SCAFFOLD Demonstration

Object resolution server and minimal L2 network configuration

March 1, 2010

1 Implementation

For our demonstration, we have implemented version 0.1 of the SCAFFOLD protocol and network architecture. To retain a degree of backwards compatibility while injecting the essential components of our architectural reboot into the network, we implemented the SCAFFOLD protocol using the IPv4 packet header, repurposing the source/destination IP address for flowID: 8 bit ssID, 8 bit hostID, 16 bit sockID, and source/destination port for objectIDs (16bit).

Although the lack of bits limits routing and naming scalability, reusing the IPv4 header allows us to explore and evaluate the SCAFFOLD design with a view to interoperability. We retain L2 compatibility with commodity switches by leaving MAC addresses intact. On the wide-area, placing the ssID bits in the upper portion of the IP address allow us to route between domains over BGP announced prefixes.

SCAFFOLD's network routers are OpenFlow switches. We set forwarding rules on IP prefixes for matching flowIDs in label routers, and exact port numbers for resolving objectIDs in object routers. Our decision to limit SCAFFOLD to manipulating UDP/IPv4 headers was also for rough compatibility with the OpenFlow specification. The controller is built on the NOX network control platform, which is designed to manage the forwarding tables of OpenFlow switches. Our extensions to NOX, about 1100 lines of python and 1500 lines of C++ in total, introduce an RPC-like API for managing host and object-related events.

The SCAFFOLD network stack provides a socket library that directly mirrors traditional BSD sockets. To enable rapid prototyping and to exploit existing socket functionality (*e.g.*, buffering), we implemented SCAFFOLD's stack mostly as a user-level network stack. Our implementation was tested and evaluated on machines running Ubuntu Linux, and it supported send and receive rates up to 324 Mbps (more in §3). The implementation consists of about 7500 non-blank lines of C++ code.

2 Test Environment

The SCAFFOLD test environment currently consists of a 6-node topology with 3 hosts, 2 label routers, and a single combined object router/controller, as depicted in Figure 1. Each node is an Sun X2200 server with 2 Quad-Core 2.3 Ghz AMD-64 CPUs and 3 GigE ports, running Ubuntu 9.04. The host nodes are configured with kernel-mode Click, SCAFFOLD stack daemon running in user-level Click, and the statically-linked SCAFFOLD stack application library. Label routers run an unmodified version of OpenvSwitch, which is an OpenFlow switch implementation, including a kernel datapath module. The object router/controller machine runs a modified OpenvSwitch that supports round-robin object resolution, along with the NOX-based SCAFFOLD object controller.

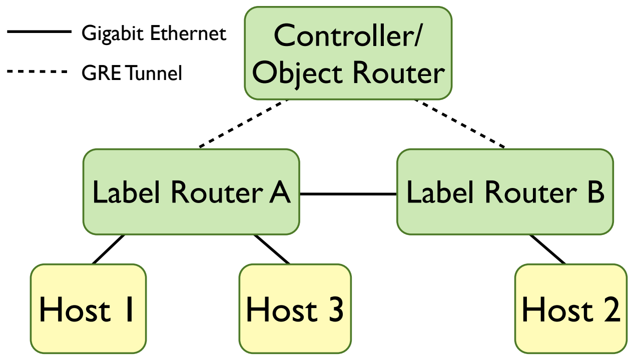

Figure 1: **Simple SCAFFOLD network topology for performance micro-benchmarks and mobility/fail-over experiments.**

Hosts 1 and 3 connect to label router A and host 2 connects to label router B. All links between hosts and label routers are switched GigE. Each label router has a direct connection to the other, as well as point-to-point GRE tunnels to the object router. All SCAFFOLD network elements are connected to a common switched GigE control network for establishing administrative channels to the controller.

Although the label and object routers reside in the same layer-2 subnet in our testbed, we do not assume this in the general case. Since SCAFFOLD flowIDs repurpose the IPv4 header, preventing direct routing of SCAFFOLD packets over IP, we leverage ethernet over IP GRE tunnels to provide L3 point-to-point connectivity across subnets between label routers and control plane elements .

We ported three network applications to demonstrate the performance and efficacy of the SCAFFOLD network: Iperf, TFTP (a UDP-based FTP program), and PowerDNS (a popular high-performance DNS nameserver). Modifying these programs to use the SCAFFOLD socket API required reasonably small changes: 240 lines for Iperf (out of 5934 LoC), 90 for TFTP (3452 LoC), and 160 for PowerDNS (15K

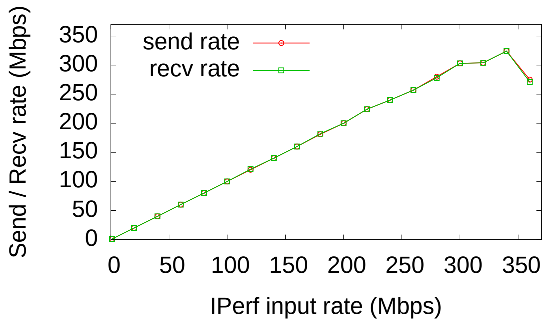

Figure 2: **Send and receive rate as a function of Iperf input rate for 1400 Byte SCAFFOLD bound datagrams. Packets are forwarded through 2 label routers between the Iperf client and server.**

LoC). Iperf required more changes, as it deals primarily with network performance measurement and thus has networking system calls throughout the code. In TFTP, we added an additional application-level continuation mechanism and adaptive RTT-based timeout for use in our failover and client mobility experiments; these totaled another 136 lines of code. Finally, we note that while we only modified PowerDNS's UDP-based front-end DNS interface (not the back-end pluggable storage), the port took less than 3 hours.

3 Throughput and Latency

Figure 2 shows SCAFFOLD throughput for bound datagrams between hosts 1 and 2 which are 2 label router hops away. With a 1400 byte payload, a peak send rate of 324 Mbps was matched by a receive rate of 324 Mbps, with 1.611 ms jitter and 0.019% packet loss. When the input bandwidth was increased beyond this rate, both send and receive rates dropped off as shown. The drop in send rate indicates a bottleneck in the SCAFFOLD user-level stack. An otherwise idle SCAFFOLD host transmits packets with a latency of 53.49 μs for 1472 byte payloads. The maximum packet loss observed during the test was 1.4%. Next, we measured the round-trip time (RTT) for maximum- and minimum-sized SCAFFOLD bound datagrams between the 2 hosts in our test environment with 2 label router hops between them. Stack latency was determined by recording timestamps when packets are sent or received on host interfaces and comparing them against application timestamps for the same packets. From Table 1, we infer that stack latency accounts for 54% to 60% of the RTT in our setup, which was around 500 μs. Microbenchmarks for various SCAFFOLD API methods are shown in Table 2. Note that *connect_sf* is a few orders of magnitude larger than the RTT, due to SYN packets requiring resolution at the object router.

From these numbers, we conclude that significant optimization—such as moving Scafd functionality into the kernel—will be necessary to reach the Gbps rates that BSD sockets provide today. However, having a prototype that offers reasonably good performance allows us to rapidly port applications and experiment with new protocol features.

Metric	Payload bytes	Mean μs	Stdev μs
RTT	18	397.16	47.57
RTT	1472	504.82	72.65
Stack receive latency	18	89.86	10.03
Stack send latency	18	48.21	8.71
Stack receive latency	1472	83.28	17.42
Stack send latency	1472	53.49	11.75

Table 1: **Network vs stack latency for bound datagrams**

Method	Task	Mean μs	Stdev μs
connect_sf	Object resolution and handshake	2925.00	494.18
bind_sf	Register an object with Controller	3069.40	141.58
send_sf	Send 18 byte payload to Scafd	69.21	20.84
send_sf	Send 1472 byte payload to Scafd	56.95	23.76
listen_sf	Set listening within Scafd	80.4	5.28
close_sf	Send FIN, and receive FIN-ACK	600.30	285.51
close_sf	Close socket on receiving RST	14.80	3.68

Table 2: **Latency of SCAFFOLD socket calls for bound datagrams**

4 Session Migration and Client Mobility

To evaluate SCAFFOLD's connection reestablishment facility, we performed two experiments to demonstrate session migration and client mobility. These experiments are meant to demonstrate SCAFFOLD's qualitative behavior, as opposed to providing any true quantitative evaluation.

In the first experiment, a TFTP client on host 1 begins to download from TFTP server 1 on host 3 over a bound flow. Note that these hosts are both physically connected to the label router A. Five seconds into the download, server 1 decides to shed load, *i.e.*, by calling `close(·, SEND_FAIL)` on its socket, which causes the client to receive a FAIL in response to its packets. The client stack on host 1 issues a RSYN to reestablish the connection with a new TFTP instance. TFTP Server 2 on host 2 (connected to the label router B) receives the object-router-resolved RSYN and completes the 3-way handshake. Note that this switchover is transparent to the TFTP client application. When the TFTP client resends its "failed" data ACK, this new server 2 receives the notification and resumes the download process in midstream. The migration is repeated again at 11.3s when server 2 closes its connection. As shown in Figure 3, the impact on client throughput and progress is negligible, with a less than 100ms delay introduced by the TFTP ACK-retry timeout. Process failover would work in a similar manner: The SCAFFOLD stack responds to packets destined to a failed socket with a FAIL message.

In the second experiment, a TFTP client on host 1starts a file transfer with a server on host 2, connected to label router B. Midway through the download, host 1 migrates from label router A to label router B: We literally unplugged host 1 from one switch and plugged it into the other. On link-up detection, the SCAFFOLD stack on host 1 re-issues a *join* request to the controller. When the controller sees that host 1 has

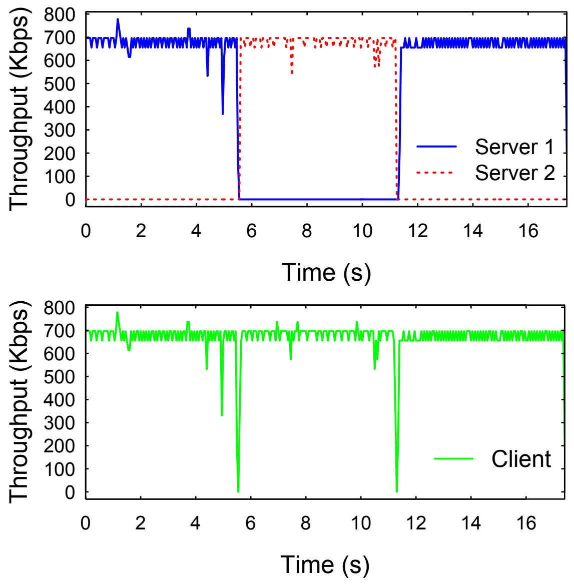

Figure 3: ***TFTP Server Failover. Top:*** **TFTP server 1 sheds an active client connection at 5.6s, causing server 2 to resynchronize transparently with the client. At 11.3s, server 2 offloads the same client again, causing the flow to migrate back to server 1. *Bottom:* The TFTP client's throughput remains consistent throughout its download, with only small outages ($< 100ms$) during failovers between servers.**

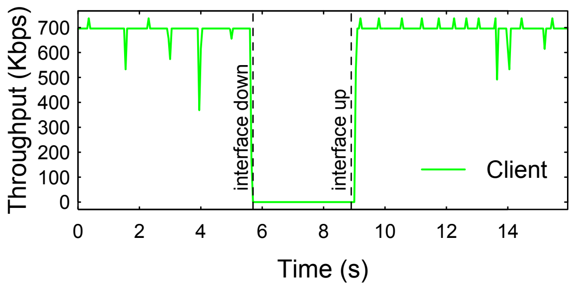

Figure 4: ***TFTP Client Mobility.*** **The client transfer is interrupted at the 5.7s mark when it "moves" between label routers. The interface revives at 8.9s and the client stack re-joins the network at its new location, resulting in a new hostID and triggering connection reestablishment. The transfer continues within 100 ms once the connection is reestablished.**

changed gateway label routers, it assigns host 1a new hostID, expunges the old hostID from the object and label routers, and reinstalls forwarding and object resolution rules referencing the new hostID label. Host 1's stack then resyncs its open connections, including the TFTP client's bound flow to the TFTP server. Once again, when the TFTP client's ACK-retry succeeds after stack reinitialization, the download continues without a hitch over the migrated connection.