows the results of the runs to the destination via ESnet. Pipechar shows that the bandwidth dropped sharply between hops 3 and 5, and only minor traffic was recorded between hops 5 and 8. This data shows that the connectivity problem was likely beyond hop 8 (hop 9 and further were believed to be part of a private network). Traceroute was also unable to penetrate beyond hop 8. ESnet provided us with physical bandwidths and MRTG graphs for the tested links. The information and plots confirmed the test results of pipechar. Confirmation was obtained by subtracting the traffic indicated in the MRTG graph from the physical bandwidth.

The low bandwidth between hops 4 and 5 in Figure 5 is because the link was shaped (carved) to provide T1

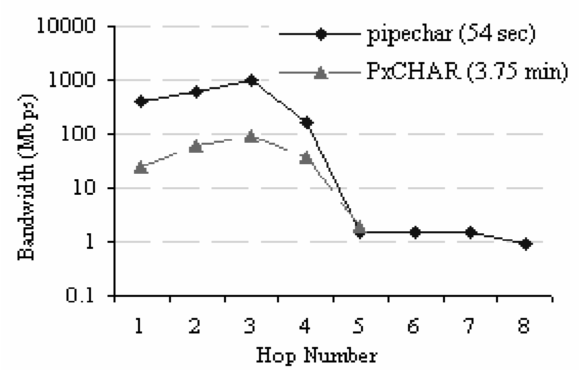

Figure 5a: NCS and PxCHAR results (log scale)

speed (see Figure 5a). This shaping caused the hop differential based probing to stop functioning. In Figure 5 we can see that both pchar and PxCHAR were unable to measure beyond hop 5, whereas, pipechar went through the shaped tunnel before stopping at the private network segment.

Figure 6 shows the results of tests between Web100 [Web99] experimental hosts at LBNL and SLAC. The path between these sites has a bandwidth asymmetry. A UDP stream from LBNL to SLAC can get 67Mbps, and 92Mbps in the reverse direction. But, a normal TCP stream from LBNL to a FreeBSD machine at SLAC gets around 6Mbps (34 Mbps if the TCP buffer size is tuned).

The pipechar result shows that the maximum available bandwidth from LBNL to SLAC is about 67 Mbps due to a bottleneck in the last link to the destination host. .

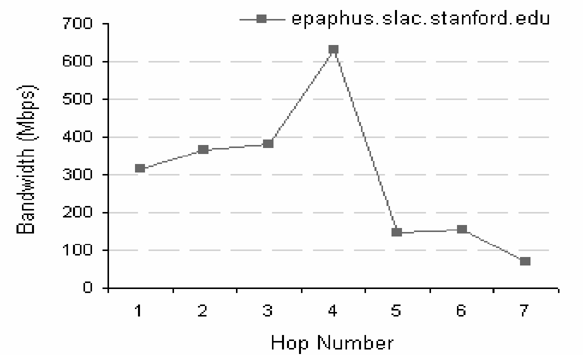

Figure 6: Pipechar test over an asymmetric path

Paths that are asymmetric in length will not affect our algorithm, because this asymmetry introduces a constant into the roundtrip time. It increases or decreases the RTT for each car, but not the time interval between cars. This change in the round trip time is subtracted out in our algorithm.

6.3. Traffic and processor utilization

Compared with other existing tools, NCS is a lightweight probing process. On a 200MHz x86 machine, the cpu usage of NCS is about 1% of the available capacity. On machines with a CPU faster than 500 MHz, the cpu usage is less than 0.3%. Network usage is also low. Since the train size is adaptive, we have a range of minimum to maximum packets. In synchronous mode, NCS sends 4-24 packets (one train with 4-12 cars) to a node (average of 14 packets); in asynchronous mode, NCS sends 3-8 trains (each with 6-12 packets) to each node (average of 54 packets). Pchar sends 1472 packets to each node and uses a size range between 32 and 1500 Bytes. Pathrate always sends about 4360 packets to measure an end-to-end bandwidth. Using the same number of packets as pathrate, NCS could have probed ~310 nodes in synchronous mode, or ~80 nodes in asynchronous mode.

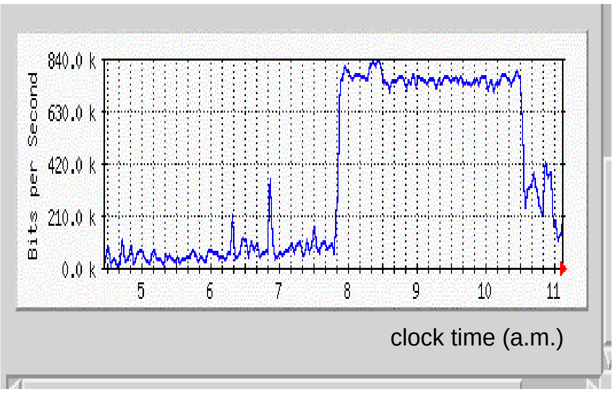

Figure 7: MRTG graph for pipechar BW usage at node 5 in Figure 5

Figure 7 is an ESnet MRTG graph that shows network traffic for a time period from 5am to 10am in the morning. NCS performed very aggressive probes at 10:37am, 10:51am and 10:55am. Each probe had a duration of 55 seconds. Figure 7a shows the 10am to 11am time frame expanded to show bandwidth used for probing. The rectangles indicate the testing periods. These probes correspond to the probes discussed in Figure 5.

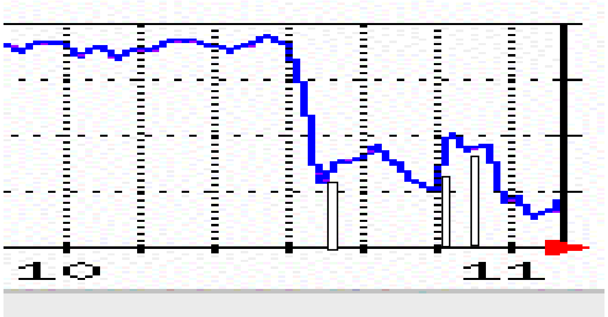

Figure 7a: An exploded view of 10:00~11:00 from Figure 7 with the pipechar probe times indicated

7. Summary

Available bandwidth, round trip time, dynamic bottleneck and load balancing are critical to congestion control and performance tuning. We presented in this paper a new mechanism, NCS, designed to provide this information. NCS can be run as a daemon or interactively using the pipechar command line interface. NCS's design and algorithms provide efficient, timely and accurate information for every link along a path. The NCS service does not require any special environment and can return information in real time.

NCS provides several important enhancements to the previously available algorithms for network measurement. These enhancements bring improvements in both accuracy of the results and efficiency of the network probing. It is our experience that a probe for available bandwidth of a path using pipechar completes in a relatively short amount of time. Early results indicate that NCS is not only a good network characterization tool, but it is also good for network monitoring, problem identification and troubleshooting.

8. Acknowledgments

We would like to thank Brian Tierney and Jason Lee for their feedback on this paper and for providing problematic networks for testing; and to thank Bill Nickless and other Argonne Lab. folks, Joe Burrescia, Chin Chen Guok and ESnet for assisting with our tests and confirming the results.

9. References

[Bla95] Uyless Black, Network management standards: SNMP, CMIP, TMN, MIBs, and object libraries. New York: McGraw-Hill, c1995.

[CC96] R.L. Carter and M.E.Crovella, "Measuring Bottleneck Link Speed in Packet-Switched Networks," Performance Evaluation, vol. 27,28, pp. 297-318,1996.

[Cho98] Kenjiro Cho, "A Framework for Alternate Queuing: Towards Traffic Management by PC-UNIX Based Routers", In Proceedings of the USENIX Annual Technical Conference, June, 1998

[DRM01] C. Dovrolis, P. Ramanathan, D. Moore, What do packet dispersion techniques measure? In Proceeding of IEEE INFOCOM, April, 2001.

[Dow99] Allen B. Downey, Using pathchar to estimate Internet link characteristics, proceedings of SIGCOMM 1999, Cambridge, MA, September 1999, 241-250.

[FJ93] Floyd, Sally and Jacobson, Van. Random Early Detection Gateways for Congestion Avoidance. IEEE/ACM Transactions on Networking, Vol. 1 No. 4, August, 1993.

[FKSS01] W. Feng, D.D. Kandlur, D. Saha, K.G. Shin, "Stochastic Fair Blue: A Queue Management Algorithm for Enforcing Fairness", IEEE INFOCOM, April 2001.

[Jac88] Van Jacobson. Congestion Avoidance and Control. In Proceedings of ACM SIGCOMM, 1988.

[Kes91] Srinivasan Keshav. A Control-Theoretic Approach to Flow Control. In Proceedings of ACM SIGCOMM, 1991.

[LB01] Kevin Lai and Mary Baker, "Nettimer: A Tool for Measuring Bottleneck Link Bandwidth", Proceedings of the USENIX Symposium on Internet Technologies and Systems, March 2001.

[LB99] Kevin Lai and Mary Baker. Measuring Bandwidth. In Proceedings of IEEE INFOCOM, March 1999.

[Mil97] Mark A. Miller. Managing internetworks with SNMP: the definitive guide to the Simple Network Management Protocol, SNMPv2, RMON, and RMON2. New York, M&T Books, c1997.

[Pax97] Vern Paxson. Measurements and Analysis of End-to-End Internet Dynamics. PhD thesis, University of California, Berkeley, April 1997.

[SM98] J. Semke, M. Mathis, and J. Mahdavi, "Automatic TCP Buffer Tuning", Computer Communication Review, ACM SIGCOMM, vol. 28, No. 4, Oct. 1998

[Sav99] Stefan Savage. Sting: a TCP-based Network Measurement Tool. In Proceedings of the USENIX Symposium on Internet Technologies and Systems, 1999.

[Tie01] B. Tierney, "TCP tuning guide for distributed application on wide area networks", USENIX & SAGE Login, vol. 26, No. 1, Feb. 2001

[DiffServ] http://www.ietf.org/html.charters/diffserv-charter.html

[Jin91] http://www.itg.lbl.gov/~jin/network/net-tools.html

[Jin91A] http://www.itg.lbl.gov/~jin/ACS.html

[Jac97] ftp://ftp.ee.lbl.gov/pathchar/

[Mah99] http://www.employees.org/~bmah/Software/pchar

[RFC2474] K. Nichols, S. Blake, F. Baker, D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, December 1998

[RFC2475] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang and W. Weiss, "An Architecture for Differentiated Services", RFC 2475, December 1998.

[RFC2990] G. Huston, "Next Steps for the IP QoS Architecture", RFC 2990, November 2000

[Web100] http://www.web100.org