CRAK: Linux Checkpoint/Restart As a Kernel Module

Hua Zhong and Jason Nieh
Department of Computer Science
Columbia University

Abstract

Process checkpoint/restart is a very useful technology for process migration, load balancing, crash
rollback transaction, job controlling and many other purposes. Although process migration has not y
widely used and available commercial system is rarely seen, the shifting of computing facilities from
supercomputers to networked workstations and distributed systems will eventually make it play a m
more important role.

In this project we chose Linux as our platform, and describe in detail our design and implementation
an innovative transparent checkpoint/restart package for Linux. It is the first package for Unix/Linux
the following features: (1) it doesn't require modification of existing code and (2) it supports migratin
sockets.

1. Introduction

A process is an entity that is actually running in the operating system. A process has an identifier (PI
register set, an address space and/or other certain resources such as opened files. In modern opera
systems a process runs in its own address space and is separate from other processes. A process i
with the operating system, or the kernel, by system calls; it can also communicate with other proces
inter-process communication mechanisms provided by the operating system.

Process checkpoint/restart has two phases. The first phase is to save the running state of a process
usually includes register set, address space, allocated resources, and other related process private
second phase is to re-construct the original running process from the saved image and resume the
from exactly the interrupted point.

There are several problems with existing checkpoint/restart systems. First, all of them require modif
existing code, either the kernel or user applications. Second, except some written-from-scratch proc
migration operating systems (such as Sprite), they can not preserve opened network connections. T
general-purpose operating systems such as Unix was not designed to support process migration, so
checkpoint/restart systems built on top of existing OSes usually only support a limited set of applicat

In this project we designed and implemented a Linux checkpoint/restart system by means of kernel
The primary contributions are:

- Transparency to legacy applications without kernel modification
- No run time overhead besides actual checkpoint/restart
- True migration without "stub" processes or "home nodes"

- Virtual networking with very low overhead
- Network connection migration support

This paper is organized as follows: section 2 briefly describes the motivation and application of checkpoint/restart; section 3 describes related work; section 4 describes the design; section 5 descr kernel module implementation; section 6 is discussion of network socket support; section 7 describe results; section 8 is conclusion and future work.

2. Motivation and Application

Checkpoint/restart has various applications:

- **Process migration**: Transparent process migration is used for distributed load balancing and controlling systems. Processes are migrated from one host to another to achieve more efficie utilization. We will describe it in more detail in section 3. On the other hand, although there is connection between checkpoint/restart and migration, they are not exactly the same thing. Checkpoint/restart can be used in many other applications.
- **Crash recovery and rollback transaction**: A process can easily return to a previously checkp state. This is especially useful for long-running applications such as scientific computation. In checkpoint can be used to reduce the overhead.
- **System administration**: System administrators can checkpoint processes before shutting dov machine and restart them after the machine is up again or on another machine.

Although checkpoint/restart is a useful technology, it is still mainly a research subject and has not co production use. The reasons are:

- **Lack of support from popular operating systems**: Most operating systems such as Unix we designed for checkpoint/restart. It's very hard to add such functionality without significant char kernel.
- **Lack of commercial demand**: Checkpoint/restart is primarily used for high performance distri systems. There is no big demand in the market.
- **Transparency and Reliability**: Checkpoint/restart ought to be both transparent and reliable fc use, which is difficult.

3. Related Work

In this section we describe some well-known process migration and checkpoint/restart systems with design and complexity. Dejan S. Molijicic and his co-workers have conducted a very thorough surve process migration [1]. Current migration systems can fall into one of the following categories:

Clustering Operating Systems with Single System Image

These are kernel level systems which require substantial kernel modifications or being written from Examples are MOSIX, Sprite, V Kernel, Amoeba and Solaris MC. They provide automatic load bala

make workstations and servers run cooperatively as if they were a single multiprocessor computer b of transparent process migration. Usually each process still has a "home node" where it originated. process appears to be always running at its home node even after it's migrated to another node.

These systems can not suspend processes and save them offline.

Sprite Network Operating System was developed at U. C. Berkeley from 1984[2]. It provided a tran process migration environment for a cluster of workstations and a Unix-like programming interface. process had a home machine and appeared to run on that machine even if it had been migrated. K talked to each other via inter-kernel RPC. IPC and TCP were implemented by a shared network file that processes at different locations can access the same resources. This project terminated in 199 never became a success.

MOSIX is a popular distributed operating system [3]. It has had a lot of versions developped on varic systems, and the latest version 7 runs on Linux. MOSIX was designed as a scalable system. Nodes independent of each other and join and leave the cluster dynamically.

In MOSIX each process is tied to a Unique Home Node (UHN) where the user logins in. All the proc user's session have the sane UHN. Processes migrated to other nodes use local resources whenev possible, but interact with the user's environment through the UHN. If the UHN dies, all processes ti die. MOSIX uses TCP/IP to implement global interprocess communication.

Virtual Machine Systems

Virtual machine systems provide a software-emulated hardware environment. Different virtual mach separated from one another. The image of an entire virtual machine can be saved to disk and resur Examples are Disco, IBM VM/370 and Vmware.

These systems can only do checkpointing on a per-VM basis, which is too coarse-grain. The entire machine image has to be saved (typically several hundreds of megabytes) even if only a small part memory is actually being used. This results in very expensive operations.

Operating Systems that Support Checkpointing

Examples are Compute Capsule, Epckpt, Fluke, KeyKOS and Discount Checking. They provide ke support for checkpointing. Compute Capsule and Epckpt supports checkpointing and migrating a si process or a group of processes. Fluke supports very limited support for checkpointing and has no practical use. KeyKOS and Discount Checking can record the entire memory image to stable storag similar to the hibernation functionality commonly available on PC operating systems,

Compute capsule is a kernel-level process migration system developed at Stanford University [4]. compute capsule contains a group of active processes and their associated environments. Comput virtualize the application interface to the operating system and repartition state ownership so that all machine-dependent information is moved into capsules from the kernel. Each capsule has its own view of the system. A set of new system calls are implemented to support capsule management, ch and restart, which adds some overhead. The kernel is also heavily patched to support this new leve abstraction. Currently it works on Solaris 7.

Epckpt is a Linux kernel patch developed by Eduardo Pinheiro at Federal University of Rio de Jane
[5]. It installs a default signal handler in the kernel. Checkpointing a process is as easy as sending
The signal handler then will be activated and checkpoint the current process.

It has the following features:

- Transparent checkpoint/restart
- Minimize checkpoint image size by not checkpointing shared libraries and code sections
- Directly send checkpoint image to remote host
- Checkpoint a group of processes in the one-parent-many-children fashion

It also has a few drawbacks. One big problem is that it adds excessive information logging to the ke
instance, in order to know which files are opened it patches `open` and `close` system calls. It also p
mmap, fork, exit, etc and introduces a lot of overhead and makes the implementation rather comple:
minimize the overhead, it even adds one more system call, `collect_data`. By default no informa
logged. If you want to checkpoint one process, however, you have to call this system call before it s
initiate the logging. Without this logging the process can't be checkpointed. That is to say, you have
in advance whether you will checkpoint a process before you launch it. This is very inconvenient and
inefficient. Epckpt is also not very flexible in supporting parallel processes.

User-level Checkpointing and Process Migration

Adding kernel-level process migration support to industry standard operating systems is difficult as t
not designed for this. When changing the kernel is not possible, the only choice is to do it in user-sp
User-level checkpointing is not transparent. It needs to modify, recompile or relink user code. It does
legacy applications, and usually requires that applications only use a limited set of system calls. Ex
are Condor [6], CoCheck [7] and Libckpt [8].

The typical approach is to Install a signal handler to do checkpointing. It's similar to what Epckpt doe
signal to the process and the signal handler will checkpoint itself, except that the signal handler is in
space. Condor uses this approach. Libckpt uses a timer handler (which is also a kind of signal han
periodically and incrementally checkpoint itself, but this approach is only suitable for rollback recove
process migration.

There are also user-level systems that provide process migration on a programming language or mi
level. Examples are Legion, Emerald, Rover and Abacus. They support checkpointing as a basic d
object-oriented languages. Of course they require that the application be written in these languages

In summary, there are two basic categories of checkpointing or migration systems: kernel level and u
MOSIX, Compute Capsule and Epckpt are all kernel level, while Condor, CoCheck and Libckpt are
level. They have different levels of transparency, complexity, portability and performance. Transpa
means whether user applications need to be modified, recompiled or relinked. Generally speaking, a
support to kernel leads to better transparency, but more implementation complexity and less portabi
is a kernel-level system.

4. Design

In this section we describe the design of CRAK. Our design is not limited to Linux, but we will take I example to explain certain concepts.

4.1 Goals

In this project, our goal is to design and implement a light-weight general-purpose transparent checkpoint/restart package for existing operating systems.

- **Built on Existing Operating Systems.** We want the system to work on general-purpose oper: systems. We do not want to write a new operating system. We chose Linux as our platform be rapidly growing, totally free, and has source code available.
- **Transparency to User Applications.** We want our package to be general-purpose and able t checkpoint legacy applications, so we have to do it in the kernel space.
- **No Kernel Modification.** By implementing CRAK as a kernel module, we achieved virtually th level of transparency as kernel patches but avoided changing the kernel itself. This makes it n easier to use.
- **Support Parallel Processes.** Parallel processes are a group of processes that run in parallel coordinate by means of IPC (inter-process communication).
- **Support Network Applications.** No packages can migrate network applications on Unix/Linu: want to support it.
- **Good Performance.** We do not want to degrade the performance of existing systems. Espec do not want CRAK to add any run-time overhead to the system other than the actual checkpoi restart.
- **No home nodes or stub processes.** Processes should only depend on the nodes they are c running. Unlike many other migration systems, we do not want to introduce home nodes or st processes to avoid performance degradation or extra point of failure.

Since we wanted to build a kernel-level system, we considered current kernel-level systems, especi MOSIX, Compute Capsule and Epckpt. We adopted many of Epckpt's ideas because it doesn't cha kernel as much as the first two, and thus suitable for migrating to a kernel module.

There are several issues we want to discuss (network socket is discussed in section 6):

4.2 Assumptions

The basic assumptions of CRAK are:

- The operating system must support loadable kernel modules. Kernel module is defined as a p developed separately from kernel that can be loaded and run in the privileged mode. Many op systems support modules, such as Linux, FreeBSD, Solaris and Windows.
- Process private data, such as address space, register set, opened files, are accessible and re from a kernel module.
- Assume a homogeneous environment. All machines should have the same architecture, OS a installed.
- Assume the process can continue to access the same files on all machines. The files are eith global file system (such as NFS, Coda) or installed locally (such as /bin, /lib).
- Our main concern is process migration, not crash recovery/rollback.

- It's not strictly a must, but we'll focus on Unix environment. We will use Unix semantics in this

4.3 Basic Approach

CRAK does process migration by checkpointing and restarting. The basic procedure is:

1. Install and load kernel modules on all machines. Modules run in kernel space but does not re
   kernel modification.
2. Processes communicate via virtual network address translation
3. Request a process or a group of processes to be checkpointed. The request can be local or re
   can be issued by user explicitly or by an automatic migration system.
4. Checkpoint command is sent to the kernel module on the local machine
5. Kernel module stops processes to be checkpointed
6. Kernel module saves the process state to file. The process is then killed.
7. Kernel module takes care to save appropriate socket and file state
8. A new process is created on a new machine where the process is migrated to
9. The new process filled with checkpointed process state
10. The network addresses are translated for new machine by the kernel module
11. Other processes that communicated with the migrated process are notified and updated

The user interface is:

1. **Checkpoint**: the user should be able to specify which process(s) to checkpoint and where to s
   checkpointed image (save to disk or send over network). The user should also be able to con
   checkpointing behavior, such as whether to kill the process after the checkpoint, whether to ch
   the child processes, and whether to turn on some optimization switches, etc.
2. **Restart:** the user should be able to pass a checkpointed image and restart the previous proces
   it.

An automatic process migration system can be built on top of the checkpoint/restart capability and c
like load-balancing without human intervention.

4.4 Checkpoint Mechanism

4.4.1 Save Entire State

All the process private data need to be saved and recovered during a migration. This includes: add
space, register set, opened files/pipes/sockets, System V IPC structures, current working directory, s
handlers, timers, terminal settings, user identities (uid, gid, etc), process identities (pid, pgrp, sid, etc
and any other data that need to be saved.

Address space and register set are the two mandatory components we have to save and recover to
migrate a process, while others can be optional depending on the type of application.

Some migration systems use a paging technique that only transfers a checkpoint image page to the
only when the page is needed.   We don't use this technique and simply save/transfer everything at
for the following reasons:

- The implementation is far more complex
- It requires the previous process is still kept at the original node when the new process is runni
- With our optimization for process code and shared library sections, we already avoid transferri
  sections (typically 90%), so the benifit of paging is much smaller and thus not worth the effort.

4.4.2 Optimizations of Saving Address Space

An address space is made up of a few sections, each being a continuous memory block with a star
an end address and an access attribute (`read`/`write`/e`x`ecute and `p`rivate/`s`hared). For example, a
with access `'rwp'` means that it's readable, writable and private.

Saving address space is straightforward. We simply walk through all the sections, and for each one
its position, access attribute and content to the checkpoint image file.

The question is the overhead. Even a very simple process may have a total image size of several r
and large applications like Netscape could have 50M. Can we reduce the image size? The answer
For example, code sections are read-only, which means it's always the same as the original executa
the disk. We can then find the content directly from there when we do restart instead of duplicating i
checkpoint image file. Of course, this assumes the same binary exists on all the machines. This is
on a global network file system or all the machines have the same file installed locally.

So which sections can we choose not to save? We can skip a section if we know it's identical as in
original disk file:

- **This section is mapped from a file** So we know where to find it when we do the restart.
- **This section is not writable, or writable but shared**. This guarantees it's the same copy as i
  file. Only sections that are modified privately need to be saved.

Typically those sections include code sections of the program and shared libraries.

4.4.3 Local Parallel Processes Are Migrated Together

If a group of processes communicate with each other by means of local IPC, they have to be migrate
together. If they use global IPC such as network sockets or named piles on a global file system, the
have to be migrated together.

Some migrating systems reimplement local IPCs with global IPCs. For example, Sprite implements
shared file system. This is out of the scope of our paper.

4.4.4 Atomicity and Consistency

We want to save consistent state of the running processes. We stop the target processes before we
checkpointing. This ensures that they are not running and their states are not changed during the
checkpointing.

Stopping a process doesn't necessarily mean that its state will not change. For example, if a signal
a stopped process, its signal pending state will be changed. There are several reasons we don't ne
about this problem:

- If two processes talk to each other by signals, they probably belong to a group of processes a[...] be migrated together. Thus, they should both be stopped before thecheckpointing so it's impo[...] a signal to be sent meanwhile.
- If the signal is sent casually, it probably doesn't matter if it's lost anyway. Signal is not a reliab[...] mechanism itself anyway.

4.4.5 Kernel Module

All the other migration systems have one common drawback: they require modification of existing c[...] kernel or user applications), thus are difficult to deploy. For example, MOSIX and Compute Capsule[...] kernel heavily. Moreover, it's not possible to patch the kernel if the operating system doesn't have s[...] available.

Fortunately, most operating systems support the concept of dynamically loadable kernel modules. K[...] modules are typically loaded at boot time or on demand when certain functionality is needed. Once [...] module is loaded, it becomes part of the kernel address space and runs in privileged mode.

One of CRAK's key design concepts is to do checkpoint/restart as a kernel module, which is the mo[...] challenging problem in building CRAK. Although a kernel module runs in the same privileged mode[...] only cooperate with the kernel instead of changing the kernel behavior at will. For example, we can[...] change the scheduling or swapping behavior in a module if the kernel doesn't provide this kind of ho[...]

Many packages, like Condor and Epckpt, use Unix signal handling mechanism to do checkpoint. W[...] process needs to be checkpointed, a special signal is sent to it. The real work is done by the signal[...] In another word, the process checkpoints itself. In this approach, the checkpoint can be done in the[...] process' context. It's then very easy to access the address space and register set.

However, we can not add a special signal and a default signal handler with a module. As a result, v[...] checkpoint a process in its own context; instead, we have to checkpoint it directly from another proc[...] 'checkpoint' program). The basic problem is then how to access an arbitrary process' private data fr[...] kernel module, especially the address space and register set. This will be discussed in 5.2 and 5.3.[...] other hand, most checkpointing operations such as saving opened files have no much difference be[...] kernel patch or a module.

Restart is very similar to the system call execve: the kernel loads the checkpoint image into current[...] (the 'restart' program) address space, and the current process is completely replaced. It's basically [...] to a kernel patch or a module.

Another problem is the kernel usually only provides a limited set of APIs forkernel modules. In a ke[...] we can call any global kernel functions, but in a module we can only call the "exported" functions. I[...] comply with the APIs, there are only two ways to overcome this constraint:

- Read the kernel source and duplicate the code we want to use. The problem is code duplicati[...] size and maintenance overhead (suppose in a new version of the OS the corresponding code [...] changed).
- Patch the kernel so that it exports the wanted functions. The problem is it violates our design [...] patch the kernel".

We will discuss how we solve this problem on Linux in section 5.7.

4.4.6 Process Cleanup

The old process needs to be cleaned up and recourses be reclaimed when we do a migration. Fort
this can be done by the kernel automatically. We simply kill the old process, and the kernel will take
the rest, such as reclaiming allocated memory, closing all the opened files and flushing dirty pages,

4.5 Restart Mechanism

Restart recovers processes from saved checkpoint images. It's relatively simple and straightforward
very similar to the system call "execve". What it does is:

- Create a new process
- Maintain parent/child relationships for parallel processes
- Recover process address space by mapping memory sections from the checkpoint image
- Recover register set
- Reopen files with the same descriptor numbers
- Recover network socket

Unfortunately, not all the data can be guaranteed to recover. For example, if thePID number is alrea
it can't be recovered. If the file to open is already opened and locked by another process, we can n
again.

4.6 Special Issues for Networked Processes

Special steps need to be taken for maintaining open network connections. After the process is migr
network addresses need to be translated correctly. Particularly, the migrated socket is mapped to tl
physical address. The other end of the connection should also know this new mapping so it continu
to the new node instead of the old one.

4.7 User Space or Kernel Space Solutions

Although CRAK is a kernel level system, some operations can also be done in user space. An ope
be done in user space if there is a system call to do the same job. User space implementation is us
preferred because it's easier to do (right) and more portable. The basic philosophy is that an operat
may have very different internal data structures and algorithms across different versions, but it alwa
(reasonably) stable application programming interface.

For example, to restore a file descriptor we could directly mock with the file table in the kernel, but tl
work if in a later version the OS changes the way it uses file tables. On the contrary, if we do it in us
as described in 5.4, as long as the OS still provides "open" and "dup2" system calls (which is highly
since they are standard APIs), our code will continue to work.

However, sometimes we are still in favor of kernel space because it's usually faster. For example, t
file seek pointer, we could use the "lseek" system call in user space, but it requires a context switch
implementation is actually very simple - just a one-liner. In this case we just do it in kernel.

5. Implementation

In this section we describe our implementation of CRAK as a kernel module on Linux 2.2.x/i386.

The basic picture of checkpoint/restart is, needless to say, composed of two phases: checkpoint and Checkpoint means that we save the running state of a process to a image file. Restart then restarts process based on the image file. So the basic questions are 1) what states we need to save; 2) how the states; and 3) how to recover the states.

We first give an overview of the implementation, and then describe the key components we need to recover.

5.1 Overview

There are basically two ways of using a kernel module from a user application. The first is using a d (usually in /dev). The kernel module registers itself with a device file when it starts up, and user app can interact with the module by standard file operations (open, close, read, write, ioctl, etc). The othe through the /proc interface. The module can register entries in the /proc pseudo file system, and inte user applications by reads and writes. We chose to use the first approach.

A device file interface, `/dev/ckpt`, is implemented. It provides IOCTL interface for checkpoint and We encapsulate this device file in a helper library (ckptlib.c) to provide the following application prog interfaces.

```
int checkpoint ( int fd, int pid, int flags);
```

Three parameters are passed: descriptor of the file the checkpoint image should be written to; pid of process; the flags.

Flags can be or-ed by the following:

- `CKPT_KILL`: If set, the target process is killed immediately, or it would continue to run.
- `CKPT_NO_BINARY_FILE` : If set, code sections of the binary don't get dumped.
- `CKPT_NO_SHARED_LIBRARIES` : If set, shared libraries don't get dumped.

These flags are taken from Epckpt. Their meanings are already discussed in section 4.5. We also l another thing from Epckpt that we pass a file descriptor instead of a path name to the checkpoint rou kernel then writes checkpoint image to this descriptor. In this way the image is not necessarily save disk. Instead, it can be sent directly to a remote machine via a socket. This significantly reduces th checkpoint overhead for process migration.

```
int restart (const char * filename,  int pid, int flags);
```

Three parameters are also passed: file name of the checkpoint image; a pid; flags. This system cal image, then replaces current process by the checkpointed process.

The last two parameters are mainly for group checkpoint/restart:

- `pid`: If the `RESTART_NOTIFY` flag is set, when the restart is finished the kernel will send a `SI` signal to the process specified by `pid`. If `pid` is 0, send it to the parent process.
- `flags`: Two flags are supported. `RESTART_NOTIFY` tells the kernel to notify a certain proces the restart is done, and `RESTART_STOP` tells the kernel to stop the restarted process immedia (usually use `RESTART_NOTIFY` at the same time so that another process can continue it).

As mentioned before, we pass a file descriptor to the checkpoint routine, and can directly send chec image to remote machine. Can we do the same thing for restart: receive checkpoint images from re machine and directly restart it without saving to the disk? Currently we can't mmap requires a concr anyway.

We also developed a suite of user utilities:

- **ck**: checkpoint a process and save to a file.
- **restart**: restart processes.
- **dump**: dump the information of a checkpoint image file.

5.2 Address Space

On Linux all the processes have an address space from 0 to 4GB, but only a small part of it is actua Each process has a "task_struct" stucture which holds all the private process data. From the structu walk through all the memory sections (called Virtual Memory Area, or VMA).

During the checkpoint we need to access the target process' address space from the kernel. Comm functions like `strcpy` wouldn't work because pointers in kernel and user space have completely d meanings. Instead, the Linux provides a set of helper functions (`copy_from_user ()`, `copy_to_` etc) that allow data transfer between kernel and the current process. But now we want to access a address space that is not the current one, so we can't use these functions either.

Each process has its own page directory and page tables that are initialized in a fork and switched t context switch. All the processes have an address space from 0 to 4GB, but mapped to different ph memory regions due to their different page structures.

The kernel also has its own page directory and page tables, but it is special because it just maps the address to itself. For instance, a pointer of `0x00004000` in a process may actually points to a phys address of `0x01234000`, but in the kernel a virtual address is exactly its physical address.

Now suppose we want to access the address *addr* in a process *p*. The physical address of *addr* is a of both *p* and *addr*. Since this physical address is also the virtual address in the kernel, we can ther access it (using `strcpy` or direct pointer dereference). A function, `get_kernel_address ()`, is implemented. It takes *p* and *addr* as parameters and returns its physical address by using *p*'s page and page tables.

Another problem is that this *addr* may have been swapped out. In this case, it has an invalid page t and we need to manually swap it in so that it can have a physical address.

5.3 Register Set

To be able to restart a process, we have to make sure that all registers have the same content as th
checkpointed point. In a context switch, the currently running process' register set is saved and the
switches to another process selected to run, and we need to save it during a checkpoint. As a resul
process we want to checkpoint must not be running, otherwise the saved register set is not in sync
run-time register set.

To recover we simply replaces the register set in the current stack with our checkpointed values. Wh
return from the restart call, they will be loaded into the registers.

The only problem is where it is. On Linux, a process' task structure is within the same 8KB frame w
kernel stack, and the register set is saved at the top of the stack, so there is a simple connection be
locations of process' task structure and its register set location. Assuming `struct task_struct *p` is
to the task structure, then the corresponding register set location is:

```
struct pt_regs *regs = ((struct pt_regs *)(2*PAGE_SIZE + (unsigned long)p)) - 1;
```

5.4 Opened Files

There are basically three types of files: named files, pipes and sockets. They have different forms c
Named files are identified by path names. Pipes are anonymous and directly identified by inode nu
The identity of sockets is protocol dependent but typically is a tuple of network addresses and ports.
to save the identity of a file based on its type. For all of them we also need to save the values of file
we make sure that when we restart the process all the file descriptors refer to the original files.

In the case of named files, if we have a global file system (like NFS), we can just save the path nam
file is not on a local file system (e.g., /tmp), or it has different path names on different machines, we
save the entire content of this file instead of just the path name.

Then how to restore them? Since after the restart the process continues to refer to these files by th
descriptors, we have to ensure that these files are opened with exactly the same file descriptors as
The answer is using `dup2()`. `dup2()` duplicates a file descriptor and both old and new file descrip
to exactly the same file. Because the code is fairly simple, we attach it here:

```
int open_force (int fd, char * filename, int flags, int mode) {
  int ret = open(filename, flags, mode);  // open the file
  if (ret < 0 || ret == fd) return ret; // if error or already the desired descriptor jus
  if (dup2(ret, fd) < 0) return -1; // dup the descriptor, note this should not fail
  close(ret); // close the original descriptor
  return fd;
}
```

This function forces the file to be opened with the specified file descriptor. We just call this function
space for each descriptor we saved. Since descriptors are unique, there are no conflicts.

For pipes it's more complex because it involves a group of processes, but the idea is the same. We
parallel processes in 6.6.

For network socket we'll discuss in section 7.

Currently our current implementation only supports saving the pathnames of opened files, not their

works if we have a global file system as discussed in section 4.5.

5.5 Other States

We also checkpoint and restart the following states:

- **Current working directory**. It's restored by calling 'chdir' in the user space.
- **Terminal mode**. Certain applications set the terminal mode, like vi or emacs. This is done in space by C functions `tcgetattr()` and `tcsetattr()`.
- **Signal handlers**. This is done in the kernel.

5.6 Parallel Processes

Very few packages support checkpoint/restart for parallel processes.

Epckpt supports it by passing a flag to the system call to tell it to checkpoint not only the specified p also all its children. In another word, it is implemented in the kernel. There are two major drawback

- **Inflexibility**. It only supports one fashion of process family: one-parent-many-children. Since i hard-coded in the kernel, it's very hard to extend.
- **Complexity**: The complexity of group checkpoint/restart lies in synchronization. All then proce interact with each other should be checkpointed/restarted simultaneously. For instance, if two are talking to each other via a pipe, checkpointing and killing one of them will cause a broken probably kill the other as well before it gets checkpointed. In order to checkpoint and restart p processes, epckpt uses complex synchronization mechanisms in both kernel and user space.

We made significant effort to improve it. Instead of semaphores we use signals to do synchronizatic almost all the support is done in user space:

- **Simplified kernel task**: Kernel only checkpoints one process at a time. It doesn't know anyth parallel processes.
- **Simplified synchronization**: Synchronization is done by signals. For checkpoint, first send a to all the processes in the target group to "freeze" them, then checkpoint one by one. For rest second and third parameters are used. A control process is responsible for restarting all the p Each time a process is restarted, the second parameter is set to the pid of the control process, `RESTART_NOTIFY` and `RESTART_STOP` are set. Once it's restarted, it's stopped immediatel control process is notified. After all the processes are restarted, the control process sends a S: to all of them to let them run.
- **Much more flexibility**. by moving the support to user space, potentially we can checkpoint/res kind of process hierarchy.

5.7 Kernel Symbols

As we discussed in section 4.2, sometimes we need to access kernel functions or variables that are exported by the kernel to our module.

There is a dirty trick: `/boot/System.map` . This file is generated each time a new kernel is compile

contains absolute addresses for all kernel symbols (even those not exported to modules), and we ca
use them.

For example, if there is an unexported kernel function `tcp_v4_rehash()` we want to use, we can
System.map file and find the following line:

```
c0173e00 t __tcp_v4_rehash
```

then we can use this function by directly calling the address `c0173e00`.

6. Network Socket

Network socket migration is the most challenging part in process migration. It's difficult because it in
machines. In this project we implemented a prototype of TCP/IPv4 socket migration, and have been
successfully migrated certain network applications.

6.1 Approaches to Socket Migration

When a process is migrated, we have to find a way to keep the previous network connections so tha
continue to talk to its peers, which are typically at a remote site.

One approach is that we keep a "stub" process at the original node, whose responsibility is to keep
connection and forward the traffic to and from the new node. The other end doesn't need to know al
goodness is that all the machines that need to support migration are under our control.

However, there is one big problem in this approach: each process is thus tied to the first node it run
is usually called a home node). If the home node, the process dies. This weakens the usefulness of
migration.

In this project we adopted a different, more direct approach. We do not use stub processes. Instead
migrate a process, we save the network stack and restart a new connection on the new node. We a
the remote peer of the migration so it can talk to the new address.

In this way when the process is migrated it no longer depends on the original node. There is a draw
however, that now we have to be able to access the remote peer and change its socket information

6.2 The Sock Structure

In the Linux kernel, all sorts of sockets (inet, unix) have a struct "socket". For inet family, socket has
to the struct "sock", which has all information of a TCP/IP socket.

A "sock" is identified by a unique tuple (protocol, saddr, sport, daddr, dport), where "s" means "sourc
"local", and "d" means "destination", or "remote". The following shows how a sock gets its identity a
relates to the BSD socket API:

```
socket() -> create the socket structure in the kernel, and set protocol.

bind() -> get sport (if saddr is not zero it also gets saddr)
```

```
connect(), accept() -> get full identity (protocol,  saddr, sport, daddr, dport)
```

During the socket migration, we need to change the sock information, especially addresses and por example, for the migrated process we need to change its saddr and possibly sport, and for the remo we need to change its daddr and possibly dport. It's not just as simple as to change the values, beca sock structure has interactions with various kernel data structures.

Socks are put in various hash tables to make lookup more efficient. The most important ones are th hash (tcp_bhash), listening hash (tcp_listening_hash) and established hash (tcp_ehash). For examp an incoming packet comes in, and if it's for an established connection, the kernel will lookup in the to to find the corresponding sock structure based on the hash key (the full address tuple in the packet).

The following is a summary of how the sock is linked in the hash tables.

	Description	Hash key	Sock field	Relevant functions
Protocol*	all socks bound to a local port		sklist_next, sklist_prev	add_to_prot_sklist, del_from_prot_sklist
Bind	all socks bound to a port	sport	bind_next, bind_pprev	tcp_v4_get_port, tcp_put_
Listening	all socks in listening state	sport	next, pprev	tcp_v4_hash, tcp_v4_unh tcp_v4_rehash
Connected	all socks in connected state	saddr, sport, daddr, dport		

When we change its address/port, we need to make sure the sock is still in the correct hash tables.

Another thing is the routing table. Each sock structure has a 'rtable' that caches the information as w send the packet, thus avoids looking up in the global routing table (in Linux it's called Forwarding In Base, or FIB) each time it sends out a packet. If we change the remote address, we have to rebuild information.

[*] in kernel 2.4 the single protocol linked list no longer exists.

6.3 Three Steps to Migrate a Socket

To be simple, we call the process we checkpoint A, the other end B, and the new node C.

The approach to migrate the socket is very straightforward: firstly, we checkpoint A and leave B still secondly, we restart A on C and recover the socket to "established" state which points to B again; th change B to point to C, thus recover the whole connection.

6.3.1 Half-close the Connection

When A is checkpointed, all the open files will be closed. For a TCP socket, a FIN packet is sent to connection is then closed.

To be able to migrate the connection without disturbing B, when we checkpoint A we manually set the state to `TCP_TIME_WAIT`, then it won't send FIN packet to B anymore when it's closed.

We also need to save the TCP stack information. Most importantly the sequence numbers so later we resume the connection. For simplicity, we save the whole TCP stack and restore everything except pointer fields that are apparently non-migratible.

6.3.2 Half-recover the Connection

When A is restarted, we rebuild all the sockets it previously had. The idea is to establish a connection end without calling `connect()` or sending any packet out to the wire.

This is done both in user and kernel space.

In user space, we call `socket()` and `bind()` to create the socket. In kernel space, we fill in remote remote port and other TCP stack information we saved. Since we change its remote address and port need to rehash it. Then we fresh the `rtable` cache. At last we manually set the state to `TCP_ESTABLI`

By the way, listening ports can be simply recovered by calling `listen()` in the user space. Since it involve the other end, it's trivial.

6.3.3 Fully Recover the Connection

The last thing left is that the other end, B, still knows nothing about the new location C. We need to daddr and dport pf B's socket to C.

When A is restarted, **restart** will invoke **chsock** on B. Currently the remote execution is done by `rsh` but it can be replaced by our own protocol.

7. Results

Hardware: Intel Celeron 433MHz, 128M memory
OS: Redhat 6.1, Linux kernel 2.2.14

All client machines were on a 100MB Fast Ethernet network.

The test was done over NFS v3. The NFS server was a dual-processor Sun4u Sparc running Solaris 256MB memory. The file system involved in the test was on a seagate 4.2GB SCSI disk (there were other disks hosting other file systems on the server). The Linux client ran with NFSv3 UDP support.

7.1 General Test

We tested CRAK on some common Unix applications and it worked fine for most of them, such as `cat`, `find`, `grep`, `ls`, `more`, `emacs`, `vi` and for parallel processes like `grep|cat`. However, it to restart or produced unexpected results for certain applications. For example, `top` always segfault restart.

For network applications, we have been able to migrate telnet, ssh, and some X applications. Theore could also restart daemons such as telnetd, but because telnetd forks login and the latter forks a she current implementation of the user utility "restart" can't deal with 3-layer hierarchy yet, we can't do th it's just lack of user level support. All the kernel work has been done.

The basic procedure of testing networking applications is as follows:

All operations are done as root. We run the application (e.g., telnet) from machine A to connect to E "ck pid image" on A to checkpoint A. Then we do "restart image" on machine C, then C can continu B over the recovered connection. To be able to run rsh on B, we need to have A and C in B's ~root/

7.2 Overhead

To be able to support checkpoint/restart, Epckpt does a lot of information logging. Compute Capsule the extra capsule management overhead. For example, a capsule creation operation costs 74.6us.

On the contrary, CRAK has no overhead unless you actually issue the checkpoint/restart commands primary overhead is saving the checkpoint image to disk.

Below is a simple test on three cases: find, grep, and parallel processes of grep|cat. In each o checkpointed with and without the CKPT_NO_BINARY_FILE and CKPT_NO_SHARED_LIBRARIES optimization (the first and second rows respectively). Time was measured by inserting gettime funct beginning and end of kernel routines. We can see that:

- By optimization we typically save **80%-90%** time and space.
- Checkpoint overhead is proportional to the size of checkpoint image.
- Restart overhead is surprisingly small and relatively constant, since we only do a mmap for all sections instead of reading their content. It doesn't count later page fault overhead, however.

Application	Checkpoint Time (ms)	Restart Time (ms)	Image Size (KB)
find	218.3	4.1	166
	2182.0	4.4	1246
grep	504.7	2.1	406
	2365.9	4.5	1522
**grep	cat**	558.7	9.5	523
	4347.8	10.0	2719

[*] these tests were run several times and averages were calculated.

8. Conclusion and Future Work

We have demonstrated that it is possible to implement a general-purpose transparent checkpoint/re package for a popular operating system like Linux. We also demonstrated that we could make it wit patching either kernel or user applications. This level of transparency has never been achieved by a packages.

References

[1] Dejan S. Milojicic, Fred Douglas, Yves Paindaveine, Richard Wheeler and Songnian Zhou, *Proc Migration*, HP Labs, AT&T Labs-Research, TOG Research Institute, EMC, and University of Toront Platform Computing.

[2] Ousterhout, John K., Cherenson, Andrew R., Douglis, Frederick, Nelson, Michael N., and Welch, *The Sprite Network Operating System,* IEEE Computer, 21(2), February, 1988.

[3] Ammon Barak and Oren La'sdan, *The MOSIX Multicomputer Operating System for High perform Cluster Computing.* Institute of Computer Science, The Hebrew University of Jerusalem.

[4] Brian K. Schmidt, *Supporting Ubiquitous Computing with Stateless Consoles and Computation C (PhD Thesis)*, Stanford University (http://suif.stanford.edu/~bks/).

[5] Eduardo Pinheiro, *Truly-Transparent Checkpointing of Parallel Applications (Working Draft)*, Fed University of Rio de Janeiro UFRJ

[6] M. Litzkow, T. Tannenbaum, J. Basney, M. Livny. *Checkpoint and Migration of UNIX Processes i Condor Distributed Processing System.* University of Wisconsin Madison.

[7] J. Pruyne, M. Livny. *Managing Checkpoints for Parallel Programs.* University of Wisconsin Madis

[8] J. Plank, M. Beck, G. Kingsley. *Libckpt: Transparent Checkpointing under Unix.* Princeton Univer