TincVPN Optimization

Derek Chiang, Jasdeep Hundal, Jisun Jung

Abstract

We explored ways to improve the performance for tincVPN, a virtual private network (VPN) implementation. VPN's are typically used for creating a (usually private) network between computers connected by the public Internet. TincVPN is noted for being easy to configure, but is about 5 - 15 times slower at sending data between two machines that are otherwise linked together by a sufficiently a fast connection, including virtual servers running on the same physical host.

In this paper we first detail the design and architecture of tincVPN and how it relates to the observed performance issues. Then we propose two key optimizations, buffering the data packets that tincVPN sends and switching the mechanism by which tincVPN decides whether there is incoming data available to be read from the Linux system call *select* to the the system call *epoll*. Finally, we compare the performance profiles of the unoptimized and optimized versions of tincVPN and discuss areas for further improvement. Unfortunately, we have not seen the gains expected from our implemented optimizations because of tinc's administrative packet overhead.

Introduction

This project was initially motivated by the SuperCloud project at Cornell University, which has the goal of enabling users to flexibly shift the virtual servers (and other resources) that they are using across different cloud providers (for example, Amazon's Elastic Compute Cloud and Microsoft's Azure). SuperCloud initially used tincVPN as its solution for establishing a network between two different cloud providers. It was chosen because it is incredibly easy to configure, which allows for increased development speed, especially for rapid testing between different networking configurations for varying numbers of servers across the provider clouds. The drawback with using tincVPN is its performance. When testing between any two hosts, the tincVPN interface achieved about 20% of the data rate when compared to the standard host interface, whether it was physical or virtualized. Typically the performance of the former was only about 10% of the latter.

Though the creators of SuperCloud have shifted away from using a VPN based solution to set up networking between virtual servers, this project is still valuable because VPN's see widespread use despite their typical inefficiency. Even of most users have an emphasis on the "private" side of the private network a VPN provides and our work is on the "network" side, it is likely that any low level networking gains will preserve the privacy features while making the VPN experience more responsive and enjoyable.

In this project, we install and test tincVPN, analyzing it's architecture and performance profiles to identify likely optimizations. Then we implement chosen optimizations and observe if any significant performance gains are realized.

Related Work

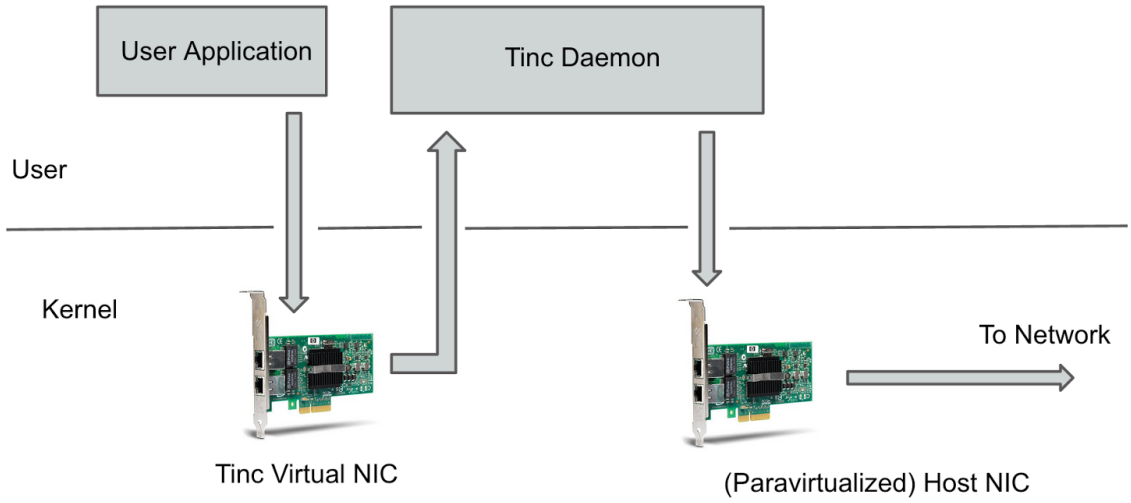
The obvious alternative to using tincVPN would be to use another VPN implementation that has better performance characteristics. Some implementations likely manage and send data packets more efficiently than tincVPN, which may be a worthwhile tradeoff even if they are harder to configure. One possible future direction for this project would be to benchmark competing VPN solutions and perhaps overhaul tincVPN's low level sending routines based on the best implementations seen. However, most Linux VPN's, including the popular OpenVPN, are implemented in user space and would eventually suffer the same context switch ceiling.

Another solution is to use a virtualized network switch instead of a VPN, and this was the option chosen by the Supercloud project that originally inspired this work. Using a virtual switch has significant advantages. A virtualized network switch is does not need to modify incoming data packets and the one chosen by the SuperCloud project, OpenVSwitch implements large parts of its data path as a kernel module, which reduces the amount of kernel switches needed to send data through the switch. Additionally, machines that send data to the switch do not need to install anything or modify their networking configuration in any special way. The disadvantage of virtual switches is that they are not as easily configured as VPN's, though advances in software-defined networking are closing the gap.

Background/Tinc Architecture

We begin with a discussion of the generic architecture of most Linux VPN solutions and move to a discussion of Tinc specific portions and compare its performance profile to a standard networking interface.

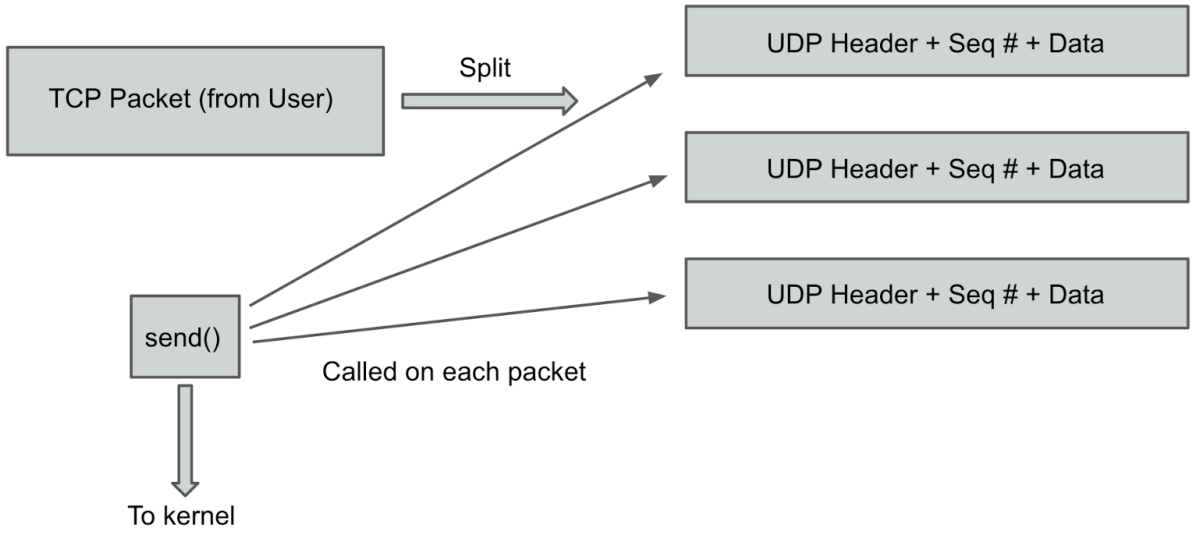
Linux VPN's are typically implemented such that they present themselves as a network interface to user applications that want VPN access. The diagram on the following page illustrates this setup and applies to OpenVPN and other implementations as much as it does to tinc.



The diagram shows one limitation of VPN's: as they are typically not implemented as kernel modules, sending a packet from the user application requires two system calls. One happens in the user application, and the other in the VPN daemon that directs the user's data to the correct host and interface for the virtual network IP address that the user specified by sending the right bits to an actual network interface card on the host. Different VPN implementations differ in how they manage the set of virtual IP's for the virtual network, how they make routing decisions to direct data to the appropriate hosts, and how they fragment and reassemble user data. However, the path that the user application's data flows through usually looks exactly like the above.

This indicates that one possible improvement for tincVPN could be to move significant bits of the tincVPN daemon to kernel space as a kernel module, thus eliminating one very expensive context switch. However, if these system calls were the bottleneck, we would notice that tincVPN would likely achieve close to 50% of the data rate of a standard interface, but this was not the case.

One specific design choice that was discovered was that tincVPN splits large data packets from user applications into UDP packets with a sequence number that are sized to the maximum transmission unit for the network path that the tincVPN packets take. It then transmits these packets through individual calls to *send*. This means that for every system call that a user application performs to send large packets tincVPN is performing several *send* system calls to forward data to the real network interface, likely adding a significant amount of overhead to the operation of the VPN. The following diagram illustrates this process.



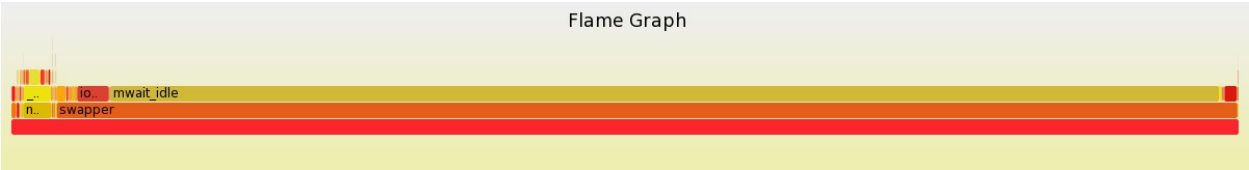
Analysis of Tinc Performance

We created a test setup to confirm that the send calls were consuming lots of time. The test setup consisted of two virtual machines running on a single host, both with tincVPN installed. The tool ‘netperf’ was used to generate a large amount of traffic as 16384 bytes packets from one of the virtual machines to the other.

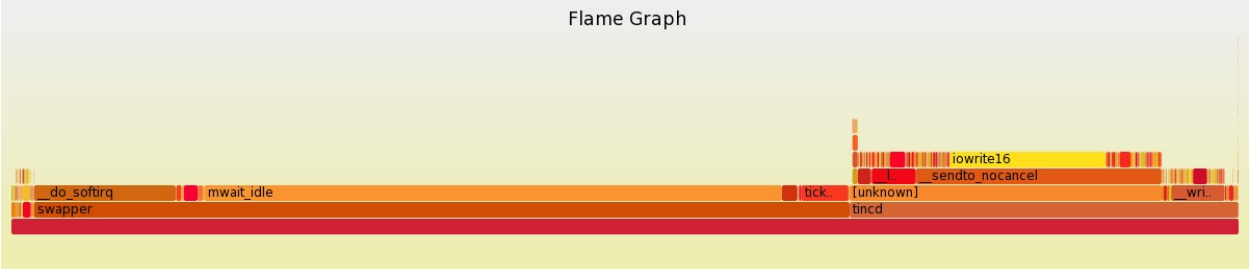
Netperf consistently achieved a data rate of about 1800 MBits/s between the standard networking interfaces and about 200 MBits/s between the tincVPN interfaces. Considering a single *send* call as one packet transfer, this reflects a packet rate of approximately 13,700 packets/s for the standard interface and approximately 15,625 packets/s for tincVPN. The latter is achieving a slightly higher rate for system calls, but since the packet size here matches the maximum transmission unit of 1600 bytes, the overall data rate is much lower.

The kernel profiling tool ‘perf’ was used to generate flame graphs of time spent in various kernel calls for the netperf tests for both the standard interface and the tincVPN interface. The results are seen below.

System Wide Kernel Calls w/o Tinc (using perf)



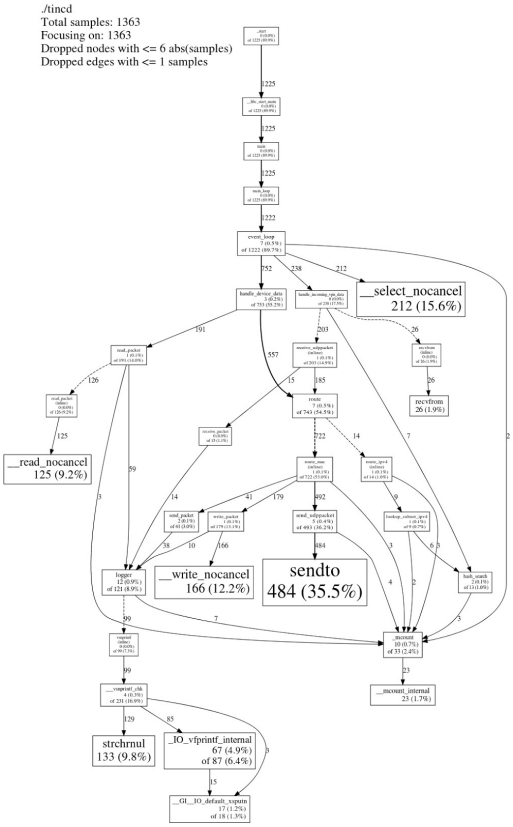
System Wide Kernel Calls w/ Tinc (using perf)



The flame graphs confirm the findings of the code review, showing that tincVPN spends far greater amounts of time in the *__sendto_nocancel* call that is called along the path of the userspace *send* system call. In fact, sending data with netperf through the standard interface barely spends time in any kernel function, instead idling the vast majority of the time.

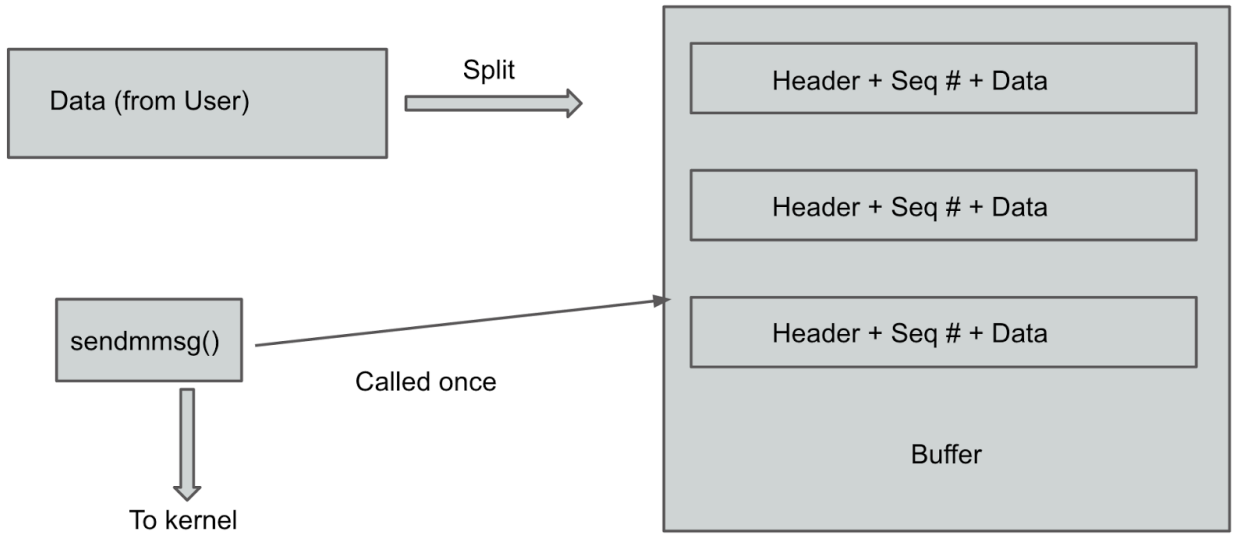
Further confirmation came through running a tincVPN daemon attached to Google's performance profiling tools (gperftools). This generates data by samples the process to see which system call it is in at any given point in time. Gperftools gives us the call graph below which shows the percentage of time that the tincVPN daemon spent in various systems calls. The *send/sendto* system call dominated, though the time spent in the *select* call was also significant, giving us another route for optimization.

Tinc Profiling (using gperftools)



Design and Implementation of Optimizations

From the analysis above, we first decided to attempt to reduce the number of *send* system calls. The solution is to buffer the packets generated by tincVPN, sending a filled buffer as part of one system call. We would like our layer for sending packets to reflect the diagram below, as opposed to the per-packet *send* send calls seen before.



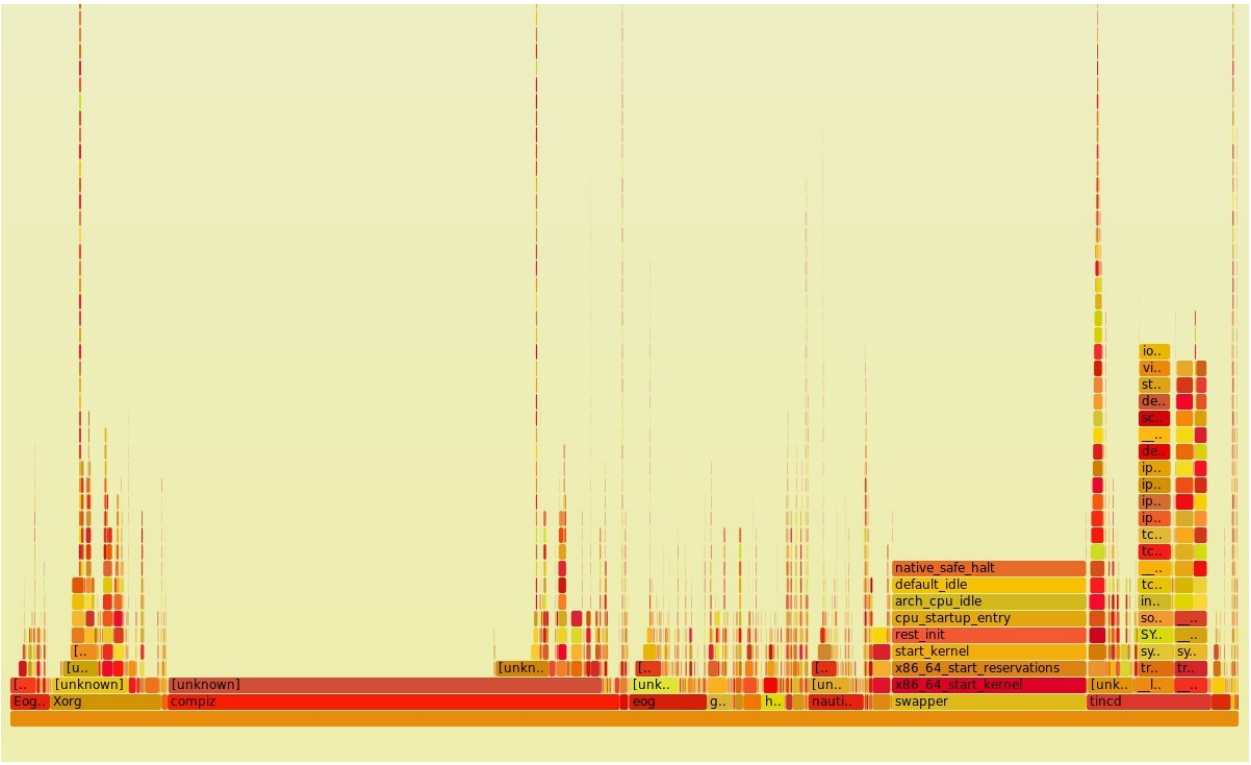
Newer versions of Linux include a *send_mmsg* system call that allows for exactly this implementation. The optimized tinc code takes a few new steps. At initialization it sets up space for a buffer of packets, allocating an array of *vpn_packet_t*, the same struct that tincVPN splits larger packets into. On an call to send one of these small packets, it is instead typically copied (via *memcpy*) to the next free space in the buffer. When the buffer is filled by the last packet added to it, a helper function is called to parse the individual packets in the buffer into a series of *msghdr* structs representing the individual packets to be sent. These are then added to a *mmsghdr* struct that is passed into the *send_mmsg* system call which actually sends the packets.

The other optimization done as a result of the call graph analysis was to switch from using the *select* system call to the *epoll* system call. This was more straightforward because their semantics are nearly the same. However, though *select* took a significant amount of time as indicated by the samples gathered by gperftools, the benefit of this switch is likely to be small since *epoll* is only slightly more efficient than *select*.

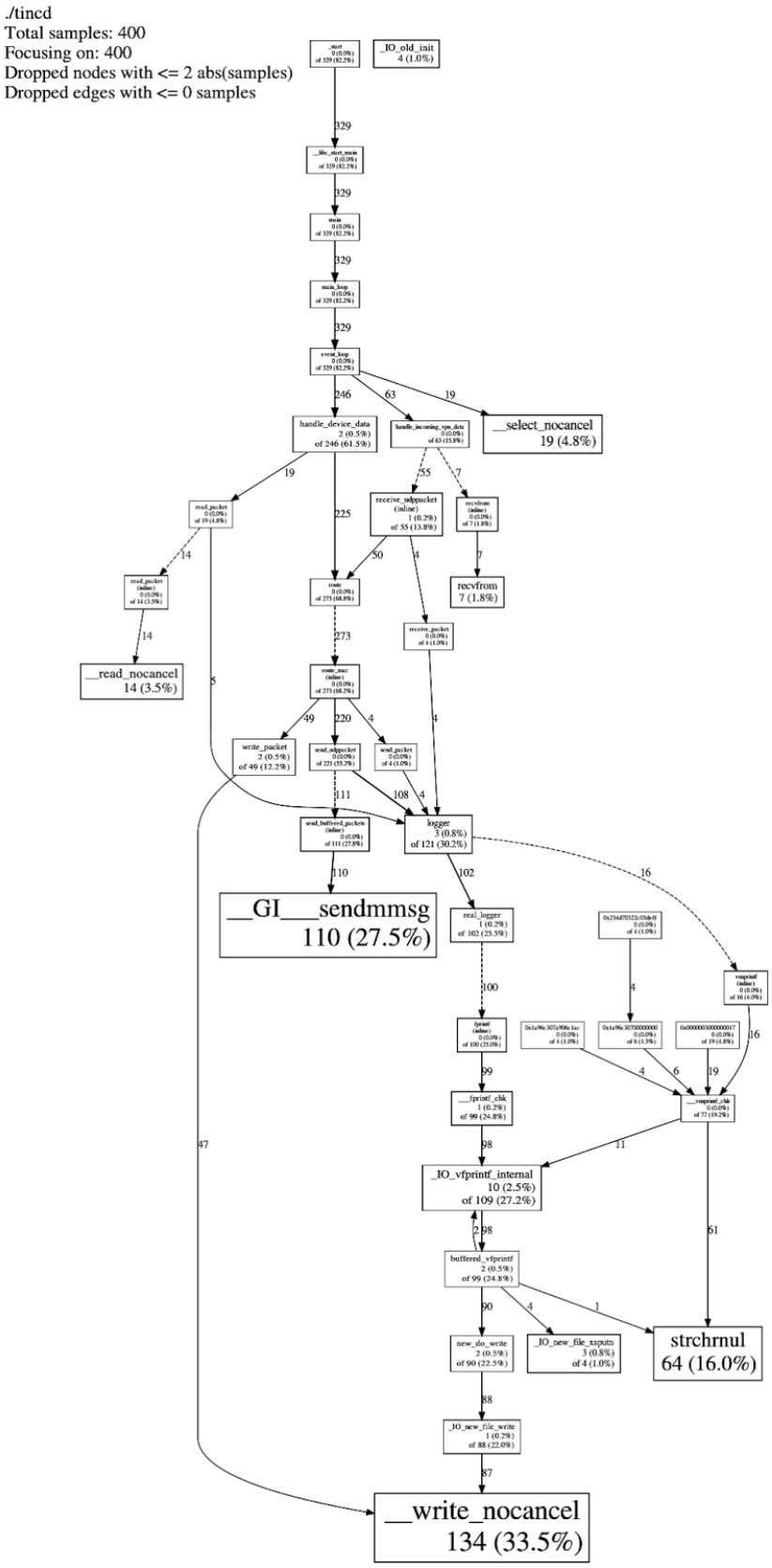
Evaluation

The following figures are the results from the optimized TincVPN implementation.

Kernel Flame Graph - Optimized TincVPN



System Call Graph - Optimized TincVPN



The flame graph and call graph show a slight shift in the performance profile, but it turned out that the performance was worse in the optimized code. Initially we used a very simple buffering mechanism where a buffer was flushed only on being filled, but this prevented tincVPN from sending through administrative packets used for establishing its routes and confirming receipt of chunks of data because these operations require a response. This was discovered through tincVPN's own logging as well as custom logging added where necessary. The final implementation included a coarse grained marking of packets as administrative or not, forcing a flush of the packet buffer whenever an administrative packet was sent. It turned out that the flushing happened far too often to make up for the overhead of copying the packets in memory.

Future Work

We have identified a few major areas left for improvement. The first is making benchmarking more automated. We have implemented a script to generate a flame graph, but we have to generate the call graph separately as gperftools cannot be run on a detached process. Combining these two would cut the testing process time in half. Additionally, the configuration of the VM's to use tincVPN, though simple, is still a manual process, making it hard to switch operating systems details like the level of hardware virtualization.

An easy test for justifying properly implemented buffering would be to increase the MTU available on the interfaces in the virtual machines used for testing. Though tricky to get right, switching from standard Ethernet frames to jumbo frames would significantly reduce the number of *send* calls be case of the 500% increase in possible packet size. Once this improvement is verified, a finer grained separation of tincVPN's administrative and data packets can be attempted to further reduce the number of overall send calls.

TincVPN runs in single event loop and does not exploit the parallelism possible on modern machines. An initial attempt was made to parallelize send calls, but it was discovered that Linux serializes send calls to the same interface, so no performance improvement was observed. There are other ways to use multiple cores and this should be explored.

Finally, writing pieces of TincVPN as a kernel module once current bottlenecks are resolved may allow it to come close to matching the performance of the standard network interface as it will be able to avoid the current minimum of two system calls on every packet send.

Conclusion

For this project, we familiarized ourselves with the design and architecture of tincVPN to reason about its performance issues. After studying these, we came up with several ways to improve performance and implemented some of them. Our optimizations were buffering the data packets to reduce the number of sends tincVPN invokes and changing the way it reads incoming data from using select to epoll. We then compared the performance evaluations of

tincVPN with and without these changes. Although we were unable to achieve significant performance improvement by implementing these optimizations, we have discussed future work that might result in greater performance gain.

Appendix

The OPTIMIZATION_README.md in the attached source details how to configure tincVPN across two machines and then the run the profiling tools.