ABSTRACT
This paper demonstrates the performance gain of MIGSOCK over Zap, an user-level approach to network process checkpointing and restarting. We consider two scenarios where process checkpointing and restarting could take place. In the first scenario, where processes are aggregated in one Process Domain (pod), it takes Netfilter three times as much time as MIGSOCK to handle translation and forwarding of 10,000 packets. In the multiple-pod scenario, modeling multiple-service support, assuming the time to deliver a pod is constant, it takes about 10 pods for Netfilter to break even with MIGSOCK. In other words, MIGSOCK only suffers a performance hit in checkpointing 10 pods and beyond. We argue that this scenario can be avoided by load balancing services with a threshold of 10 per server. Based on empirical analysis, we conclude that MIGSOCK is an ideal solution to network process checkpointing in comparison to Zap.

General Terms
Process checkpointing

Keywords
Process checkpointing, MIGSOCK, Zap, Netfilter, process migration

1. INTRODUCTION
Process checkpointing is defined as an act of saving a process’s states and restart it at some later time. A successful process checkpointing mechanism requires the same process to be restarted at the point where it left off upon checkpointing.

There are numerous benefits to process checkpointing. For instance, it promotes transaction rollback of a system. By checkpointing important processes of a server periodically, the same processes can be restarted upon the server’s failure. Transaction rollback is automatically guaranteed since the last checkpointed image before the server’s failure signifies the last successful transaction. By restarting this image, transaction is automatically rolled back. Another important benefit of process checkpointing is server-load balancing where a heavily loaded server can checkpoint CPU-intensive tasks and offload them to some less loaded servers based on some sort of load-balancing mechanism. Other benefits include increase of data accessibility and ease of system administration.

The current difficulty as discussed in [1] is that current process checkpointing mechanisms are local to particular systems. These mechanisms have some sort of dependence on system resources, for example. Zap[3] attempted to address this kind of problem by creating a virtualization layer on top of the operating system to remove dependencies that processes have. The idea was sound; they did not, however, successfully address translation of resources for resources that a system does not have. In that instance, it is not so clear whether the solution to resource dependencies is achieved.

In addition, current process checkpointing ignores socket checkpointing. Because of the wide spread of network applications, programs are no longer stand-alone entities. Many programs rely on communication with servers to keep themselves updated. The client/server model makes it imperative to extend current process checkpointing to network process checkpointing. Successful checkpointing of a network process requires the establishment of the previous connection and its states upon restart.

There are many approaches to network process checkpointing that ranging from network layer all the way up to the application layer. Many existing network process checkpointing mechanisms provide host mobility by doing some sort of address translation across the network stack layers. However, these approaches introduce the need of a proxy to take care of the current connection during checkpointing and migration. A proxy is placed as a care-of for the checkpointed process[4]. Data on the current connection is buffered at the proxy. Some sort of mechanism to stall the connection to the remote process at the proxy prevents the remote process from disconnecting from the migrating process due to the long delay of receiving a response. Upon restarting, the data buffered at the proxy is then forwarded to the mi-
grated process. Existing connection is taken care of to reflect changes of the migrated process’s physical address by some means. The need of a proxy and the tight dependence of processes on host inherited from host mobility make those approaches impractical.

MIGSOCK[1] is a network process checkpointing utility that is built as a kernel module. Unlike most of the network process checkpointing approaches, it does not require the presence of a proxy. Furthermore, since it uses CRAK [9] to accomplish process checkpointing, it removes the dependency of a network process on its host. To take care of data transfer during migration, the remote process is put to sleep. This mechanism is accomplished by modifying the kernel to look for MIGSOCK special migration messages.

This project seeks to validate the performance of MIGSOCK against Zap. We compare both approaches in two possible scenarios that model single and multiple service support from a server. In what follows, Section 2 describes the design of MIGSOCK. Section 3 gives an overview of Zap. Section 4 explains how Netfilter processes packets coming in. It also discusses the relationship between Netfilter and iptable. Section 5 illustrates the experiments and their setups in evaluating the overhead of MIGSOCK in checkpointing and restarting and of Netfilter in translating packets and directing it to the server. We also show the experiment setup to simulate a multiple-service scenario. Results are interpreted and our winner is determined in Section 5.4. Section 7 covers related works and compares our system to existing approaches. Finally, we present our concluding remarks in Section 7.

2. DESIGN OF MIGSOCK

The design of MIGSOCK can be broken down into two levels: the kernel level and the user level. Three components of the MIGSOCK are the MIGSOCK kernel, MIGSOCK module, and MIGSOCK user-level controlling program. Figure 1 shows the layered design of MIGSOCK. The MIGSOCK kernel is responsible for encoding and checking MIGSOCK TCP-specific packets. Upon the arrival of a TCP-specific packet, the kernel puts the remote process to sleep. Upon the arrival of another special TCP packet, the remote process is woken up.

The module is responsible for passing data between the kernel and the user space. It serves as a bridge between the MIGSOCK kernel and the controlling program. For example, when the information about a socket needs to be obtained in preparation for socket serialization, the module reads it from the kernel, where it references the socket’s inode and copies it to a buffer held by the controlling program. The controlling program would like to notify the remote host to resume for communication, the command is issued as a system call through the kernel module. The kernel then responds the command by generating a TCP-specific packet to be sent over to the remote host.

To migrate a socket, the controlling program first stops the migrating process, which is a client/server type of program using BSD socket programming API. It then hijacks the socket and sends a request to the migrating process’s remote peer. The controlling program serializes the states of the socket. The entire process is then be checkpointed by CRAK. To resume the communication, CRAK is called to restart the migrating process except its socket’s states. The MIGSOCK controlling program issues a MIGSOCK kernel module system call to reinstate the socket. It then hijacks the socket again and sends a restart request to restart communication with the migrating program’s remote peer.

Figure 2 shows the MIGSOCK message timeline. The MIGSOCK messages are guaranteed to arrive at the remote peer due to reliability provided by TCP. The dotted vertical lines are user-level controlling programs that hijack the communication with the remote process to perform socket checkpointing and restarting. Note that it is possible to either serialize the socket and send MIG REQ simultaneously or deserialize the socket and send MIG RST simultaneously in a pipelining fashion for the sake of efficiency.
3. OVERVIEW OF ZAP

Zap provides a virtualization layer on top of an operating system that introduces a Process Domain (pod). A pod is a group of processes that have the same virtualized view of the system. By utilizing Zap, each process within a pod is able to perform its service without problems of dependencies and conflicts of the underlying system.

Specifically, Zap is responsible for translating system calls between a pod and its host operating system thus avoiding resource inconsistency, resource conflicts, and resource dependencies. For example, within a pod, a process has a virtual process ID. When such a process is checkpointed and restarted, the process will have the same process ID since it lives within the pod. If there were no pod virtualization, the restarted process would not be able to be migrated over with the same process id as it used to have. The process ID that it had before might be occupied by some other process in the destination host. The virtualization is neat, as processes can be run regardless of the underlying system.

Network migration by Zap is clean, differing from end-to-end, network-layer, transport layer, proxy-based split TCP connection, and socket library wrapper approaches[8]. There is no modification of the end systems, the kernel, TCP, or transport layer protocol. Because everything is virtual, all processes within a pod communicate with a virtual IP address. The virtual IP address is translated to an external IP address by Zap for communication between processes in/outside of the pod. It preserves the established connection without destroying semantics of the application-layer, network-layer, or below.

Zap migrates network processes by first installing a proxy process that will stall current connections. Zap then suspends the pod by stopping all processes within, saving virtualization mappings and process states. Upon system restart, Zap restores processes in the pod by populating their process states. It then informs its remote host and resumes the communication. The pod and its remote host still keep their own addresses. Zap takes care of the virtual to physical address mapping to ensure that both ends do not need to worry about the connection virtualization and translation. It uses Netfilter to accomplish address translation between the pod and its communication partner. Address translation is done twice as a packet goes out and arrives at either end. The packet’s destination address first gets translated to the physical address before being sent off the wire. Once it arrives at the other end, it gets translated again to the virtual address before entering into a pod.

Since both MIGSOCK and Zap use CRAK as a basis for process migration, the paper focuses on the overhead of MIGSOCK in checkpointing and restarting sockets and the overhead of Netfilter in translating and directing packets.

4. NETFILTER

"Netfilter is a set of hooks (or chains) inside the the Linux kernel that allows kernel modules to register callback functions with the network stack. A registered callback function is then called back for every packet that traverses the respective hook within the network stack." There are five default hooks, PREROUTING, INPUT, FORWARD, POSTROUTING, and OUTPUT, placed evenly across the kernel routing to ensure that packets are properly processed before being forwarded. Each chain has a set of rules that determine the fate of packets.

As a packet comes in, it enters the PREROUTING chain where it usually gets mangled. Information in the packet’s header such as Source Address, Destination Address, or TOS get modified. This alters the routing decision as it passes the next stage. If the packet is destined for the host, it passes to the INPUT chain before it gets passed further up to processes waiting for it. Otherwise, if the kernel does not know what to do with the packet or does not have forwarding option enabled, the packet gets dropped. On the other hand, if forwarding is enabled, it gets passed to the FORWARD chain. For the packets from the host to the outside world, they enter the routing decision before leaving for the OUTPUT chain, where they can get mangled before being sent out the wire. Finally, all packets that are either forwarded through or from this host pass the last chain of POSTROUTING before leaving the host. One can see that packets entering a Linux box will have to pass at least three of these chains (PREROUTING, FORWARD, and POSTROUTING). This ensures some sort of security to the Linux system as a packet gets identified and appropriate action is taken.

Within a rule, a target or an action is defined. Default actions provided in Netfilter are ACCEPT, DROP, QUEUE, and STEEL. When a packet matches a rule in a chain, its fate is determined by the action associated with the rule. If a packet matches none of the rules in a chain, the default action will be taken in the chain. Users can define their own actions which can either be a chain referenced by the default chains in Netfilter or an extension to the default action.

Iptable is a user-level program that allows users to manipulate packets with iptable's services. Each service is associated with a table. Some notable services are NAT, filter, and mangle. These tables contain chains in Netfilter. By specifying rules within the chains, iptable provides an easy way for users to control packets in and out of a system. Readers are recommended to look at the manual page for iptables for more details.

5. EVALUATION

This section discusses the experiment setups for evaluating the overhead of MIGSOCK in checkpointing and restarting and the overhead of Netfilter in translating and directing packets. Although the source code of Zap is not available, it is sufficient to measure the overhead of Netfilter. As discussed before, MIGSOCK and Zap use CRAK to do process checkpointing and restarting. What is left for MIGSOCK to do is to send special messages to the remote host to either stop or resume it and to de/serialize the socket states. The experiment to capture the overhead of MIGSOCK focuses on these items.

As for Netfilter, it is expected that the overhead comes from translating and redirecting a packet. The experiment here is to determine the time it takes to send a packet to a server with Netfilter enabled and disabled. The difference in time between the two scenarios (Netfilter-enabled and Netfilter-disabled) will reveal the overhead of Netfilter.
5.1 MIGSOCK Overhead Experiment Setup

The overhead of MIGSOCK checkpointing and restarting derives mainly from the latency involved in sending and receiving special messages. As discussed before, these messages are MIGSOCK-specific TCP packets that notify the remote host to block packet transmission on the particular socket and to wake up the process upon completion of socket migration. In capturing the latency, time stamp is recorded before and after related MIGSOCK system calls. The difference in time should give a fairly accurate approximation of the overhead.

Since measuring the overhead of checkpointing and restarting a socket is the focus of this evaluation, the user-level socket API programs will concentrate on the single-socket communication. In other words, the server program will only accept and service one incoming connection from the client. Both the server and client are single-processed programs.

The server and client are located close to each other (about 60cm apart) to eliminate overhead due to propagation delay. The server will first initiate itself for the client to establish a connection. The client will be migrated to the same machine to isolate additional overhead introduced by a different distance. In summary, the procedures of this part of the experiment are:

1. The server initiates itself on Host A.
2. The client on Host B connects to the server on Host A.
3. (a) The MIGSOCK system call is called to stop the client and put the server to sleep.
   (b) The MIGSOCK system call is called to serialize the states of the socket with which the client establishes the connection to the server.
4. The CRAK checkpointing program is called to save client process's states and kill it.
5. The CRAK restarting program is called to create a client process on Host B and populate it with the serialized image of the process.
6. The MIGSOCK restarting program is called to recover the socket from the serialized image of the socket and wake up the server.
   (a) The MIGSOCK system call is called to de-serialize the states of the socket with which the client established the connection to the server.
   (b) The MIGSOCK system call is called to start the server and resume the connection.

Only the times in steps 3 and 6 are measured since, they are the only MIGSOCK-related system calls. The time is measured by inserting a timestamp before and after the call. The difference indicates the overhead of each of the system calls.

<table>
<thead>
<tr>
<th># Time (in ms)</th>
<th>Checkpointing</th>
<th>Restarting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg.</td>
<td>1.53</td>
<td>2.79</td>
</tr>
<tr>
<td>Variance</td>
<td>0.04</td>
<td>0.84</td>
</tr>
<tr>
<td>Std. Dev.</td>
<td>0.19</td>
<td>0.92</td>
</tr>
</tbody>
</table>

5.2 MIGSOCK Overhead Results

Figure 3 shows the time it takes to checkpoint and restart a socket. The experiment is conducted over 10 trials. The average, variance, and standard deviation were calculated. Assuming the data is normally distributed, one expects 99.7% of the checkpointing data to lie between 0.57ms and 2.10ms. One would also expect 99.7% of the restarting data to lie between 0.03ms and 5.55ms. We take the worst time in the ranges of checkpointing and restarting and used the sum to compare MIGSOCK's performance to Zap. In short, in the worst-case scenario, it takes MIGSOCK 2.10ms to checkpoint and 5.55ms to restart a socket. The total time it takes for MIGSOCK is 7.65ms.

5.3 Netfilter Overhead Experiment Setup

This section determines the effect of Netfilter with respect to packets and the rules. Sections 5.3.1 and 5.3.3 show the experiment setup for determining the overhead of Netfilter with respect to incoming packets and rules, respectively. Sections 5.3.2 and 5.3.4 bring out results from these experiments.

5.3.1 Is the Overhead Proportional to the Size of Data Packets and Their Quantities?

This section seeks the overhead that is involved to transmit a packet passing through Netfilter, if any. It is hypothesized that the overhead of Netfilter is proportional to the number of packets it needs to process. Furthermore, it is also hypothesized that this overhead increases as the packet size increases.

The experiment requires a client to pass packets to a server and the server to pass the same packets back. There are two scenarios: regular and Netfilter-enabled. In a regular scenario, a client will connect to the servers public address and port directly. In a Netfilter-enabled scenario, a client...
connects to the servers public address and port. Packets are routed by Netfilter to a virtualized IP address where the server is listening on connections and to which its port is bound. Figure 9 demonstrates these two scenarios.

To measure the Netfilter overhead, the time it takes for packets that left the client to arrive back at the client is recorded. Although the propagation time is included between the client and server in the measurement, it is negligible due to distance. The client and server machines are set up in such a way that their distance is about 60cm apart. Furthermore, since the same setup is used for both regular and Netfilter-enabled scenarios, variation in the measurement would have to be due to Netfilter.

To conduct the regular scenario of the experiment, one would first fire up the server and then the client. Within the client, timestamps are inserted before the sendto() and after the recvfrom() sys calls. Experimental results are shown in Section 5.3.2.

The Netfilter-enabled scenario contains 4 steps: creating a virtualized IP, adding a rule to invoke Netfilter, starting up the server program, and the client program. The server binds its socket to the virtualized IP and port (10.0.0.1:4884). The client connects to the public IP address and port of the server (128.2.213.1:4884). Packets with the right destination will match the rule in the server’s Netfilter and get directed to the virtualized address of the server. Once the delivery of packets is complete, the server sends the same packets back to the client to log the time of the arrival of sent packets.

5.3.2 Results of Packet Overhead

<table>
<thead>
<tr>
<th># of Packets</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>(reg) Milliseconds</td>
<td>702</td>
<td>7,047</td>
<td>70,050.5</td>
<td>703,517</td>
</tr>
<tr>
<td>(netfilter) Milliseconds</td>
<td>704</td>
<td>7,062</td>
<td>70,645.5</td>
<td>706,345</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of Packets</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>(reg) Milliseconds</td>
<td>1,111</td>
<td>11,102</td>
<td>111,109</td>
<td>1,110,822</td>
</tr>
<tr>
<td>(netfilter) Milliseconds</td>
<td>1,113.5</td>
<td>11,139</td>
<td>111,334.5</td>
<td>1,113,109</td>
</tr>
</tbody>
</table>
The times for regular and Netfilter-enabled scenarios for each packet size are shown in both graph and table forms for purposes of clarity. Difference in times are in the order of 3.5 seconds for 1,000,000 800-byte packets. It indicates that Netfilter is scalable.

Figure 11 shows the difference in time vs. the number of packets of 3 sizes. The difference gets larger as the number of packets grows. However, the size of packets does not influence the time. The three lines are all within small time differences. Although it contradicts to one of the hypotheses mentioned earlier in this section, it does make sense. As a packet passes through Netfilter, only the header of the IP packet is looked at regardless its size. Since the header of an IP packet is constant, the time it takes for Netfilter to filter the packet is also constant.

5.3.3 Is the Overhead Proportional to the Number of Rules?
This section seeks the effect of Netfilter as its rule table grows. It is hypothesized that as the number of rules in the NAT table increases, the time it takes to find the rule that matches with some header information of a TCP packet also increases. More specifically, if there is one rule in the Netfilter table, and the time it takes for Netfilter to route a packet that matches with the rule is \( t \), then the time it takes to match that rule in a table of \( N \) rules will be \( O(N \times t) \).

The experiment setup is as before, in which a client passes packets to a server and the server passes the same packets back. In a Netfilter-enabled scenario, a client connects to the server’s public address and port. Packets are routed by Netfilter to a virtualized IP address where the server is listening on connections and to which its port is bound.

The experiment contains the same number of steps as before: creating a virtualized IP, adding a rule to "invoke" Netfilter, and starting up the server and the client program. In this experiment, however, the number of rules added is greater than 1.

There are 50 rules added to the NAT table in Netfilter. Out of the 50 rules added to the table, only one rule is matched with incoming packets. Mismatched rules fall 3 categories. Rules in the first category direct an incoming port to an outgoing port. They specify what incoming port should be mapped to to what outgoing port. In this category, rules are intentionally made with ports that do not match port 4884 to be mismatched. Rules in the second category specify a destination address that does not match with the address of any incoming packet. Rules in this category are intentionally made with address that is way off to disqualify them. Rules in the third category change the source address of a packet to an address of different port as packets go out to the wire. The third category affects Netfilter in a different chain from the first and second categories. Rules in this category should be ignored since they only affect packets that go out of the receiving host.

The following code shows a sample of all 3 categories:

```
Chain PREROUTING (policy ACCEPT)
  target  prot opt source destination
[first category]
  DNAT  tcp -- 0.0.0.0/0 0.0.0.0/0
  tcp dpt:2000 to:10.0.0.1:2000
  DNAT  tcp -- 0.0.0.0/0 0.0.0.0/0
  tcp dpt:3000 to:10.0.0.1:3000
...
[second category]
  DNAT  tcp -- 0.0.0.0/0 128.2.213.2
  to:10.0.0.1:2000
  DNAT  tcp -- 0.0.0.0/0 128.2.213.2
  to:10.0.0.1:3000
...
Chain POSTROUTING (policy ACCEPT)
  target  prot opt source destination
[third category]
  SNAT  tcp -- 0.0.0.0/0 0.0.0.0/0
  to:10.0.0.2:2000
  SNAT  tcp -- 0.0.0.0/0 0.0.0.0/0
  to:10.0.0.2:3000
...
```

Example commands to issue those 3 types of rules, in addition to the correct rule, are listed as follows:

First Type:
```
iptables -t nat -A PREROUTING -p tcp
  --dport 45000 -i eth0 -j DNAT --to 10.0.0.1:25000
```
Second Type: iptables -t nat -A PREROUTING -p tcp
```
  --d 128.2.213.2 -i eth0 -j DNAT --to 10.0.0.1:26000
```
Third Type: iptables -t nat -A POSTROUTING -p tcp
```
  --o eth0 -j SNAT --to 10.0.0.2:7000
```
Correct Type: iptables -t nat -A PREROUTING -p tcp
```
  --i eth0 -j DNAT --to 10.0.0.1:1000
```
5.3.4 Overhead Fluctuates with the Number of Rules

Results are collected from sending 1,000, 10,000, 100,000 packets of size 200 bytes with rules varying from 1 to 50, with incremental of 5. Figure 12, 13, and 14 are graphs that show rules against time for 1000, 10000, and 100000 packets, respectively.

Looking at these three graphs, one can see that time tends to fluctuate as the number of rules increases. In some instance, the increase in the number of rules actually corresponds to a decrease in time. However, from Figures 12 and 14, an increase in time does appear if one only looks at the beginning (5 rules) and ending points (50 rules). According to the documentation of Netfilter, rules are looked up sequentially until the right match is found. This leads to the conclusion that if there is any increase in time, the rule space must be fairly diverse and large. This experiment shows that Netfilter with 50 rules spanning two chains is resilient to stress of at least 100,000 200-byte packets. Although the experiment did not completely confirm the aforementioned hypotheses, it did bring out a practical value of Netfilter.

5.4 Overall Findings

Based on the two experiments conducted in Section 5.3, it is evident that Netfilter performs better than expected. Although the empirical data shows some overhead, it is minimal for large quantity of packets regardless of their sizes. The overhead encountered for sending 100,000 packets is about 3 seconds. The number is obtained by subtracting 1,924,456ms from 1,927,430ms in Figure 10. In a typical scenario where a web client accesses a web page from a web server, the server will fulfill the client’s request in less than 100,000 packets. It means that the overhead of less than 3 seconds is anything but significant.

It is fairly scalable under data-intensive applications, too. Assuming a 5GB video streaming is delivered using UDP protocol to a process within a pod, it will take about 17 minutes to ensure the video stream be delivered safely and viewed properly. Depending on the need of end-to-end applications, translation using Netfilter might be suitable.

With regard to MIGSOCK, however, the overhead of Netfilter to MIGSOCK is on the order of 3 for 10,000 packets. In other words, to migrate a socket and let the remote know about the migration are cheaper than making it invisible to the remote host. One would expect the difference to increase as the number of packets to the remote host increases. In fact, the order of magnitude of the overhead is ten fold as the the number of packets increases by ten fold. Netfilter takes 30 times as much in time as MIGSOCK for 100,000 packets. It takes 300 times as much for 1,000,000 packets.

\[ \text{Overhead} = \frac{3 \times 10^7}{1500/100,000 \times 3/60} \]

It is because Netfilter only looks at the header of a packet

\[ \text{Overhead} = \frac{19276-19245.5}{10} \]

Since the overhead of Netfilter is independent of packet sizes, one can obtain the overhead by looking at Figures 6 and 8 as well.

\[ (5 \times 10^7)/1500/100,000 \times 3/60 \]

where 1500 is the size of an ethernet packet, 3 is the time it takes to process 100,000 packets, and 60 converts seconds into minutes.

\[ (19276-19245.5)/10 \] where 19,276 and 19,245.5 are obtained from Figure 10 and 10 is the total time it takes in milliseconds to checkpoint and restart a socket in Section 5.2.
Serving different types of services, represented by different rules where each one is responsible for directing traffic to a pod in Zap, Netfilter has also shown its scalability. 50 rules were created and looked up by Netfilter in a relatively fast speed. However, one can get a lower number of 7.65ms using MIGSOCK than 712ms, according to Figure 12, using Zap. MIGSOCK can model a pod as a multi-threaded program. Each thread represents a process and shares the same virtual IP address and port with the rest of the threads. By iteratively checkpointing and restarting all multi-threaded programs, each pod can be migrated and resumed at a desirable host. Therefore, it is efficient to use MIGSOCK providing translation of packets to different pods as opposed to Netfilter. Assuming the time to deliver a packet to a pod is constant with respect to the number of pods, it will take about 10 pods for Zap to break even with MIGSOCK. Therefore, MIGSOCK wins on the front of serving multiple pods.

6. RELATED WORKS

MSOCKS[2] is a transport layer approach to host mobility. In this scheme, however, an explicit proxy is introduced to replace stream mapper[6]. The proxy serves the function of splicing and re-splicing connection between a mobile client and a server. The introduction of a proxy preserves TCP’s end-to-end reliability and semantics as opposed to the stream mapper within the mobile host that modifies TCP protocol.

As sound as MSOCKS and [6] are, they assume that the process on a mobile host does not change, i.e., it will stay on the mobile host during the period of migration. MIGSOCK, on the other hand, removes the dependency of processes on a host. It provides more flexibility since a process is not bound to a host and can be moved from one host on an address to another host on a different address.

Mobile TCP socket[5] is an approach for migrating mobile hosts by inserting a virtual layer on top of the transport layer. The virtual port layer, between the socket API and the transport layer, serves the purpose of ensuring communication between two end hosts be connected at all time. The way it achieves it is by making an association of the current connection between two end hosts. Upon setting up the TCP connection, the mobile socket layer (MSL) system exchanges information between the pair of virtual ports. Information exchanged includes a virtual ID that identifies the connection, the size of virtual port buffer in relationship to the sending buffer at both ends. The virtual ID is kept at the migrating host for reconnection in the future.

However, the mobile TCP socket approach requires modifications, tweaking, or addition of some OSI layer. It is cleaner in the sense that underlying layer is not changed. However, the fact that a thin layer is introduced inevitably brings in overhead. Performance evaluation was not discussed so it is hard to know how much impact the overhead is on the system.

Unlike Mobile TCP[5], a network-layer approach, an end-to-end approach to host mobility argues that although mobility support at lower layer makes application programming easier, it comes at significant cost, complexity, and performance degradation. Furthermore, mobility modes such as TCP or UDP are constrained by lower-layer approaches. The paper[7] introduces end-to-end approach to host mobility by utilizing DNS and developing TCP migration options with existing TCP protocol.

The scheme is similar to MIGSOCK in the sense that they are all end-to-end approaches. Furthermore, both all made modifications on the TCP protocol. While implementation of synchronizing and resuming network states was the same, MIGSOCK went further to decouple the relationship between processes and mobile hosts. In other words, processes now can move among mobile hosts, which can move among different domains.

7. CONCLUSIONS

In conclusion, the paper demonstrates that MIGSOCK performs better than Zap. For providing a single service, MIGSOCK is about 3 times faster than Zap. The performance gap is wider if the number of packets exceeds 10,000 packets. For providing multiple services where each service resides in a different pod, it takes about 10 pods for Zap to break even with MIGSOCK assuming that Zap’s overhead in providing forwarding service to a pod is constant to the number of pods. The bottleneck of Zap happens at translating packets and directing them to a pod. This overhead is inherited and unavoidable regardless what underlying translation mechanism Zap adopts. For Zap to break this barrier, it means shifting the responsibility of translation to each pod. Although transparency and virtualization requirements are violated, they are not completely broken. A hybrid system of MIGSOCK and Zap indicates a viable approach to network process checkpointing and restarting.

8. REFERENCES


