Migratable Sockets (MIGSOCK)

Kevin Lee
Advisor: Katia Sycara, Joeseph Giampapa
April 15th, 2004

The Information Networking Institute is a cooperative endeavor of:

Electrical and Computer Engineering
School of Computer Science
Graduate School of Industrial Administration
Heinz School of Public Policy
Agenda

• Motivation
• Process Checkpointing
• Overview of Socket Programming
• Overview of Process and Threads
• Implementation
• Evaluation
• Future Works
• Demo (in NSH 1615 after the presentation)
Motivation

- Components to network process checkpointing:
  - Process’s states
  - Socket’s states
  - Resources (e.g. shared libraries, etc.)

- Focus on socket checkpointing
  - Useful for IP roaming where a host migrates from one place to another
  - Process states checkpointing has been explored extensively
  - Resource checkpointing is not easily handled, difficult to do in a general way
Agenda

- Motivation
- **Process Checkpointing**
- Overview of Socket Programming
- Overview of Process and Threads
- Implementation
- Evaluation
- Future Works
- Demo (in NSH 1615 after the presentation)
Process Checkpointing

What is Process Checkpointing?

- Act of saving a process's states so that the same process can be restarted in some later time.
- Benefits include fault tolerance, transaction rollback, server-load balancing, data accessibility, ease of management of processes, ip roaming, etc.

Approaches

- **OS-level**
  - OS Core or OS kernel
  - Better performance but complex to implement
- **User-level**
  - User-level libraries
  - Easier but no program transparency; for example, libckpt requires checkpointing code to be inserted in the user programs
  - Requires library linking.
Network Process Checkpointing

- **What is a Network Process?**
  - A process that communicates with another process using sockets.
  - Checkpointing requires using existing connection to establish a successful communication with the remote process upon restart.

- **Approaches**
  - Transport-layer: MSOCKS, use TCP Splice
  - Network-layer: Use TCP Migrate options and stream mapper
  - End-to-End: TCP migration options within existing TCP protocol
  - End-to-End: Zap

- **Pitfalls**
  - All of these approaches require some sort of proxy
  - Tight dependence of processes to host
MIGSOCK

- What is MIGSOCK?
  - A kernel module to checkpoint sockets
  - Uses CRAK to checkpoint process’ states
  - No proxy
  - Removes dependence of a process to a host
  - It can only checkpoint a single socket

- Operational Assumptions
  - Hosts are MIGSOCK-enabled
  - Process migration system in place (CRAK)

- Possible Extensions
  - Checkpoint multiple sockets
  - Checkpoint multi-processed program
  - Checkpoint multi-threaded program
  - Checkpoint multiple sockets for multi-processed/threaded programs

- Goals
  - Try to accomplish the above MIGSOCK extensions to the current design
  - Maintain user transparency
  - Maintain modularity
  - Maintain performance advantages to user-level approaches
Agenda

• Motivation
• Process Checkpointing
• **Overview of Socket Programming**
• Overview of Processes and Threads
• Implementation
• Evaluation
• Future Works
• Demo (time permitting)
Socket Programming API in Action

Client side

Opening client’s socket

socket

connect

write

read

close

Server side

Opening server’s socket

socket

bind

listen

accept

read

write

Main loop: await connection request from next client

EOF

The INI is a cooperative endeavor of:

- Electrical and Computer Engineering
- Grad School of Industrial Administration
- School of Computer Science
- Heinz School of Public Policy
The INI is a cooperative endeavor of:

- Electrical and Computer Engineering
- Graduate School of Industrial Administration
- School of Computer Science
- Heinz School of Public Policy

**Life of a TCP Packet**

- **Top Half**
  - Application Layer Socket Call
  - C Socket Library
  - Socket Layer Read
  - TCP Layer Read

- **Bottom Half**
  - struct sock
  - TCP Layer Processor
  - IP Layer Processor
  - Network Driver

**User Space**

**Kernel Space**

**MIGSOCK Modification**
Migsock Message Timeline
Finding a Socket’s Inode
Example of Finding a Socket’s Inode

```c
struct files_struct *files;
struct file * file = NULL;
struct inode *inode;
struct socket *sock;
files = proc->files;
file = files->fd[3]; // the file descriptor for the socket obtain from
// the process table

inode = file->f_dentry->d_inode;
if (!inode->i_sock || !(sock = &(inode->u.socket_i)))
{
    retval = -ENOTSOCK;
    printk("MIGSOCK bug: Checking for socket failed.\n");
    goto done;
}
```

Agenda

- Process Checkpointing
- Overview of Socket Programming
- **Overview of Processes and Threads**
- Implementation
- Evaluation
- Future Works
- Demo (time permitting)
Similarities and Differences of Process and Threads

- **Processes**
  - Has its own virtual memory, open files, pid, and process-specific contexts
  - fork processes only shared standard I/O descriptors

- **Threads**
  - All share the same virtual memory and open files
  - No need of external inter-communication mechanism; pipes
Fork a Process and Spawn a Thread

- Parent Process
  - Child 1
    - Open “a1” => fd[3]
  - Child 2
    - Open “a2” => fd[3]

- Parent Thread
  - Thread 1
    - Open “a1” => fd[3]
    - Access “a2” => fd[4]
  - Thread 2
    - Access “a1” => fd[3]
    - Open “a2” => fd[4]
Challenges of Checkpointing Multi-threaded/processed Programs

- States of each process
- Open files
- Sockets: replicated across threads
  - Whoever owns the socket should have rightful position to checkpoint the socket and update
  - Need to identify which thread that a socket belongs to
  - Need to communicate to other threads that the socket is checkpointed => no data should be sent
  - Need to update socket state information to prevent stale states across all threads upon restart
Agenda

• Motivation
• Process Checkpointing
• Overview of Socket Programming
• Overview of Process and Threads
• Implementation
• Evaluation
• Future Works
• Demo (in NSH 1615 after the presentation)
Design Overview for Checkpointing Sockets

- User Program
- MIG Controlling Program
- MIGSOCK Module
- MIGSOCK Kernel

User Space

Kernel Space

The INI is a cooperative endeavor of:

Electrical and Computer Engineering
Graduate School of Industrial Administration
School of Computer Science
Heinz School of Public Policy
MIGSOCK Components

- **User-Level Controlling Program**
  - Stop user program
  - Send REQ MIG/RST

- **MIGSOCK Module**
  - Indicate to the kernel about special messages
  - Serialize and deserialize sockets

- **MIGSOCK Kernel**
  - Encode special messages
  - Check for special messages
  - Put process to sleep
Design Overview for Checkpointing Processes

- User Program
- CRAK Controlling Program
- CRAK module
- MIGSOCK Kernel

User Space
Kernel Space
CRAK Components

- **User-Level Controlling Program**
  - Save process current directory, terminal, call checkpoint
  - Resume process current directory, terminal, call restart

- **CRAK Module**
  - Implementation of checkpoint
  - Implementation of restarts
Multiple Sockets for Single Process Checkpointing and Restarting

- User-level controlling program iterates through multiple sockets to send REQ MIG/RST
- MIGSOCK module support reading inode of each socket descriptor
- Use CRAK as before
- MIGSOCK kernel unchanged: same mechanism to encode TCP packets and put remote process to sleep
Checkpointing Multi-threaded and Multi-processed Program

- User-level controlling program iterates through each process and checkpoint each by calling CRAK module
  - Need to find out the process tree
  - Caveat: Parent process should be killed last to avoid premature aborting from checkpointing threads

- Kernel Module: unchanged, use CRAK
  - Saves file state by accessing struct file
  - Serialize the state of the open files to disk

- Kernel: unchanged
Restarting Multi-threaded and Multi-processed Program

- User-level controlling program forks a process iteratively and populates the process with the serialized states by calling CRAK module
  - Need to find out the process tree
  - Caveat: Parent process should be created first to simulate thread environment

- Kernel Module: unchanged, use CRAK
  - Use dup/dup2 to duplicate file descriptors
  - Populate the file descriptors with file states from serialized files

- Kernel: unchanged
Checkpointing and Restarting Sockets for Multi-threaded and Multi-processed Progs

- User-level controlling program iterates through multiple sockets to send REQ MIG/RST; use CRAK module to checkpoint and restart the process
  - Caveat: Need to make sure request for migration is sent across all processes before serializing to avoid loss of socket states
- Kernel Module supports reading inode of each socket descriptor; CRAK module unchanged
- Kernel: unchanged
Test Programs

- count_client.c & count_server.c: for checkpointing multiple sockets
- thread.c: for checkpointing multi-threaded program
- thread2.c: for checkpointing multi-processed program
- thread_client.c and thread_server.c: for checkpointing multiple sockets for multi-threaded programs
- proc_client.c and proc_server.c: for checkpointing multiple sockets for multi-processed programs
Agenda

• Motivation
• Process Checkpointing
• Overview of Socket Programming
• Overview of Process and Threads
• Implementation

• Evaluation
• Future Works
• Demo (in NSH 1615 after the presentation)
Zap

- User-level approach to network process checkpointing
- Use CRAK for process migration
- Hedges on resource translation
- Provides a virtualization layer on top of the OS that introduces a PrOcess Domain (pod)
- A pod is a group of processes that have the same virtualized view of the system
- Zap migration procedure:
  - A temp. process to serve as a proxy to stall current connection
  - Suspend the pod
  - Migrate the pod
  - Restart the pod
  - Contact the remote host
  - Zap then updates virtual to physical address mapping using Netfilter
- No access to the source
Netfilter

- Set of hooks (or chains) inside the Linux kernel that contain callback functions along the network stack
- Kernel module registers callback functions to process the packet as it traverses the network stack
- Default hooks are PREROUTING, INPUT, FORWARD, POSTROUTING, and OUTPUT
- Each hook contains a list of rules that specify what to do with the packets
- Each rule has an action (or target)
- Default actions are ACCEPT, DROP, QUEUE, and STEEL
- iptable is an user-level program to manipulate packets with its services
- Services are NAT, filter, mangle, etc.
- Each service is represented by a table, consisted of a list of chains
- Before MIGSOCK, only support translation of IP addresses
- After MIGSOCK, support translation of ports
Evaluation Goals

- Find out the overhead of checkpointing and restarting sockets using Migsock
- Find out the overhead of translating addresses using Netfilter
- Since both use CRAK for process checkpointing, results should reflect closely the overhead Migsock and Netfilter
Overhead of Migsock in Checkpointing and Restarting

- Setup: insert time stamp code in user controlling program to capture MIGSOCK system calls
- Time is measured as the round trip time for the MIGSOCK messages to get to the server and ack back
  - There is no explicit ack back from the remote process to the migrating process as the ack message did not pass up to the app. layer
  - Placing the timestamp right after the MIGSOCK sys call is enough
Results

- Results gathered over 10 trials
- Single socket checkpointing:
  - Avg: 1.53 ms
  - Variance: 0.04 ms
  - Std. Dev.: 0.19 ms
  - 99.7% of data lie between 0.57 ms and 2.10 ms
- Single socket restarting: about 6.5 ms
  - Avg: 2.79 ms
  - Variance: 0.84 ms
  - Std. Dev.: 0.92 ms
  - 99.7% of data lie between 0.03 ms and 5.55 ms
- Worst-case scenario:
  - Checkpointing: 2.10 ms
  - Restarting: 5.55 ms
  - Total: 7.65 ms
Overhead of Netfilter

- **Setup:**
  - Public IP: 128.2.213.2
  - Connects 128.2.213.1:4884

- **Goals:**
  - Measure overhead with respect to size of packets and the number of packets
  - Measure overhead with respect to the number of rules
### Overhead wrt Size of Packets of 200 bytes

<table>
<thead>
<tr>
<th># of Packets</th>
<th>1000</th>
<th>10000</th>
<th>100000</th>
<th>1000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Millisecs</td>
<td>702</td>
<td>7047</td>
<td>70051</td>
<td>703517</td>
</tr>
<tr>
<td>Millisecs (w/ Net)</td>
<td>704</td>
<td>7062</td>
<td>70646</td>
<td>706345</td>
</tr>
<tr>
<td>Difference (Millisecs)</td>
<td>2</td>
<td>15</td>
<td>595</td>
<td>2828</td>
</tr>
</tbody>
</table>
# Overhead wrt Size of Packets of 400 bytes

<table>
<thead>
<tr>
<th># of Packets</th>
<th>1000</th>
<th>10000</th>
<th>100000</th>
<th>1000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Milliseconds</td>
<td>1111</td>
<td>11102</td>
<td>111109</td>
<td>1110822</td>
</tr>
<tr>
<td>Milliseconds (w/ Net)</td>
<td>1114</td>
<td>11139</td>
<td>111335</td>
<td>1113109</td>
</tr>
<tr>
<td>Difference (Milliseconds)</td>
<td>3</td>
<td>37</td>
<td>226</td>
<td>2287</td>
</tr>
</tbody>
</table>
**Overhead wrt Size of Packets (800 bytes)**

<table>
<thead>
<tr>
<th># of Packets</th>
<th>1000</th>
<th>10000</th>
<th>100000</th>
<th>1000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Millisecs</td>
<td>1926</td>
<td>19246</td>
<td>192459</td>
<td>1924456</td>
</tr>
<tr>
<td>Millisecs</td>
<td>1926</td>
<td>19276</td>
<td>192797</td>
<td>1927430</td>
</tr>
<tr>
<td>(w/ Net)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Difference</td>
<td>0</td>
<td>30</td>
<td>338</td>
<td>2974</td>
</tr>
<tr>
<td>(Millisecs)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Observations

- Overhead increases with the number of packets
- Difference about the same; Netfilter only looks at the header
Overhead wrt the Number of Rules

- Netfilter is supposed to match rules sequentially
- Setup:
  - Send 200-byte packets of various numbers; 1000, 10000, 100000
  - 1 to 50 rules were added to Netfilter
  - Only 1 rule matches with the description of the packet
  - The only one matched rule is the last rule in the rule table; this can be accomplished by adding it to the rule table last
- Rules in 3 categories:
  - Direct incoming port to outgoing port of the virtual address
  - Specify a destination address that does not match with the one of incoming packets
  - Change the source address to a different one as packets go out (in POSTROUTING chain)
Observations of Netfilter wrt Rules

- Overhead fluctuates with increase in number of rules
- Assuming the time it takes as the number of rules increases is constant, it takes 143ms (712ms/5) to direct a packet to a pod
- This means that beyond 10 pods (143ms/7.65ms), Netfilter will break even with MIGSOCK
- However, one can place a threshold of 14 to prevent and load balancing pods among more than one server
Overall Findings

- Netfilter performed better than expected; 100,000 packets only take 3 seconds
- Also fairly scalable under data-intensive applications; 5GB video stream takes only 17 minutes
- MIGSOCK performs better in the order of 3 magnitude than Netfilter for 10,000 packets; the difference increase by ten fold as the number of packets increases by ten fold
- It takes 10 pods for Zap to break even with MIGSOCK; one who uses MIGSOCK can avoid break-even point by load-balancing across servers
Agenda

- Motivation
- Process Checkpointing
- Overview of Socket Programming
- Overview of Process and Threads
- Implementation
- Evaluation
- Future Works
- Demo (in NSH 1615 after the presentation)
Future Work

- Support fully functional multiple sockets checkpointing for multi-threaded/multi-processes programs
  - Identify the association between socket and thread
  - Update states of a socket across threads to avoid staleness and socket unusability
- Provide security features to MIGOSCK
  - No mechanism to prevent a malicious user from hijacking connection
  - Need to provide an authentication protocol for the controlling programs to authenticate themselves to the remote host
- Provide compatibility to non-MIGOSCK kernel
  - Reserve bit in the TCP packet for migration usage
  - Expand kernel module to check for migration messages and put process to sleep
Conclusion

- Single Service (only one pod on the server): 3 times as much time as MIGSOCK
- Multiple Services (many pods on the server): in the worst scenario, it takes 10 pods for Zap to break even with MIGSOCK
- Zap is sound, but translation takes way too much time
- But MIGSOCK is not completely perfect since it requires all systems to be ran on MIGSOCK kernel
- Maybe an integrated approach where translation is done within a pod
Questions