Slick: A Framework for High Throughput Network Applications in the Kernel

Alex Gartrell

April 11, 2011

1 Introduction

With the increasing use of the Internet and Networked services, the ability to make such services perform better, specifically in terms of increasing throughputs and, by extension, the number of requests that can be handled, is more important than ever. This has been shown by the changes in architecture of such services from having many processes, to having many threads, and finally to having a single thread with a fast event loop. Still, such services are constrained by the maintenance of the process abstraction (i.e. the isolation of tasks from each other and the inner workings of the operating system), which imposes a great cost on every network transaction.

In order to avoid the overheads associated with maintaining the process abstraction, we shift the networked service into a kernel module, where it can be loaded directly into the kernel's execution path. We made this movement reasonably straight forward by providing a set of abstractions that mirror the libevent library for event based programming in a user process. Additionally, we provide a handful of utilities that would be otherwise unavailable in the kernel in contrast to userland, such as Thrift protocol parsers and libevent-style buffered sockets.

We have shown quantitatively that the kernel approach to networked services dominates the user process approach for small granularity sends and receives in terms of throughput with similar latencies. Additionally, we have provided abstractions that have made it possible to port user process networked services without completely rearchitecting the existing code base or being constrained by a framework that makes assumptions about the service.

Slick differs from prior work in that its main focus is making the Linux kernel a hospital place for which to write networked services without compromising on performance. This is different from other attempts which have involved writing a kernel from scratch for the purpose of networked services [3], or providing a more specialized caching layer for static content and requests [2] which is particularly powerful for certain workloads [4] but does not generalize to as many networked services as slick.

Our microbenchmarks for `send` and `recv` with small buffers show that there is certainly a much larger cost for user process network transactions than their kernel counterparts, as

the throughput for the kernel variant with small payloads (32 to 128 bytes) ranged from 30 to 60 percent higher. This was further corroborated by our larger scale bench mark that involved forwarding messages from many clients to many servers, where the throughput for small messages was also far better.

Providing a networked service from inside the kernel certainly has its drawbacks: application developers must deal with a different and kernel-specific set of primitives, crashes are fatal for the entire system, and many assumptions may not be forward compatible with future kernel releases. However, there are certainly advantages in terms of performance. Additionally, we believe that the abstractions offered by Slick minimize the pain of porting existing services to or writing new services for use in the Linux kernel. Thus, application developers who find themselves constrained by user process overheads with small workloads should consider slick a viable avenue through which to increase the performance of their service without putting forth a tremendous amount of effort.

2 Prior Work

The X-Kernel [3] specifies a top-to-bottom approach that allows programmers to specify protocols and have the kernel distribute messages to the appropriate processes based upon their protocol. Slick differs from this in that it rests upon a well established kernel, the Linux kernel, instead of a brand new one.

The Adaptive Fast Path Architecture [2] [4] provides a network interposition layer within the kernel that allows for the caching of static content in memory. This results in the ability to serve static content at a much higher rate. Slick differs from this in that it is meant to place as much of the application as desired in the kernel; shortly, AFPA could be reimplemented using Slick.

Goglin, Glück, and Primet [1] provided a special Network API for faster MPI-type message passing for use with filesystems, in which they were able to take advantage of kernel space to offer fewer copies, which improved latencies. This paper differs from their work in that it addresses the improvements of using the standard API in the kernel as compared to in user space. The work done by Gogle, Glück, and Primet could be used to further improve Slick, which is one more argument for handling networked services in kernel.

3 Approach

The slick architecture as discussed here has three layers: a socket notification layer called Klibevent, a socket buffering layer called BufferedSockets, and a Thrift parsing layer called ThriftSockets. The Klibevent varies greatly from the traditional approach to the problem, a poll, select, epoll, etc. loop, but the other layers are more or less equivalent to their user process equivalents, which is significant because it means that they can be ported to standard programs for more convenient testing of Slick-based utilities.

3.1 Klibevent

Klibevent works by interposing on regular network event functionality. In order to support the wide number of network protocols currently available to developers, the Linux Kernel utilizes something similar to object oriented polymorphism, which is that they pass around structs filled with function pointers. This allows more generic code (like that used in the networking system calls) to operate more generically.

Listing 1: An example of "object orientedness" in the Linux Kernel

```
566     return sock->ops->sendmsg(iocb, sock, msg, size);
```

This approach is taken throughout the Linux Kernel's networking code, which is of special interest to us for its socket activity notification mechanisms. These callbacks within softirqs after events like the arrival of data or a change in the state of a TCP connection. By overwriting these callbacks in an intelligent way, we are able to gather this information with next to no additional overhead.

Listing 2: Callbacks within internal socket datastructure

```
319 void (*sk_state_change)(struct sock *sk);
320 void (*sk_data_ready)(struct sock *sk, int bytes);
321 void (*sk_write_space)(struct sock *sk);
322 void (*sk_error_report)(struct sock *sk);
323 int  (*sk_backlog_rcv)(struct sock *sk,
324                        struct sk_buff *skb);
325 void (*sk_destruct)(struct sock *sk
```

When we are notified through this mechanism, we add an event to a workqueue, a queue that is used to serialize jobs and distribute them among one or more worker threads. In doing so, we do some bookkeeping to ensure that we aren't re-entering a callback for any single Socket as a convenience to the programmer, who would likely otherwise have to handle this chore herself.

3.2 BufferedSockets and ThriftSockets

BufferedSockets rest on top of the Klibevent sockets and serve two purposes: to aggregate received data so that it can be "peeked" for the purposes of parsing, and to exist as an overflow for data that cannot be sent in full so that we can assure atomicity for single-message sends. It does this by providing its own versions of read and write, as well as versions for peeking and atomically sending.

Listing 3: API of BufferedSocket abstraction

```
20 int BufferedSocket_peek(BufferedSocket *bs, char **buffer);
21 int BufferedSocket_discard(BufferedSocket *bs, int len);
22 int BufferedSocket_read(BufferedSocket *bs, char *buff, int max_len);
23 int BufferedSocket_write(BufferedSocket *bs, char *buff, int max_len);
24 int BufferedSocket_write_all(BufferedSocket *bs, char *buff, int len);
```

`ThriftSockets` further extend `BufferedSockets` by allowing a user to send and recv `ThriftMessages`. This is very convenient for the programmer, who no longer needs to worry about constructing and deconstructing the messages herself.

Listing 4: API of ThriftSocket abstraction

```
75 int ThriftSocket_next(ThriftSocket *ts, ThriftMessage *tm);
76 int ThriftSocket_discard(ThriftSocket *ts);
77 int ThriftSocket_write(ThriftSocket *ts, ThriftMessage *tm);
78 int ThriftSocket_forward(ThriftSocket *from, ThriftSocket *to);
```

4 Evaluation

Here, we have evaluated Slick using both an echo module, a microbenchmark designed to test `send` and `recv` throughput at various buffer granularities, and a forwarder module, a more complete benchmark involving a toy protocol that allowed us to test Slick under more realistic loads involving many-to-many communication. In both instances, we tried to minimize the amount of work done that would be similar in cost in both environments (e.g. string parsing).

4.1 Application

Our echo protocol is dead simple. The client connects to the server, sends the size of buffer, N to use, and then saturates the socket. The server simply receives N bytes, and sends them back.

Our forwarder protocol is slightly more complex, and is build upon Apache Thrift, a commonly used software framework for remote procedure invocation. We have implemented the asyncronous call `msg`, which has two fields: an integer field representing the intended destination server, and a string field that can be arbitrarily sized to represent different payloads. Additionally, there are a number of calls that allow us to easily initialize the forwarder in various ways that were not included in the experiment.

Listing 5: Forwarder protocol used for benchmarking

```
service Forwarder {
    oneway void msg(1: i32 key, 2: string payload)
}
```

4.2 Reference Implementations

The user process reference implementation of the echo server is as straightforward as the protocol suggests. It simply accepts connections, receives a buffer size, and receives and sends as long as the connection stays alive. It is single process.

Listing 6: User process implementation of echo server

```
while((fd = accept(listener, NULL, NULL)) >= 0) {
```

```
        recv(fd, &buff_size, sizeof(buff_size), 0);
        buff_size = ntohl(buff_size);

        while((amt = recv_all(fd, buff, buff_size, 0)) > 0)
            send_all(fd, buff, amt, 0);

        close(fd); fd = -1;
    }
```

The user process reference implementation of the forwarder server is slightly more involved. Due to the requirement that the forwarder support multiple clients, we implemented it using libevent, which provided an interface similar to that of klibevent and is a pretty widely accepted mechanism for implementing this type of server, as it allows us to avoid overheads associated with thread creation and scheduling at the cost of a slightly more complex program model. Other code was reused as much as possible, including modules like the protocol parser, to give us as fair of a comparison between Slick and the user process implementation as possible.

4.3 Hardware

There were two machines in the experiment, `fawn-desktop2` and `slick0`.

Table 1: Configuration of test machines

Machine	CPU	Memory	Network
`fawn-desktop2`	Intel Core i7 @ 4 x 2.80 GHz	8 GB @ 1066 MHz	1 Gbps
`slick0`	Intel Atom @ 1 x 1.67 GHz	2 GB @ 667 MH	1 Gbps

Both machines were running the 64-bit server edition of Ubuntu Linux version 10.10. The linux kernel run by each was version 2.6.35-22.

Additionally, both machines were connected to eachother on the same gigabit network switch.

Load generation and performance measurement were performed by `fawn-desktop2`, while the actual service was provided by `slick0`.

5 Data

Figure 1: Comparison of kernel and user echo services

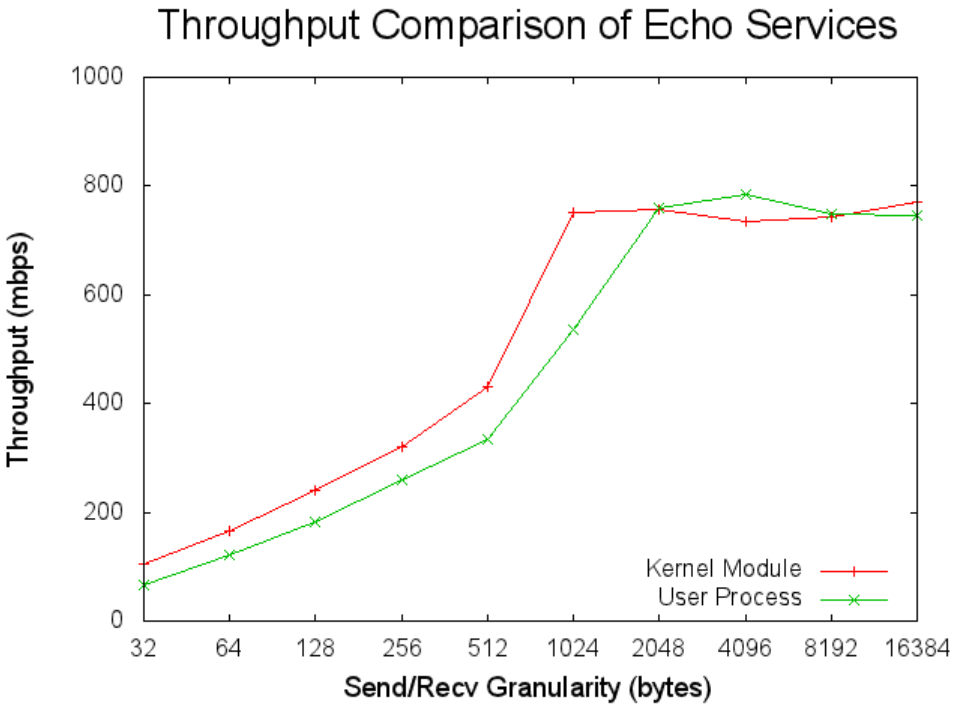

Table 2: Throughput of kernel and user echo services

Buffer Size	**User Process**		**Kernel Module**	
	Throughput	Latency	Throughput	Latency
32 bytes	65.57 mbps	37 ms	103.66 mbps	31 ms
64 bytes	120.37 mbps	32 ms	166.24 mbps	28 ms
128 bytes	183.59 mbps	28 ms	239.50 mbps	27 ms
256 bytes	258.51 mbps	31 ms	319.28 mbps	37 ms
512 bytes	334.72 mbps	36 ms	431.90 mbps	37 ms
1024 bytes	536.86 mbps	33 ms	751.85 mbps	15 ms
2048 bytes	759.93 mbps	13 ms	757.17 mbps	15 ms
4096 bytes	784.33 mbps	17 ms	734.33 mbps	21 ms
8192 bytes	747.26 mbps	16 ms	741.97 mbps	17 ms
16384 bytes	746.63 mbps	17 ms	772.09 mbps	12 ms

Figure 2: Comparison of kernel and user process forwarders

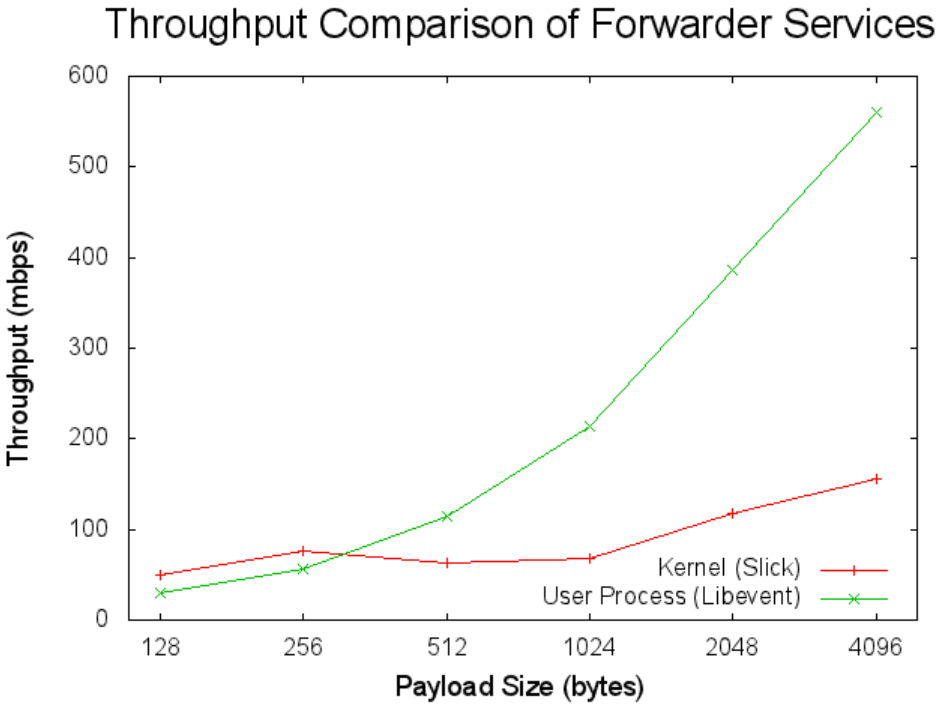

Table 3: Throughput of kernel and user process forwarders

Payload Size	Kernel (slick)	User Process (libevent)
128 bytes	49.15 mbps	30.51 mbps
256 bytes	75.53 mbps	56.94 mbps
512 bytes	63.46 mbps	114.86 mbps
1024 bytes	67.79 mbps	213.16 mbps
2048 bytes	118.25 mbps	386.63 mbps
4096 bytes	155.79 mbps	560.33 mbps

6 Discussion

The echo service tests show that the kernel approach dominates the user process approach when the granularity of sends and receives are small. This is almost certainly due to the additional overheads of invoking `send` and `recv` from a process as compared to from inside of the kernel. This shows that there is certainly merit to the idea of in-kernel networked services for certain workloads, if only due to the saved expense.

The forwarder service tests are more ambiguous. As expected, the Slick router dominates the User Process router for small payloads, where the sends and recvs must be small.

However, this advantage disappears as the performance of the Slick router actually degrades as the payload size increases. This is likely due to a bug in the implementation, as there is no reason that the router should be slower than the user process equivalent.

7 Summary

With this work, we have not yet shown that moving a networked service into the kernel is universally advantageous. Certainly, there are situations where the extra protection provided by the kernel to user processes is either mandatory or it is simply not worth the software engineering effort required to make code "kernel-proof." Slick is not the correct approach for these situations, regardless of the performance gains it offers.

Additionally, we have failed to show that Slick is advantageous across all networking workloads. For any real conclusions to be drawn regarding the utility of kernel-level networked services, many aspects of the framework would need to be revisited and many tests would need to be rerun. However, the results of the tests with small payloads do show that Slick is, at the very least, promising, and deserves further investigation. Currently, work on these issues is ongoing.

References

[1] B. Goglin, O. Gluck, and P. Vicat-Blanc Primet. An efficient network api for in-kernel applications in clusters. In *Cluster Computing, 2005. IEEE International*, pages 1 –10, 2005.

[2] E. C. Hu, P. A. Joubert, R. B. King, J. D. LaVoie, and J. M. Tracey. Adaptive fast path architecture. *IBM Journal of Research and Development*, 45(2):191 –206, 2001.

[3] N.C. Hutchinson and L.L. Peterson. The x-kernel: an architecture for implementing network protocols. *Software Engineering, IEEE Transactions on*, 17(1):64 –76, January 1991.

[4] Philippe Joubert, Robert B. King, Rich Neves, Mark Russinovich, John M. Tracey, Robert B. King , Mark Russinovich , and John M. Tracey . High-performance memory-based web servers: Kernel and user-space performance.