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

Alex Gartrell
Advised by David Andersen

May 1, 2011

Abstract

With the increasing use of the Internet and networked services, the ability to make such services perform better, specifically in terms of increasing throughput and, by extension, the number of requests that can be handled, is more important than ever. Despite many interface and architecture improvements, 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. Slick tackles this problem by providing a convenient interface for providing these services in the kernel. In this work we explore the creation of this framework and the performance implications of its use.

Contents

1. Background

1.1 Introduction

With the increasing use of the Internet and Networked services, the ability to make such services perform better, specifically in terms of increasing throughput 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 user processes, 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 micro-benchmarks 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.

1.2 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 file systems, 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.

2. Approach

Today, many high performance servers are evented, which means that they utilize an event notification mechanism to know when to execute handlers that interact with sockets. This is best explained in contrast to forking and threaded servers, which spawn new operating system contexts which are dedicated to handling a single connection. The primary difference is that threaded servers tend to use blocking socket IO (i.e. a call to `recv` will suspend the thread until a message has arrived) while evented servers tend to use non-blocking IO and instead rely upon the event notification mechanism to "remind" them to call `recv` on a particular server when a packet has arrived. Evented servers may invoke polling functions (e.g. `poll`, `select`, `kqueue`, `epoll`) directly in their own event loop or may rely instead upon frameworks that work better across platforms such as libevent.

Listing 2.1: Simple evented server example

```
void sock_read(int fd, short event, void *arg)
{
    char buf[255];
    int len;

    event_add((struct event *) arg, NULL);

    len = read(fd, buf, sizeof(buf));
    do_something(buf, len);
}

int main (int argc, char **argv)
{
    struct event evsock;
    int socket = create_socket();

    event_init();
    event_set(&evsock, socket, EV_READ, sock_read,
        &evsock);

    event_add(&evsock, NULL);
    event_dispatch();
    return 0;
}
```

The advantage of using evented servers is that they are less resource intensive. When a connection is inactive, it uses very few resources outside of the socket and its state, while an inactive threaded connection requires all of the memory required to maintain all of the thread state, both within the user process and inside of the kernel. Additionally, the scheduling of connection handling is done directly by the process in a single or limited number of threads, which means that more intelligent decisions can be made about when exactly something should be run and the scheduling can take place cheaply in a single context that need not be switched.

The primary disadvantage of using evented servers is that they can be very difficult to engineer. A surprisingly large amount of state is stored implicitly by threads, including obvious examples like state stored to the stack, but also less obvious ones like the location of the instruction pointer. Essentially, the thread's context acts as a complex state machine, and managing all of that complexity can be a challenge, especially when it comes to particularly stateful things like parsing.

Ultimately, we made the decision to use an evented server approach for two reasons. First, the cost of memory within the kernel is very high, as the address space of the kernel is limited. Second, we believe that if someone is willing to go through the pains of a kernel implementation, the additional pain of writing evented handlers is minor, and the cost of threaded handlers is unacceptable. We did, however, make an effort to minimize the pain of writing an evented handler by providing both buffered IO and message parsing.

2.1 Handling Socket Events with Klibevent

Klibevent takes advantage of the underlying structure of the Linux kernel's network stack as well as a particularly useful deferred work construct called Work Queues to provide an interface to a developer that is very similar to the interface provided by libevent. This allows developers to program their in kernel networked applications as they would user process networked applications, which can help to simplify both porting existing applications and creating new ones.

In order to support a wide variety of network protocols, the Linux kernel allows for protocols to be dynamically loaded at run time. To make this dynamic loading possible, the kernel defines an interface that protocols must adhere to. Each protocol must provide a method for tasks like sending a message, accepting a connection, closing a connection, etc. Later, when a

send is invoked on that socket, it will perform some generic functionality – things that must be taken care of regardless of which protocol is being employed – and then fall through to the protocol-specific functionality specified when the protocol is registered. In software engineering, the use of an interface to specify such functionality is commonly referred to as the strategy pattern.

Listing 2.2: The protocol registration function in /net/core/sock.c

```
2356  int proto_register(struct proto *prot, int alloc_slab)
```

Listing 2.3: Initialization of the inet protocol family in /net/ipv4/af_inet.c

```
1632  rc = proto_register(&tcp_prot, 1);
1633  if (rc)
1634          goto out_free_reserved_ports;
1635
1636  rc = proto_register(&udp_prot, 1);
1637  if (rc)
1638          goto out_unregister_tcp_proto;
1639
1640  rc = proto_register(&raw_prot, 1);
1641  if (rc)
1642          goto out_unregister_udp_proto;
1643
1644  /*
1645   *      Tell SOCKET that we are alive...
1646   */
1647
1648  (void)sock_register(&inet_family_ops);
```

The use of this pattern is very obvious in an object-oriented language supporting polymorphism, but its use is less straightforward in the C programming language. To get around the lack of polymorphic objects in C, the kernel developers employ structs of function pointers. By instantiating such a struct with pointers to the desired functions, it is possible to approximate polymorphism, albeit without the implicit access to object-specific state available in C++ and Java.

Listing 2.4: The TCP protocol function table in /net/ipv4/tcp_ipv4.c

```
2560  struct proto tcp_prot = {
2561          .name                   = "TCP",
2562          .owner                  = THIS_MODULE,
2563          .close                  = tcp_close,
2564          .connect                = tcp_v4_connect,
2565          .disconnect             = tcp_disconnect,
2566          .accept                 = inet_csk_accept,
```

```
2567            .ioctl                  = tcp_ioctl,
2568            .init                   = tcp_v4_init_sock,
2569            .destroy                = tcp_v4_destroy_sock,
2570            .shutdown               = tcp_shutdown,
2571            .setsockopt             = tcp_setsockopt,
2572            .getsockopt             = tcp_getsockopt,
2573            .recvmsg                = tcp_recvmsg,
2574            .sendmsg                = tcp_sendmsg,
2575            .sendpage               = tcp_sendpage,
2576            .backlog_rcv            = tcp_v4_do_rcv,
2577            /* Snip */
2578    };
```

Listing 2.5: Strategy for creating sockets of a particular protocol family in /include/linux/net.h

```
212  struct net_proto_family {
213          int             family;
214          int             (*create)(struct net *net, struct
                 socket *sock,
215                                   int protocol, int kern);
216          struct module   *owner;
217  };
```

The result of all of this registering is that when a user invokes the socket system call to create a TCP IPv4 socket, the core network code creates a blank, generic socket object, which is then passed on to `inet_create` along with the desired protocol – in this case, TCP. `inet_create` then looks up the correct proto struct and uses it to help populate the socket's `sock` substructure.

Listing 2.6: Invoking `socket` to create a TCP IPv4 Socket

```
int fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

Additionally, there are a handful of callbacks that are invoked by the kernel upon specific events upon connection state changes, the creation of write space, and the availability of new data.

Listing 2.7: Callbacks within internal socket data structure

```
void (*sk_state_change)(struct sock *sk);
void (*sk_data_ready)(struct sock *sk, int bytes);
void (*sk_write_space)(struct sock *sk);
void (*sk_error_report)(struct sock *sk);
int  (*sk_backlog_rcv)(struct sock *sk,
                       struct sk_buff *skb);
void (*sk_destruct)(struct sock *sk
```

These callbacks are of special interest to us, as they provide an alert mechanism for all of the relevant actions that should elicit a response from our framework. Further, due to the quirks of implementing the strategy pattern in C, these callbacks are stored in mutable data structures associated with each socket. The net result is that we are able to overwrite these callbacks to be alerted immediately of these events without much, if any, additional overhead.

Listing 2.8: Overwriting the event callbacks of the sock structure in klibevent.c

```
83  void overwrite_callbacks(struct sock *sk, void *data) {
84      if(unlikely(standard_callbacks_not_initialized)) {
85          standard_callbacks_not_initialized = false;
86          klibevent_work_queue =
                create_singlethread_workqueue("klibevent");
87          INIT_WORK(&klibevent_utility_work,
                klibevent_utility_callback);
88          standard_sk_state_change = sk->sk_state_change;
89          standard_sk_data_ready = sk->sk_data_ready;
90          standard_sk_write_space = sk->sk_write_space;
91      }
92      sk->sk_user_data = data;
93      sk->sk_state_change = over_sk_state_change;
94      sk->sk_data_ready = over_sk_data_ready;
95      sk->sk_write_space = over_sk_write_space;
96  }
```

These events are invoked from a softirq context, which essentially means that any blocking operation (i.e. any operation that is not guaranteed to be completed immediately) could result in a crash or deadlock. As many high level socket functions are blocking, we ultimately decided that it was better to defer the actual Klibevent callbacks until they could be run in a kernel process context, which provides support for blocking operations at the cost of additional context switching latency. To do this, we took advantage of the Work Queue construct provided by the kernel, which allows you to invoke a work callback after the preceding jobs have been completed. Coupling work queues with the callback mechanisms allows us to present an interface to the user that is very similar to libevent.

Listing 2.9: Klibevent interface

```
18  typedef void (*notify_func_t)(struct socket *, void *,
        int);
19
20  int register_sock(struct socket *, notify_func_t, void *);
21  void unregister_sock(struct socket *);
```

2.2 Buffering IO with BufferedSockets

Generally, a program interacting with the network must be written with the expectation that network reads will result in short counts – a `recv` for 10 bytes can return any number of bytes between 1 and 10. This is a necessary evil, as kernel buffers may be more limited than user process buffers, and it is also generally better to receive the bytes that are available than it is to wait, possibly forever, for the rest of the N bytes you have requested.

Listing 2.10: Common pattern for fetching a message from the network

```
char buffer[expected];
while(received < expected) {
    int amt = recv(sock, buffer + received, expected -
        received);
    if(amt <= 0) return received;
    received += amt;
}
```

Obviously, this will not work in a non-blocking evented environment, where we cannot expect `recv` to make progress upon every call and instantaneously, and we must share the stack space among several connections, so at the very least, we must relegate the buffer to its own heap space that is somehow associated with the socket in question.

Additionally, there is the question of what to do when a non-blocking send fails for lack of space, either due to saturating the TCP window or the network link. One option is to reschedule the `send` for when there is enough space to carry it out (i.e. after a "write space" event has occurred), but that can lead to complications as far as buffer ownership is concerned, as the calling party is likely eager to reclaim the space for their own use. There is also a need to send messages atomically, as `send`, like `recv`, can fail with a short count, having sent a fraction of what was asked of it. To get around both of these problems, we also buffer writes. This allows us to quickly empty the buffer upon a "write space" event, without having to do the complicated scheduling required for the alternative solution.

With these considerations and implementation decisions in mind, we decided to fully encapsulate the socket within a BufferedSocket object. This gives us more tasteful primitives for atomic sends and peeking reads, which allow us to hide some of the stateful nastiness from the actual application handlers. These handlers can now simply "peek" at the socket to see if the entire message has arrived, while the `recv` and store to the buffer happens in the background, and can `write_all` without worrying that only part of the message will be sent.

Listing 2.11: 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);
```

2.3 Parsing Binary Protocols with ThriftSockets

One of the biggest problems with placing networked services in the kernel is the increased danger to system crashes and serious security vulnerabilities. Further, one of the biggest sources of vulnerabilities appears in protocol parsing. For these reasons, we decided to provide a convenient mechanism for protocol parsing to allow the application developer to work with higher level primitives.

The protocol we chose to serve as an example was Apache Thrift's binary protocol. Having been initially developed at Facebook and then adopted by the Apache foundation, we decided that the Thrift protocol was sufficiently popular to be representative of actual workloads. We went with the binary variant, because we assumed that it was more likely to be used when high performance is absolutely necessary. We also considered Google Protocol Buffers, but Thrift happened to be more convenient, and the protocols are similar enough in this context that they are more or less interchangeable for the purposes of performance evaluation.

We provided an interface very similar to that of BufferedSockets, allowing the user to peek and later discard a single message. Additionally, writes and forwards happen with full-message granularity. This eliminates most of the nastiness caused by interacting with the network.

Listing 2.12: 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);
```

This interface is actually much different from the traditional Thrift interface, which relies upon autogenerated code that encapsulates the data and

makes the function calls directly. We decided to go with our stripped down interface because we believed it would be non-trivial to implement the code generation for the kernel module environment. Additionally, we wanted to retain some flexibility, and the traditional Thrift interfaces requires you to act immediately when provided a message, while our interface allows the application to leave it in the queue as long as it would like. Finally, our interface requires far fewer copies, which provides a performance benefit.

3. Evaluation

Here, we have evaluated Slick using both an echo module, a micro-benchmark 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).

3.1 Hardware

We used three machines in our experiments: `fawn-desktop2`, `fawn-desktop3`, and `slick0`.

Table 3.1: Configuration of test machines

Machine	CPU	Memory	Network
`fawn-desktop2`	Intel Core i7 @ 4 x 2.80 GHz	8 GB @ 1066 MHz	1 Gbps
`fawn-desktop3`	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 MHz	1 Gbps

All three 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. They were connected to each other on the same gigabit network switch, which was otherwise idle during our experiments.

Load generation was done by `fawn-desktop3`, performance measurement were done by `fawn-desktop2`, while the actual services were provided by `slick0`.

3.2 Echo Micro-benchmark

3.2.1 Application

We first had to show that Slick actually performed better than the user process equivalent under ideal conditions. To do that, we created a simple benchmark that was purely an echo. The client connects to the server and every byte sent to the server is then echoed back to the client. We varied the test across various buffer granularities to see how the payload size affected the throughput. This allowed us to measure throughput and also verify that the latencies were roughly the same.

3.2.2 Implementation

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

Listing 3.1: 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 kernel implementation does exactly the same thing. Upon a socket listen event, it opens the connection and does blocking IO on it in the same fashion until the socket closes. For the most part, it does not rely upon the facilities provided by Klibevent for handling the sends and receives.

3.2.3 Results

Figure 3.1: Throughput comparison of echo services

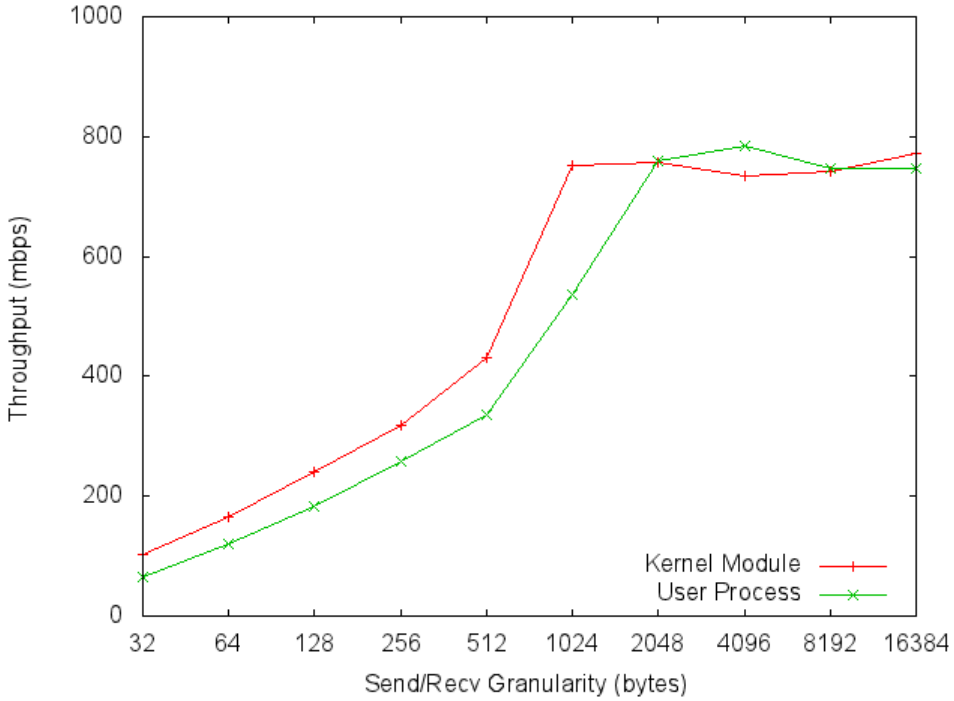

Table 3.2: Throughput and latency of echo services

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

3.2.4 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. The results above show that there is at least some merit to moving some networked services into the kernel.

3.3 Thrift Forwarder Benchmark

3.3.1 Application

Our forwarder benchmark uses Apache Thrift to describe a simple, easily forwarded protocol. Every message has two fields, an integer field representing the intended destination server, and a string field that can be sized to represent different payload sizes. Though that was the only aspect we tested, we also used the Thrift protocol to initialize the tests (connecting the forwarder to backends) and to finalize the tests (closing down those connections).

Listing 3.2: Forwarder protocol used for benchmarking

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

To generate load for the tests, we ran eight processes which connected to the forwarded and sent prerecorded Thrift messages evenly distributed among all of the destinations. On the destination server, we ran 20 instances of ttcp, and aggregated the throughput data after the test.

3.3.2 User Process Implementation

The user process reference implementation of the forwarder server was slightly more involved than that of the user process echo server. 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 widely used mechanism for implementing evented servers. We reused the parsing and buffering code as much as possible, though we placed less of an emphasis on making it robust. Otherwise, we attempted to keep the architectures as similar as possible to provide a fair comparison between the two.

3.3.3 Results

Figure 3.2: Throughput comparison of kernel and user process forwarders

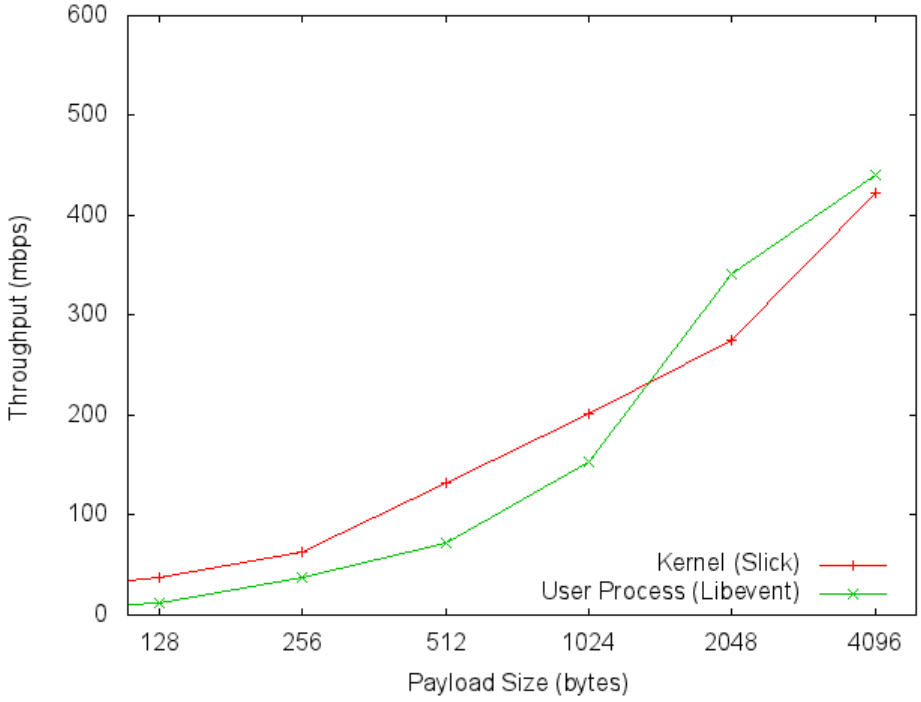

Table 3.3: Throughput of kernel and user process forwarders across two runs

Payload Size	Kernel (slick)		User Process (libevent)	
32 bytes	15.40 mbps	15.20 mbps	6.79 mbps	6.22 mbps
64 bytes	22.88 mbps	22.80 mbps	7.21 mbps	7.21 mbps
128 byes	37.00 mbps	37.00 mbps	11.47 mbps	9.04 mbps
256 bytes	63.54 mbps	63.20 mbps	36.91 mbps	36.31 mbps
512 bytes	132.60 mbps	168.60 mbps	71.45 mbps	70.32 mbps
1024 bytes	201.20 mbps	199.60 mbps	153.17 mbps	197.94 mbps
2048 bytes	275.73 mbps	414.00 mbps	341.63 mbps	296.78 mbps
4096 bytes	423.23 mbps	418.15 mbps	441.10 mbps	367.87 mbps

3.3.4 Discussion

The forwarder service tests show that the Slick implementation again shines when payload sizes are relatively small. Beyond payloads of 512 bytes, the throughput numbers become more erratic between runs and it becomes harder to make a compelling case for Slick. Still, we believe that the smaller payloads represent a meaningful set of workloads and that our results are therefor beneficial. Additionally, we believe that more work on Slick would likely result in better numbers, as there are surely a slew of performance issues that can be beaten out of the system over time.

4. Conclusions

4.1 Evaluating The Kernel Approach

There are obvious benefits to using slick from a performance perspective. Our experiments have shown that placing the service in the kernel can provide a real benefit in terms of throughput. The simple `recv` and `send` benchmark as well as the more complicated and representative Thrift forwarding benchmark show that such applications benefit from avoiding user process overheads.

However, this must be weighed against the constraints unique to kernel space, which include more severe failure modes, the potential to tightly couple code to specific kernel versions, and the lack of support for many user process libraries. Some of these issues can be overcome, both by enforcing modular design and solid engineering practices, but much of the traditional library code would simply need to be written. Ultimately, the Slick approach should only be taken when performance is of paramount importance and all other avenues have been explored. Under such circumstances though, our experiments have shown that this approach is beneficial.

4.2 Future Work

In the future, it would be useful to evaluate Slick with a wider variety of workloads, including different protocols and different traffic patterns. Specifically, it would be interested to create a reverse HTTP proxy with Slick and see if it can outperform industry standards like Apache and Nginx. It would also be interesting to test Slick in the context of a larger system across a longer period of time to investigate its impact on failures. Finally, it would be interesting to investigate its use by others to see what type of impact it actually has on the development process.

Bibliography

[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.