NetServ: Early Prototype Experiences

Michael S. Kester Eric Liu Jae Woo Lee Henning Schulzrinne
Department of Computer Science, Columbia University
{msk2117, ewl2113}@columbia.edu {jae, hgs}@cs.columbia.edu

May 2010

Abstract

This paper describes a work-in-progress to demonstrate the feasibility of integrating services in the Internet core. The project aims to reduce or eliminate so called ossification of the Internet. Here we discuss the recent contributions of two of the team members at Columbia University. We will describe experiences setting up a Juniper router, running packet forwarding tests, preparing for the GENI demo, and starting prototype 2 of NetServ.

1 Introduction

As we close the first year of this multi-year project reviving some the concepts first proposed by the active networking community of the late nineties [1, 9, 10, 11], we describe here some of the experiences within the project.

Over the last year we have done performance testing of NetServ prototype 1 using various settings and configurations and compared it to other similar platforms. We have also begun implementation of prototype 2 which lays the groundwork for the next steps that will become the focus of research over the next six months. For describing the authors' contributions, the remainder of the paper is organized as follows: a discussion of packet forwarding and measurement; current progress on prototype 2; and the next steps in the project.

2 Packet Forwarding and Measurement

Over the course of the project we have tested and measured the forwarding performance of various systems ranging from dated consumer grade equipment to current commercial grade hardware [8]. The tests can be broken down into two categories based on connections and theoretical speed: a single 100 Mb Ethernet connection vs. 2x 1 Gb Ethernet connections.

2.1 Testing at Higher Speeds

In the summer of 2009, as a proof of concept, we tested the forwarding performance of user-level Click with and without NetServ Prototype 1 and compared this to bare Linux. In the fall, we extended this testing to more modern hardware and also measured the best performance Click could offer in kernel-level mode. This was used to gauge a best-case scenario for what performance we can hope to achieve porting NetServ into kernel space. We also added FreeBSD to the mix as it uses polling without any interrupts. Click performance here topped out at about 930 Kpps. FreeBSD showed nice potential at just under 1.3 Mpps.

At this time we also developed a couple of simple scripts and a parsing program to help collect and translate the raw text generated during testing. As we began to evaluate the high speed tests we sought to establish the maximum speed that our endpoints could reliably perform. We observed strange behavior at the destination node. We found we could only reliably generate and count approximately 1.8 Mpps as discussed in Section 2.2.1.

2.2 Testing on a Commercial Router

Testing of the commercial router involved two tasks: a sanity check on the observed 1.8 Mpps generation and counting limit at node 3 of the testing system, and obtaining a maximum loss free forwarding rate (MLFFR) for a Juniper M7i.

2.2.1 Limit to Packet Counting

We observed an upper limit on the packet rate that can be reliably counted in Linux; we consistently ran into

the limit around 1.8 Mpps when directly connecting the source node to the destination node. While we have not found a definitive explanation for the cause, we believe it is related to the SMP two-CPU setup we employ for testing. SMP Linux kernels try to optimize computational capacity by load balancing work across all available CPUs. At the same time, an attempt is made to try to keep individual tasks on the same CPU they have been running on. This is done for various reasons that include minimizing memory access latency. If a task stays running on the same CPU, it is more likely to have its data warmed in the cache. In contrast, switching CPU's will result in a flushing of the cache and having to reload data which increases latency.

Tests run by Bolla and Bruschi [2] show that there is a performance deterioration when going from a one to two-CPU system. They believe this performance deficit is due to the additional overhead and computational power needed to manage concurrency in an SMP system. For example, spinlocks in an SMP system could actually cause a contention for the same code region between processors resulting in one being stuck waiting. However, they did manage to come up with some optimizations to improve forwarding performance in SMP systems. These improvements resulted in better performance than a one-CPU system. Specifically, they achieved a forwarding rate of 1.8 Mpps which is consistent with the upper limit we mention above.

These issues apply to both hard and soft interrupt handlers relating to network cards. We tried to get around the issue in our testing by pinning interrupt handling for each NIC to a specific CPU. This keeps the hard and soft interrupts local to the pinned CPU. We also ran our tests using polling drivers which should bypass a majority of the interrupt problem. The fact that we still run into problems on the receiving computer at around 1.8 Mpps suggests we are likely not getting around the problem completely. We think there are two possible explanations. The first is that such a heavy processing load with polling is interfering indirectly with interrupts of other events. These could be unrelated events just having to do with background processes or they could be interrupt events having to deal with other elements of Click. The latter seems more likely since the problems manifest in nondeterministic behavior of Click at speeds near the threshold. The second possibility is that interrupts are not completely eliminated even with the use of polling drivers in Linux. Salah, et. al. [7] present new analysis and models for studying polling schemes and interrupt disabling and enabling for NIC's. Their work describes different types of polling in detail: pure polling and New API ("NAPI") polling. FreeBSD uses a pure polling mode that completely eliminates interrupts while Linux's NAPI polling is a hybrid interrupt with polling approach. NAPI does not eliminate interrupts completely and instead only disables them when the incoming rate of packets is high. It is highly likely that even with polling drivers installed and using Click, we are still using some interrupts on the system and, thus, interfering with other aspects of operation.

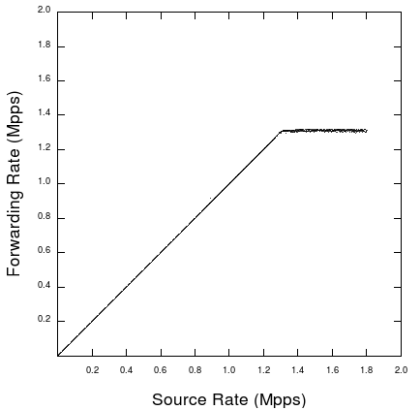

Figure 1: Juniper performance (MLFFR) using 64 byte packets.

In either event, we are satisfied that our results are reasonable based on the published results of other researchers.

2.2.2 Juniper Packet Forwarding

The Juniper forwarding was a natural extension of our previous work speed testing various configurations of Click routers and the Linux kernel. Having recently received a model M7i Juniper router, it made sense to perform our packet forwarding tests on this commercial-grade hardware and see how it compared to our previous tests. We somewhat expected the router to easily handle the load of 1.8 Mpps. However, the M7i's performance leveled off at approximately 1.3 Mpps.

The M7i is an entry level commercial grade edge router providing "ATM, channelized, Ethernet, IP services, and SONET/SDH interfaces for large networks and network applications, such as those supported by Internet service providers."[5] We suspect that it has been designed to target a reasonable throughput rather than optimized for maximum header processing. In other words, usage assumes average size packets rather than minimum size. The documentation published by the company provides conflicting throughput ratings. The "M7i Internet Router Quick Start" manual indicates "3.2 Gbps full duplex" while the M series products page on the website lists "10 Gbps of throughput" aggregate

and half duplex. Another source, the datasheet on the product line describes throughput as ranging "from over 7 Gbps up to 320 Gbps" for which the 7 Gb/s figure corresponds to the M7i. The description in the technical documentation suggests a total of 5 Gbps, 4 Gb/s across one or more Physical Interface Cards (PIC) plus a 1 Gb/s connection on the Fixed Interface Card (FIC). Multiple sources confirm that the device can be configured such that the throughput is higher than 5 Gb/s but the machine is then considered oversubscribed. For rough calculation of throughput ability we can make some assumptions about traffic and performance to rationalize our observations. First we note that, assuming the CPU of a system is the observed limiter to speed, throughput scales linearly with packet size leaving MLFFR unchanged by packet length. Next we estimate the average packet size on an edge router at 404.5 bytes per packet [6]. We use a 1.3 Mpps forwarding rate as observed in testing (Fig. 1). Given these assumptions we can estimate the throughput of our M7i at approximately 4.2 Gb/s which matches closely with suggested use listed by the company. We note that throughput changes significantly if you assume minimum or maximum (on Ethernet) sized packets. The same calculations result in 665.6 Mb/s using 64 byte packets and 15.6 Gb/s assuming 1500 byte packets. While we initially expected the M7i to perform much closer to the 1.8 Mpps we observed without any forwarding node, the performance it does achieve does not seem too unreasonable when considering that FreeBSD, which appears to have the best polling implementation we have encountered, performs similarly.

3 Netserv Prototype 2

3.1 Purpose

Prototype 2 is a significantly more robust implementation compared to its predecessor. Development is on an aggressive timeline motivated by the deadlines in place for demonstrations at The Eighth GENI Engineering Conference (GEC8) in July, and GEC9 in November [3]. We have taken an active role in two aspects of the prototype 2 development. We have attended the organizational meetings and contributed where appropriate, gaining greater insight into the needs, goals, and details of the project; and we have begun development of prototype 2 itself.

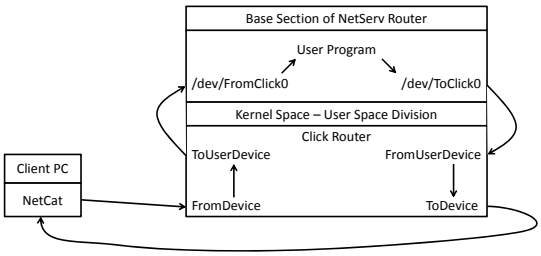

Figure 2: Packet path of a round trip from a client through the Click and user space portion of the prototype 2 base.

3.2 Demo Development

Because we had been given an invitation to the GENI demo to demonstrate NetServ's capabilities, a significant portion of our project work after the midterm presentation involved meetings with the NetServ team. It is difficult to quantify our contributions on this aspect of the project; however, our participation here was valuable for the team and especially for ourselves in seeing the bigger picture behind what can be done with NetServ. These were long and intense meetings in which we were active participants. Our work here included reading up on the background and purpose of GENI, understanding what would be expected of the NetServ team with respect to GENI, posing questions to other team members to help clarify the presentation of our storyboard, and giving input into how the demo should be presented.

A quick summary of the result is that there are two demos planned for GENI. The first is a VoIP satellite agent (VSA). A NetServ module can be used to help reduce traffic load for an SIP service when dealing with clients behind NAT's by both redirecting traffic itself and helping with keep-alive messages. The second demo will demonstrate how NetServ can be used in the context of a content distribution network (CDN). A NetServ module closer to a client's location can be used to cache content to reduce traffic load as well as for advertising or watermarking purposes.

3.3 Implementation

The base of Prototype 2 is being developed in two paths concurrently. One of the paths uses Click for filtering which packets to send to the module, actually passing the packets to user space, and then injecting them back into the packet stream (Fig. 4). The other path follows this same pattern but will rely on current Linux tools. We have begun work on the Click path and implemented the first layer of the model [4] which connects kernel level Click and an arbitrary user space pro-

gram. We use Click-supplied elements, ToUserDevice and FromUserDevice, to move packets from Click into user space and back via Linux device files. ToUserDevice can write a packet from Click into the device file. At this point, it can be read by a user-level program. FromUserDevice is the opposite. It reads from a device file and injects a packet into Click's packet stream.

In order to test the basic functionalities of these two elements, we started with a basic IP-router Click configuration and added the above mentioned elements. It was modified, specifically, to send UDP packets destined for the host's IP address and port 44444 to ToUserDevice. Our initial goal was to understand how to read the packet in a user-level application. Thus, we started with lower-level C code that could read from the device file in Linux. In order to make sure we were getting data out of the packet we expected, we used netcat to send ASCII to UDP port 44444 so that it would be redirected by the Click IP-router to the device file. We verified that the payload was what we expected.

With that accomplished, we extended the IP-router's functionality to then read a packet from the device file using FromUserDevice and send it back to the original sender using the IPMirror element. We modified the C code to write the same packet into the device file that Click reads from. The end effect is that any text a remote machine sends to UDP port 44444 using netcat is sent back by Click facilitated by the user-level C code.

After creating working C code, we were able to port this functionality to Java to allow for greatest flexibility of the upper level layers. Now that we can read and write whole packets, we will need to account for partial packets. A situation could potentially arise where only part of a packet is read into the user-level application. This will occur, for example, when the last packet read into the buffer of a user space program fills the buffer before reaching the end of the packet. The next step is allowing the user-level program to piece together packets from the device file using multiple read operations. This is not strictly necessary for the specific program we have described above since it is only sending ASCII text using netcat. However, our goal is to use this as the framework for the rest of prototype 2; thus, we want it to be able to handle a large number of packets under high traffic loads.

In order to better understand how to build the more general framework for prototype 2, we added code to lay out the bit structure of the packet we read. This will also allow us to understand how we might modify arbitrary bits of any packet. Fig. 3 shows a sample of the output generated by this bit parsing. This information will help us understand the structure of the packet in order to make any necessary modifications for redirecting a packet to the correct user-level service or NetServ module as proposed in the second prototype.

```
Version: 0100
Header length: 0101
Diff services: 00000000
Total length: 0000000000100001
ID: 0010101011011110
Flags: 010
Fragment offset: 0000000000000
TTL: 01000000
Protocol: 00010001
Header checksum: 1110011101001101
Source addr: 10000000001110110001010001011100
Dest addr: 10000000001110110001001111001110
Data: test
```

Figure 3: User space packet header parsing output.

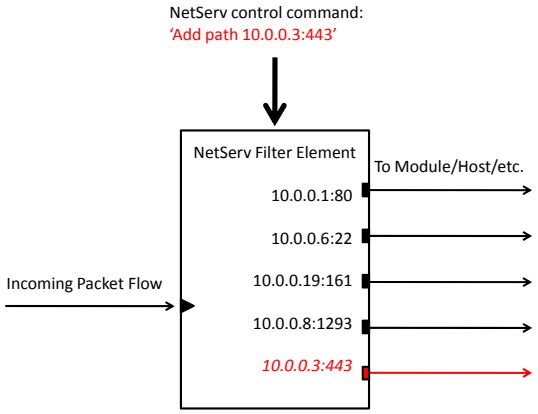

Figure 4: Custom click element dynamically adds a path to a newly installed NetServ module.

4 Future Work

As noted in Section 3.3, we are implementing the kernel-to-user transition between Click and user-level service containers. We will now be writing the code that will dynamically modify the Click path to pass arbitrary packets to the NetServ module. Once NetServ has dynamically downloaded and installed the code for a new NetServ module, it needs to tell the router it is running, what address and port it has, and how the path of packets through the router needs to change in order to accommodate the new module. We are continuing with Click as the routing platform and are coding a new element that will fulfill the role of filtering packets along

the correct paths. We will base our design largely on an existing Click element. Many elements deal with classification and selection of packets so we should easily find a starting point. Classifier, IPClassifier, and IPFilter are the most likely candidates. These elements typically receive packets from one input and can send them out any number of different outputs (paths) based on different criteria. Another promising category of elements are routing elements such as StaticIPLookup. These can also send out different outputs based on a set of specified rules. We will likely want to add handlers to the element we create so that there is more flexibility and control in router behavior. The above existing Click elements currently have few useful handlers for modifying their behavior.

As previously stated, Linux also provides methods to redirect packets to user-space. For example, IPTables also allows filtering of packets based on certain rules that are then redirected to a QUEUE that is user accessible.

References

[1] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The SwitchWare Active Network Architecture, 1998.

[2] R. Bolla and R. Bruschi. An effective forwarding architecture for SMP Linux routers. In *Telecommunication Networking Workshop on QoS in Multiservice IP Networks, 2008. IT-NEWS 2008. 4th International*, pages 210 –216, 2008.

[3] Geni exploring networks of the future. `http://www.geni.net/`, May 2010. BBN Technologies.

[4] J. W. Lee. Netserv demo at GEC9, synopsis v1.0. NetServ team, Columbia University, April 2010.

[5] M series multiservice edge routers. `http://www.juniper.net/techpubs/en_US/release-independent/junos/information-products/pathway-pages/m-series/`, May 2010. Juniper Networks, Inc. Technical Documentation.

[6] Mixed packet size throughput. `http://advanced.comms.agilent.com/n2x/docs/insight/2001-08/TestingTips/1MxdPktSzThroughput.pdf`, 2001. Agilent Technologies Technical Documentation.

[7] K. Salah, K. El-Badawi, and F. Haidari. Performance analysis and comparison of interrupt-handling schemes in gigabit networks. *Computer Communications*, 30(17):3425 – 3441, 2007. Special Issue Concurrent Multipath Transport.

[8] S. R. Srinivasan, J. W. Lee, E. Liu, M. Kester, H. Schulzrinne, V. Hilt, S. Seetharaman, and A. Khan. Netserv: dynamically deploying in-network services. In *ReArch '09: Proceedings of the 2009 workshop on Re-architecting the internet*, pages 37–42, New York, NY, USA, 2009. ACM.

[9] P. Tullmann, M. Hibler, and J. Lepreau. Janos: A Java-oriented OS for active network nodes. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, 19:501–510, 2001.

[10] D. J. Wetherall, J. V. Guttag, and D. L. Tennenhouse. ANTS: a toolkit for building and dynamically deploying network protocols. In *in IEEE OPENARCH*, 1998.

[11] Y. Yemini and S. D. Silva. Towards programmable networks. In *in IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, 1996.