Using Application-Driven Checkpointing for Hot Spare High Availability

Antti Kantee
<pooka@cubical.fi>

Cubical Solutions Ltd.
http://www.cubical.fi/

ABSTRACT

For critical services downtime is not an option. The downtime of the service can be addressed by replicating the units that provide the service. However, if the session state is important, it is not enough to simply replicate units: sharing the continuously updated internal state of the units must also be made possible. If execution can be continued on another unit after the point-of-failure without any significant loss of state, the unit is said to have a Hot Spare.

Saving the state of a unit so that it can be restored at a later point in time and space is known as checkpointing. For the checkpointing approach to be a viable option in interactive services, it must not disrupt the normal program operation in any way noticeable to the user.

The goal of this work is to present a checkpointing facility which can be used in applications where checkpointing should and can not disrupt normal program operation. To accomplish this, the responsibility of taking a checkpoint is left up the application. The implications are twofold: checkpointing will done at exactly the right time and for exactly the right set of data, but each application must be individually modified to support checkpointing. A framework is provided for the application programmer so that it is possible to concentrate on the important issues when adding Hot Spare capabilities: what to checkpoint and when to checkpoint. Checkpointing efficiency is then further increased by introducing kernel functionality to support incremental checkpoints.

1. Introduction

Hot Spare High Availability support for an application means that if (when) the primary unit fails due to a fault in either software or hardware, a reserve unit will automatically take over the responsibilities of the primary unit. Execution will continue in the reserve unit with no or insignificant loss of internal application state. In a networking context this means that for Hot Spare support to be accomplished, the relevant pieces of the internal application state must be succesfully delivered to the spare units over the network at key points during execution. In addition to delivering the state to a spare unit, the system must have some cluster control mechanism that will take the necessary steps to transfer control to a reserve unit when the current primary unit fails. Once the problems involving saving state and restoring state are solved, the rest is mostly an issue which software professionals tend to call a

SMOP[1]. Therefore, the bulk of this work will concentrate on discussing the ideas involving saving and restoring process state.

There are already plenty of good examples on fault tolerance in the world of computing. A popular example from the world of hardware is disk RAID. Certain RAID levels provide protection against data (state) loss in the case of unit failure. This is what we are looking for. However, in the world of software it is difficult to keep state across unit crashes. Therefore, most entry-level solutions for fault tolerance only include support for replacing the broken processing unit, and give the problem of keeping state less attention or outright ignore it. If the unit state at failure-time is ignored, it is not possible to provide a seamless user experience across points of failure.

The act of capturing a process state for later restoration is known as checkpointing. Traditionally checkpointing has played big role in scientific computation, where the requirements for checkpointing efficiency and application interruption have not been high on the list. If checkpointing is to be used in environments where pausing the application for abitrary periods is not acceptable, new techniques must be developed.

One of the reasons for the low efficiency of the methods mentioned above is that they are implemented below the application level and transparent to the application. While this means that the application does not need to worry about checkpointing, it also means that checkpointing cannot reach maximum efficiency, since the checkpointing facility does not know about the semantic behaviour of the application. This problem is magnified if the application is not "self-contained", i.e. it communicates with the outside world. The solution is to provide a checkpointing framework for the application, and then modify each individual application to use that framework. This way checkpointing effiency and accuracy can be increased to an acceptable level.

In Chapter 2 of this work I will concentrate on defining Application-Driven Checkpointing, and giving a general overview of the architecture. Chapter 3 discusses adapting a simple open-source application to the framework. Efficiency of the framework is discussed in Chapter 4, and the story closes with conclusions in Chapter 5. This paper will present the architecture very briefly. For a more through discussion on the subject, the interested reader is invited to look at my Master's Thesis [1].

2. Application-Driven Checkpointing

First of all, it should be noted that there are several components in a process checkpoint, and they can be divided in different ways [2]. However, I wish to define a simple division and only separate process *data* and *metadata*. Data involves memory used by application. This memory is reserved from the heap, memory reserved from the stack does not count as data in this definition (neither does it count as metadata). Metadata is all the other state related to the process, such as structures describing open files and existing threads. Usually most information involving metadata is hidden from the application e.g. in the kernel.

To record the processor physical state[2], the usual approach is to simply save the register contents for example by taking the core dump. However, we can observe that most programs are structured so that they have specific worker loops. An example of such a worker loop is a function (perhaps in its own thread) reading input from a network socket and processing it. In the usual case it suffices to record the informa-

[1] Simple Matter Of Programming

[2] By this I mean the register contents, and am less interested in the actual electrons running around.

Checkpointing Kernel Interface

```
struct cpt_range {
        void  *addr;
        size_t len;
};

pid_t   cptfork(void);
ssize_t cptctl(struct cpt_range *ranges,
               size_t nranges, int op);
```

tion that the program had a worker loop, and do a normal function call into the worker loop (with the correct input data, of course) during restoration.

The above strategy also takes care of the difficulties in checkpointing threaded programs [3], where great care must be taken to avoid other threads entering a bad state. Other threads could, for example, make a syscall between the timeframe the checkpointing thread decides to checkpoint and actually makes the checkpoint. A straightforward restoration from this checkpoint would cause invalid return values from the kernel[3]. Solutions such as suspending all threads for checkpointing have a fair performance hit, especially if checkpointing is to be attempted often for increased checkpoint granularity.

2.1. Architectural details

The implementation was carried out on two different levels. Most of the work is done by a userspace library (Hot Spare Library), but of course the work done by the kernel components is implemented inside the kernel. Ultimately the application does not need the Hot Spare Library at all, and could make the respective calls itself, but the idea was to make the application programmer be able to concentrate on the critical issues and get as much support from the system as possible.

[3] No, there are no actual returned values in this case. That's why they are funny.

The user library deals with issues related to capturing and restoring the application metadata, reserving checkpoint-safe memory, and also providing a grand unifying interface, `hs_cpt`(), for taking a checkpoint.

The kernel side takes care of providing cheap, atomic and asynchronous (from the point-of-view of the calling thread and application) snapshots of the memory area.

2.2. Checkpointing data

On UNIX®systems, the *fork*() system call creates a process, which is almost an exact duplicate of the calling process the only main difference being the process ID number. Historically, the *fork*() call really did copy the entire address space of a process when executed. However, this was mostly wasteful, since *fork*() is frequently used in conjugation with the *exec*() system call, which replaces the entire address space with a binary image from the disk. Therefore a technique called copy-on-write, or COW for short, was employed in AT&T System V UNIX. The copy-on-write property of *fork*() is close to what we are looking for: it will give us both asynchronous checkpointing ability and an atomic snapshot of the checkpoint-range.

Incremental checkpointing means saving only the portions changed from the previous checkpoint, and provides a cheap performance boost in nearly every imaginable case. Therefore it is desireable to

implement support for incremental checkpoints. Most userspace solutions employ *mprotect*() to deny writing to critical areas and track modification information with the help of a `SIGSEGV` handler [4]. However, since we are allowed to play in kernel land, it is possible to avoid jumping between the kernel and userspace to track modification information. Modification information can be tracked for example in the copy-on-write fault handler or my asking the MMU. Tracking modifications in the fault handler would have its advantages, but since the latter was easier to implement[4], it was done for this work.

Kernel interface for checkpointing data

We must modify the kernel and VM [5] to support three different operations for incremental checkpointing to be possible:

- Add and remove memory areas which contains checkpoint data.
- Take the checkpoint itself.
- Ask the kernel which pages of memory in checkpoint areas have been modified since the previous checkpoint.

The call sequency for the application (or actually a programming library) which wishes to use the interface is approximately the following:

1. Decide which memory areas contain data critical enough to be worth checkpointing. Add those memory areas.
2. Decide it is time to checkpoint. Make the checkpointing syscall.

2½. The parent process from *cptfork*() continues execution as normal, and makes all the changes it wants to the checkpoint memory areas. They are "protected" by copy-on-write.

3. The child process queries the kernel for modified areas.
4. The child process writes the changed memory areas (along with other checkpoint data, we'll get to that soon) to back storage using its method of choice, e.g. write to file or TCP socket.
5. The child process exits.

[4] At least for platforms which have an MMU that keeps this information. The SPARCv9 MMU for example does not.

2.3. Checkpointing metadata

For checkpointing metadata a slightly different approach was taken. Most of the information related to process metadata is hidden in the kernel away from the application. While we could simply add a kernel interface to extract the in-kernel information for the Hot Spare Library to trasmit over the network, it would not be a good idea. The kernel structures, such as vnodes [6], are very integrally linked to each other. Attempting to extract and restore them as opaque data is not possible without creating a huge mess. To grasp the concept of checkpointing metadata, thinking of Java Serializable [7] or Python Pickle [8] may help.

I will go over one example of capturing and restoring process metadata. The rest of the descriptions are available in my thesis [1].

Checkpointing file descriptors

There are several different type of file descriptors: normal files, pipes, sockets, crypto descriptors, and so forth. Not all of them are supported. The serialization information depends entirely on the type of file descriptor we wish to serialize. For example, for a file the important facts are the filename used to open the descriptor, the mode it was opened in, and the current seek offset into the file. None of that information applies to a networking socket and we must provide other routines for it.

The information related to file descriptors is not static: for example file offset will constantly change if the file is accessed. Therefore the library provides an

option to "refresh" the information related to a file descriptor during each checkpoint by asking the kernel. For a normal file this would consitute of calling *lseek*(), while for networking sockets it would most likely be a matter of *getpeername*() and *getsockname*(). As there is a minor cost-penalty for doing this, it is not done for all file descriptors, but rather the choice of which descriptors are critical in this respect is left up to the application programmer.

Of course there is one huge downside to querying the information at checkpoint-time: since the entity doing the checkpoint and the application itself are not (necessarily) synchronized, the state that gets written into the checkpoint does not necessarily reflect the state present in the memory dump. The application programmer is encouraged to very carefully think how important the exact file descriptor state is, and possibly even take steps to record the state in the lock-protected checkpoint-area, where it will be guaranteed to be correct. However, doing so will probably open a whole other can-of-worms™, and currently there is no easy solution to the problem.

3. Adapting the framework

Adapting the checkpointing framework to the all-important game Tetris is presented next[5] [6]. While the loss of a Tetris score may not be the most tragic episode that has hit human history, migrating the Tetris game to a nearby system when the original gets (literally) axed makes for a powerful visual effect.

[5] Creating a clustered Tetris solution was suggested by Marcin Dobrucki, obviously as a joke, but he should be more careful around humor impaired people.

[6] NetHack was of course considered, but since it, as most games, always comes with its own application-driven checkpointing mechanism (savegames), the redundancy did not seem worth the effort.

Tetris from the BSDgames package is a fairly small program. The version against which this discussion is written can be found from the NetBSD CVS Repository in *src/games/tetris* with the tag `netbsd-1-6-PATCH002`. It constitutes of less than 2000 lines of code.

The state of the Tetris game can be broken into the following elements:

- score
- current piece
- next piece
- state (of pieces already placed) on the board

There are two good choices for checkpointing places: at the beginning of each cycle when a new piece appears at the top, or each time a piece moves. The latter option introduces much overhead into the game, and the former would be a natural choice. But since it can be argued that the latter is "better" (better granularity), and it does not kill performance, it was chosen.

The worker loop

The main loop of Tetris does practically everything from user input monitoring to moving the piece to checking if the piece fits to bumping the score. Therefore it makes a very good candidate to be registered as the worker function. The only thing we need to do is take the loop out of *main*(), and place it into its own function. This is done because we need to call the main loop directly if we wish to do a restore from a checkpointed situation. If the program would go through *main*() also when restoring from a checkpoint, it would initialize its runtime state to zero, and defeat any purpose of Hot Spare checkpointing.

In addition to moving the main loop into the worker function, we also move some screen-related initialization there. This is done because we need to set up the screen also on the spare if the program

execution is handed over. Normally the Hot Spare Library provides routines for all necessary state-saving functionality, but since it was written with daemons, not interactive applications, in mind, it does not provide routines to save screen state. Nonetheless, this serves as an example of the fact that when the Hot Spare Library does not provide the necessary routines, it is possible for the application to define them in its own domain.

Finally, the code that takes care of returning screen setup to a sane state needs to be moved into the worker function after the main loop. The spare program has no knowledge it should fall back to *main*(), since the worker function was called directly from the Hot Spare Library, and will exit after returning from the worker function.

Saving state

Since this version of Tetris was written in the early 90's, it was written like most programs of old: state is kept in the data segment as global variables. This is unacceptable for us, since we need to store critical data in areas which will be included in the checkpoint.

The task of moving the information from the data segment to checkpoint-safe memory is a fairly simple one: we simply "collect" the state from global scope in the source module `tetris.c`, and create `struct tetstate`, in which all the variables essential to the state are placed. This structure is added to the checkpoint memory area when Tetris is initially started. All the references to the state variables must be fixed to point inside the checkpoint-safe structure. It can be accomplished either by using cheap tricks with the preprocessor (`#define`) or by a simple search-and-replace operation with a text editor or shell utility. Most of the time taking the effort to do an actual search-and-replace pays off and avoids unwanted and weird side-effects, although the bulk of the differences may then amount to changes in variable referencing.

Normally multithreaded programs avoid using global state and pass the context of the call as a parameter. In this case the program state will most likely already be readily contained, and no modifications such as with Tetris and other older non-threaded programs should be required.

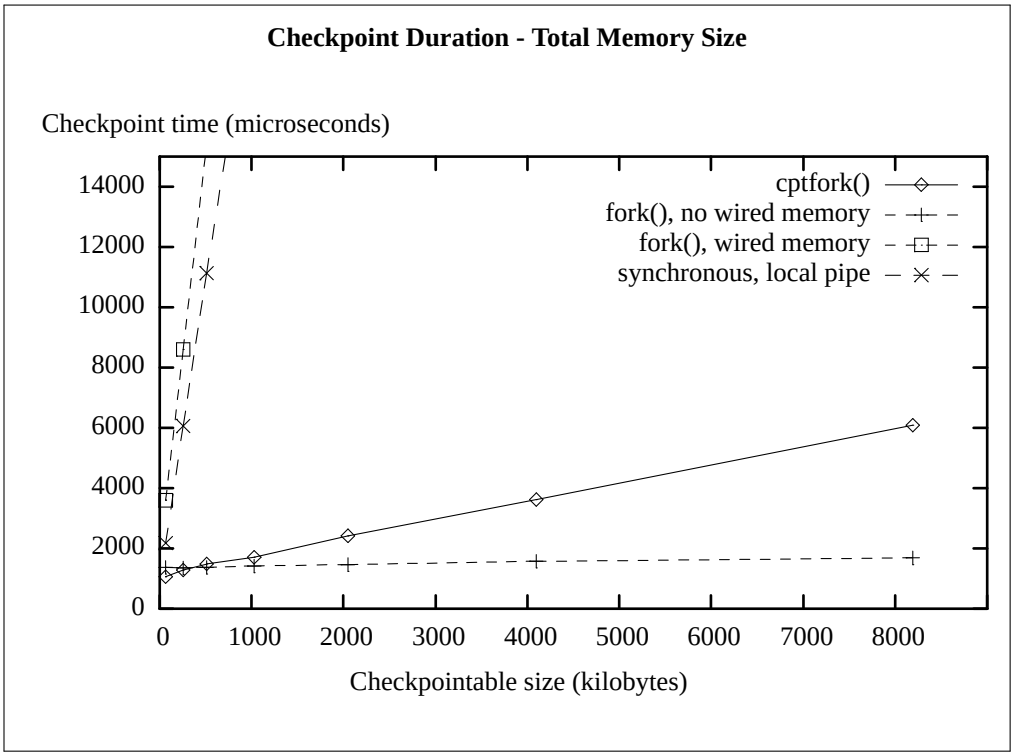
In addition to the memory and worker "thread" state, the game registers a few signal handlers. Although they could be registered via the *hs_sigreg*() facility, they are an integral part of the screen setup code. Since we run that code anyway, the signals get proper treatment even without explicitly including them in the checkpoint.

Conclusions

Adapting Tetris from the BSDgames package for application-driven checkpointing was a simple job. It was accomplished in just a few hours time after first looking at the source code. The factors that amounted to the ease of checkpointing adaption were the limited size and instantly clear intuition on what to checkpoint. The non-threaded programming approach and consequent lack of state grouping were the only difficulties encountered.

4. Performance

In this chapter I present some key benchmarks. Since we are interested in the performance of the checkpointing module and less interested in the operating system and network performance, the checkpointing process does not transmit the checkpoints anywhere for restoration. Checkpoint data is simply written into `/dev/null`. All the tests were run on a 300MHz AMD K6-2. It is not "current" technology, but this work is not targeted for any specific machine, so it is a safe choice.



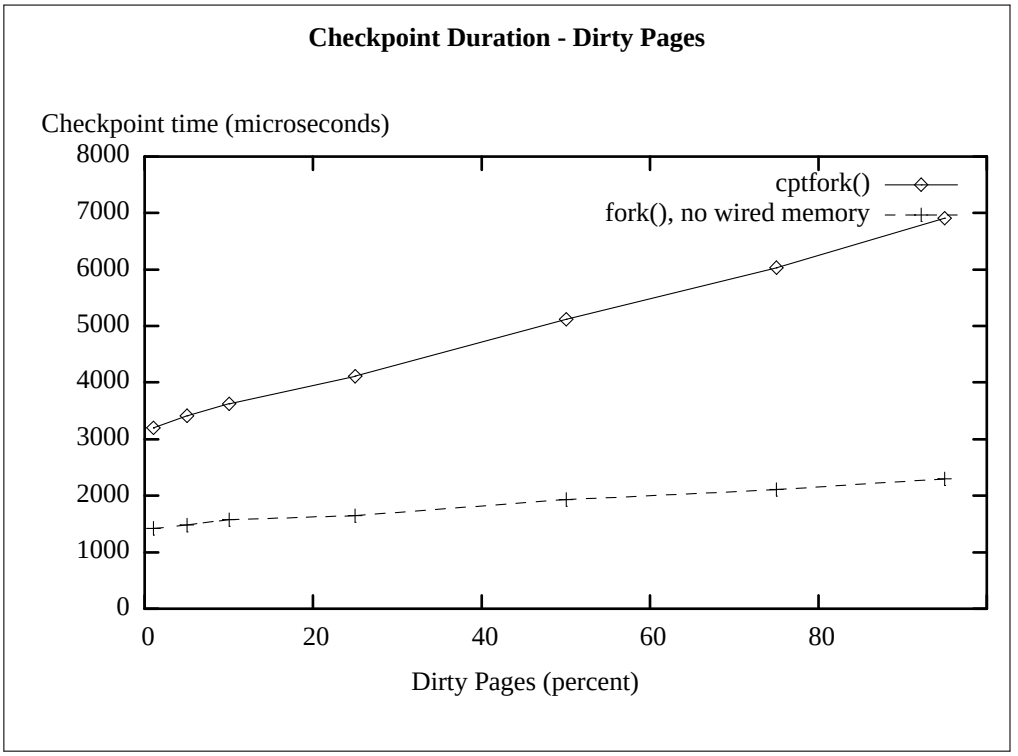
The first test examines how the checkpointing time from the application point-of-view is influenced by the amount of checkpoint-safe memory registered. This amounts to the time in between calling *hs_cpt*() and returning from the function. Between checkpoints the parent modifies 10% in sets of four contiguous pages and sleeps for one second.

For taking a synchronous checkpoint in application context the system was modified somewhat. Writing the checkpoint to `/dev/null` also in the case of a synchronous checkpoint would be unfair, since transfer speed to the spare is the limiting factor. For application context synchronous checkpoints the preferred way is getting the checkpoint contents as quick as possible somewhere else, so that the application can continue with its normal tasks. For benchmarking purposes I opted to write to a local pipe, since it is faster than transmitting the data over the network. In this case the other end of the pipe just reads data to empty the pipe buffer, but in real life it would naturally also take care of making sure the data reaches the spare units.

The results are what was expected. When dealing with wired pages, plain *fork*() is hideously expensive. This is because it copies all wired memory to the child process. As you can see, the results go "off the scale" fairly early.

Without wiring pages *fork*() is the cheaptest alternative from the application point-of-view. It starts out slightly heavier base cost than *cptfork*(), but quickly catches up and follows an almost constant trend after that. The difference in base cost can be accounted to the fact that *fork*() marks all regions copy-on-write, while *cptfork*() shares most of them. However, the price to pay for doing a full asynchronous checkpoint with *fork*() is of course the amount of data to be sent over to the spare.

The cost of doing *cptfork*() is very close to linear with an added base cost for



doing common tasks required when *fork*()ing. The linear cost can be explained by having to go through all pages marked checkpoint-safe and checking them for modification information before allowing the parent of the *cptfork*()ing process to return.

Taking a synchronous checkpoint by writing to a pipe starts out about as cheap as the non-wired *fork*()ing alternatives, but exhibits high costs when the checkpoint size is even slightly increased.

Varying Amount of Dirty Pages

To see how the amount of dirty pages affects checkpointing cost from the application perspective, a test which modifies a varying number of pages was run. The test reserved 4MB of memory and did modifications in sets of four contiguous pages. In the case where 95% of the pages were modified, 25 contiguous pages were used instead to make the test runnable. The main purpose of this test was to see if it becomes clearly cheaper to take a full checkpoint instead of using the *cptfork*() approach at some point.

The asynchronous approach exhibits a clearly linear trend in addition to the *cptfork*() base-cost. The same can be said about normal *fork*(), except that the linear coefficient is much smaller in the latter case. I am not totally sure where the linear coefficient comes from, but my educated guess is that accessing pages influeces various caches in the system. The *cptfork*() case takes more performance penalty from this, because it does more lookups than a normal *fork*(). If nearly all pages are modified, *cptfork*() is 5ms slower than plain *fork*(). This difference is significant, since the longer the checkpoint memory area is locked, the longer other threads can be blocked[7].

[7] Checkpoints are taken with important areas locked by the application to avoid them being in a "bad state" in the checkpoint.

Analysis of Results

Ultimately we wish to know which approach is the cheapest for each given situation. It is clear that application context checkpointing is not worthwhile unless there is extremely little data to checkpoint, perhaps only a page of memory or so[8]. Once the checkpoint size gets into the range of tens of kilobytes and beyond, asynchronous checkpoints stall the application for much less.

While doing full asynchoronous checkpoints employing *fork*() is a win from the point-of-view of the application, that is only half of the truth. The cost of transferring the checkpoint to spare units becomes a huge factor for applications which wish to register a myriad of memory, but only modify it seldom. This limits the granularity of full checkpoints. Available bandwidth will most likely be saturated by information which remains the same from one checkpoint to another.

Looking at all the graphs presented in this chapter, it is clear that *cptfork*() is the most performant alternative as long as there is enough memory in the checkpoint range, and if not too big a portion of that memory space is modified in between checkpoints. After reaching a high enough modification percentage a full checkpoint becomes cheaper. Unfortunately we do not know the amount of dirty pages before making the decision to checkpoint using *cptfork*(). After taking a hit from the overhead of doing *cptfork*(), it is too late to change our mind.

We could address the problem presented in the previous paragraph by recording page modification information already when the page is modified. Since our checkpointable memory ranges are marked copy-on-write, the operating system takes page faults to copy pages which are being modified. In addition to gaining knowledge on modification statistics before making any expensive decisions such as *cptfork*(), there would be other benefits.

- There would be no need to do a lookup for all the pages in checkpoint memory ranges during *cptfork*(), as the modification information could be already recorded in a simple form, such as a bitmap. This would effectively cut down the checkpoint-time from O(total_pages) to O(pages_modified).
- The scheme would also work on platforms which do not have page modification information in their MMU.

[8] Assuming of course small, kilobyte-sized pages. Megabyte-sized "large pages" are right out.

5. Conclusions

This work set out to investigate the possibility of using a checkpointing approach for Hot Spare High Availability in environments where the application is time-critical and freezing it for an arbitrary period during execution for taking the checkpoint is not acceptable.

The key idea in the approach was to make checkpointing the responsibility of the application, since it best knows what it is doing with its state as opposed to an external facility, which must treat all data as opaque. The efficiency of the architecture was enhanced by adding a kernel component, which serves the application-level library by providing information on which pieces (memory pages) have changed since the last checkpoint.

If the application itself contains vast amounts of redundant state, using application-guided checkpointing to carve out the necessary bits will increase performance dramatically. Incremental checkpointing will enhance performance more and more as the ratio of modifications between checkpoints to the entire checkpointable memory area decreases.

The Hot Spare Library was written to be both portable and flexible. It provides most of the functionality necessary for standard applications, but since checkpointing is application-driven, the application itself is free to handle anything else it needs to checkpoint.

The biggest part of the work for someone who wishes to use an application-driven scheme is of course adapting the application. It was shown that for a small application the work was just a matter of hours. For a large application, the time depends greatly on how familiar one is with the application before starting the modification task, and how the application was written. The task varies from ‘‘trivial’’ to ‘‘impossible without rewriting the entire application’’, and it is impossible to give an accurate estimate without knowing the particular application.

This work did not address the problem that unfortunately makes the approach invalid for most network services: migrating applications which depend on a persistent TCP connection is not possible[9]. There are two ways to fix the problem: either teach the application and protocol that the connection may be broken if migration takes place, or modify TCP on both endpoints to cope with migration. Unfortunately, neither approach is non-intrusive from several perspectives, and the modifications are far from trivial, either logistically or technically.

As a concluding remark it can be said that the application-driven approach was found to be a working one, and under the right circumstances and right software it can be an extremely attractive option for providing Hot Spare High Availability.

[9] I do not know if it is any condolence that the TCP problem makes just about any checkpointing approach inapplicable.

References

1. Antti Kantee, *Using Application-Driven Checkpointing Logic for Hot Spare High Availability,* Master’s Thesis, Helsinki University of Technology (September 2004).
2. Yi-Min Wang, Yennum Huang, Kiem-Phong Vo, Pi-Yu Chung, and Chandra Kintala, *Checkpointing and Its Applications,* pp. 22-31, 25th International Symposium on Fault-Tolerant Computing (June 1995).
3. William R. Dieter and James E. Lumpp, Jr., *A User-level Checkpointing Library for POSIX Threads Programs* (1999).
4. James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li, *Libckpt: Transparent Checkpointing under Unix,* Winter Usenix Conference (January 1995).
5. C. Cranor, *Design and Implementation of the UVM Virtual Memory System,* PhD thesis, Washington University (August 1998).
6. S. R. Kleiman, *Vnodes: An Architecture for Multiple File System Types in Sun UNIX,* pp. 238-247, Summer Usenix Conference, Atlanta, GA (1986).
7. Sun Microsystems, “java.io Interface Serializable,” *Java2 Platform, Standard Edition, v1.4.2 API Specification.*
8. Guido van Rossum and Fred L. Drake, Jr., “pickle -- Python object serialization,” *Python Library Reference.*