CheckSync: Transparent Primary-Backup Replication for Go Applications Using Checkpoints

by

Nicolaas M. Kaashoek

B.S. Computer Science and Engineering
Massachusetts Institute of Technology, 2020

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author . . . . . .
Department of Electrical Engineering and Computer Science
May 20, 2021

Certified by . . . . . .
Robert Tappan Morris
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

CheckSync: Transparent Primary-Backup Replication for Go Applications Using Checkpoints

by

Nicolaas M. Kaashoek

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Many distributed systems have singular, mission-critical components. The MapReduce coordinator, lock servers, etc are all examples of such components. Due to their importance, they require high availability and fault tolerance. The most common way to achieve this is through the use of replicated state machines, an approach in which the application is replicated across multiple machines. There could be as few as two in a primary/backup arrangement, or more to reduce the risk of downtime. Each instance starts in the same state, and then advances to new states in the same order. This allows for easy failover to one of the replicas in case the primary machine fails.

The use of replicated state machines, however, requires an application to expose the correct stream of operations to ensure that each machine ends up in the same final state. This abstraction is not well-suited to all applications, as it can't support multithreading and can add extra complexity for application developers. This thesis proposes CheckSync, a protocol for achieving high availability and fault tolerance via the use of checkpoints. CheckSync is designed with transparency as a primary goal: applications require little to no modification to use it. It achieves this by checkpointing the memory of an application and replicating that state from primary and a backup. Upon failure, the backup resumes from the checkpoint and continues running.

CheckSync's transparency sets it apart. Unlike the operation stream required for replicated state machines, CheckSync doesn't place constraints on the design of the application. It can suspend and capture the memory of Go applications without knowledge of the specifics of the application, as well as restore them on the backup. This is accomplished through careful analysis and recreation of the application's memory space, as well as efficient transmission of the checkpoint files to minimize performance overhead. CheckSync is evaluated with three different applications, and supports all three without any changes to their code.

Thesis Supervisor: Robert Tappan Morris
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

Robert Morris for his invaluable advice, gentle guidance, and precise feedback in all aspects of research.

Burton 4, who made every day at MIT just a little brighter.

Katharina and Yaateh, for their company during a difficult year in quarantine.

Katrina LaCurts for her joy and kindness throughout my time at MIT.

My family for their patience, love, and endless support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Distributed systems often have numerous smaller components, each responsible for a specific part of the system's operation. One or more of these components may be critical to the system's operability; if it were to go offline, the entire system would come down. Therefore, it is important that these critical components be able to tolerate and recover from failures, and to do so quickly. They must be fault tolerant and highly available.

This thesis explains the motivation behind CheckSync, a system designed to deliver fault tolerance and availability by capturing application memory, explains its design and key features, and discusses how CheckSync was implemented.

This thesis also presents an evaluation of CheckSync on three different applications: the aforementioned MapReduce coordinator, an application with a long-running mathematical computation, and an in-memory key/value store. These three use cases show that CheckSync is not only operational, but presents a practical option for achieving fault tolerance.

1.1 Making Applications Fault Tolerant is Intrusive

While fault tolerance has been an essential aspect of distributed system design[21, 9] for years, actually achieving it can be a challenge. Even in the simplest case where only two machines are used, a great deal of care must be taken. In this case, one machine acts as the primary and the other as the backup, and upon primary failure the system switches to the backup. However, the application must start on the backup in a near-identical state as when it failed on the primary for the process to work. Ensuring that this is the case requires careful coordination and synchronization between both parties in the system.

Replicated state machines[30], a common solution for fault tolerance, create a set of replicas and start them with the same initial state. From there, they replicate a stream of operations across all the replicas, which apply them in the same order to ensure that they end up at the same final state. Consensus protocols like Raft[26] and Paxos[20] are used to agree on the order despite failures. An early version of VMWare's approach to fault tolerance for virtual machines[29] and chain replication[34] are other examples that fit the same general approach.

As many applications want fault tolerance, it is important that any scheme that provides

it be able to support a wide variety of applications. The operation-based replication used in replicated state machines fails to deliver on this goal. Not all applications have a neat abstraction that fits a replicated operation log. Consider, for instance, the coordinator in MapReduce[17]. The coordinator needs to be highly available, and needs to track the state of the system, but doesn't truly have operations it needs to respond to. The general issue in identifying and exposing the right set of operations can make the design of applications that use state machine replication a difficult process.

Additionally, using a replicated series of operations necessitates that the operations be applied in the same order on all the replicas. If they aren't then the final states will not be identical. For this reason, replicated state machines cannot handle multithreaded execution. In such cases, execution is no longer deterministic, making it impossible to predict the order the operations will be applied in. The lack of support for multithreading further restricts application developers, and in combination with the complexity introduced by replicating operations makes that approach poorly-suited for delivering a transparent system for fault tolerance.

1.2 CheckSync: Transparent Fault Tolerance

CheckSync is a memory-based replication scheme for providing high availability and fault tolerance for mission-critical applications. By taking a checkpoint of the memory of an application mid-execution, CheckSync avoids the problems that replicated state machines have with general-purpose application support. Rather than relying on applications to expose the operations that modify the memory, CheckSync instead captures memory directly. This also allows it to take checkpoints of multithreaded code.

There are two components to CheckSync. First, the checkpoint/restore library. This handles both the checkpoint and restore operations. CheckSync has two different libraries. One uses CRIU[3], an existing userspace checkpoint/restore tool. The other consists of a modified Go runtime and a Go library. The"internal" checkpointing tool can make use of information unavailable to an external tool like CRIU, which creates opportunities for optimization and supports different levels of consistency. The internal library is currently unable to support all the features of applications that CRIU can, so CheckSync is evaluated using mostly the external checkpointing tool instead.

CheckSync's second component is the manager, which handles the transmission of checkpoints and monitors the state of the system. The manager is responsible for initiating failover and triggering the restoration process.

CheckSync operates by first suspending all the threads of the application, which executes only on the primary, and dumping the memory in use to disk in the form of image files. The manager then makes these images available to the backup, either by direct copy or by uploading them to a known location. When the manager on the backup detects a failure on the primary, failover is initiated. The backup reads the image files, maps them into memory correctly, and then resumes execution from exactly the point at which the checkpoint was taken.

Performing the checkpoint operation is complex. Because CheckSync supports multithreaded applications, it must be able to suspend all the threads executing concurrently with

the checkpointing operation. If any of these threads modify memory while a checkpoint is occuring, the state the backup will resume to may be inconsistent with what memory looked like on the primary. CheckSync's internal checkpointer deals with this by making use of the runtime's knowledge of all running threads and its ability to manipulate them.

The restoration process presents its own problems. Memory must be restored on the backup by some process other than the application being replicated. However, execution must return to the application once memory has been setup correctly. This is troublesome as it means that the restoring code must map the application into its own memory space without overlap, and then correctly restore the register set in such a way that execution seamlessly jumps to the application. CheckSync addresses this through careful management of memory and the mapping of pre-compiled code to non-intersecting memory locations.

The manager also needs to be careful. It cannot suspend an application for too long lest the performance overhead becomes too great to ignore. Additionally, while consensus protocols and some other operation-based replication approaches provide strong consistency, this is difficult for CheckSync. It has no knowledge of the underlying application, so it is always possible that a change happens between a checkpoint and a failure. That change will not be reflected upon resumption on the backup. For this reason, CheckSync provides timeline consistency. However, the internal checkpointer also provides applications with the option to force strong consistency instead.

1.3 Thesis Goals and Contributions

In summary, this thesis presents an alternative to existing replicated state machine approaches for fault tolerance using checkpoint/restore. CheckSync is a primary-backup style of fault tolerance that relies on the primary periodically checkpointing the running application and sending it to the secondary, which can resume from the checkpoint in the case of failure. CheckSync has been implemented and evaluate with two key goals:

- **Transparency:** CheckSync must support a variety of Go applications, which can be multithreaded, without incurring significant development costs on the part of developers.
- **Performance:** CheckSync must be able to meet performance goals such that the overhead of taking and sending checkpoints do not significantly impact the applications it is designed to work for.

The contributions of this thesis are:

- **Multithreaded Checkpointing**: The design and implementation of a Go library and corresponding modification to the Go runtime to support checkpointing of Go applications.
- **Checkpoint Replication and Restoration**: The design and implementation of a Go library to send checkpoints from a primary to a backup and restore from those checkpoints in the case of primary failure.

- **Evaluation on Sample Applications**: The implementation and evaluation of three applications using CheckSync, including comparisons with other techniques for achieving fault tolerance.

Although CheckSync achieves its main goals, the current implementation has some shortcomings:

- Both the internal and external tools have problems. The internal tool doesn't support file or network I/O, while the external tool has no option to deliver strong consistency.
- The size of the checkpoints produced by CheckSync can grow large as the state of the application increases. This means CheckSync performs poorly for applications with large state.
- CheckSync is a primary/backup system. It doesn't offer the same level of availability as Raft or other systems that use more than one replica.

In future work we hope to address these challenges by improving the internal checkpointer to include features of the external one, and then use the internal tool's runtime integration to produce smaller checkpoints.

Chapter 2

Background and Related Work

Replicated state machines is one of the most common approaches to achieving fault tolerance. An application is replicated across multiple machines, which all start with the same initial state. The application then transitions from the start state to successive states as its execution progresses. Replicated state machine protocols ensure that each replica applies the same transitions in the same order, ensuring that each replica ends up in the same final state.

2.1 Consensus Protocols

One of the most common classes of replicated state machine protocols are consensus protocols such as Raft and Paxos. They are often the go-to approach for achieving fault tolerance. Both protocols accept operations from the application, and come to a consensus about the order in which all the operations must be applied. In Raft this is done by having the replicas vote for a leader, who then proposes and commits the operations. Once an operation is committed it is durable.

An overview of this process in Raft is shown in Figure 2-1. Operations are added to a shared log by the leader. The operations are then replicated to all the backups, which also apply them. This makes the replicas' state identical to the leader's. This approach requires that the application feed the operations to the underlying protocol, which can make using consensus protocols difficult.

Not only do applications need to understand the protocol enough to know what needs to be replicated, but they also have restrictions on their design. Because the leader and its followers must apply every operation in the same order, supporting non-deterministic execution is impossible. If a system like Raft or Paxos were to process requests in parallel, there would be no way to guarantee that this invariant would hold.

An illustration demonstrating why multithreading is a problem for Raft in particular is shown in Figure 2-2. Operations enter the protocol in parallel, and are applied in parallel. This results in the order shown in step 3. However, when replica 2 reads the operations from the log, it applies them in a different order due to them racing in parallel. This means that the final state in step 7 is different than the state in step 3.

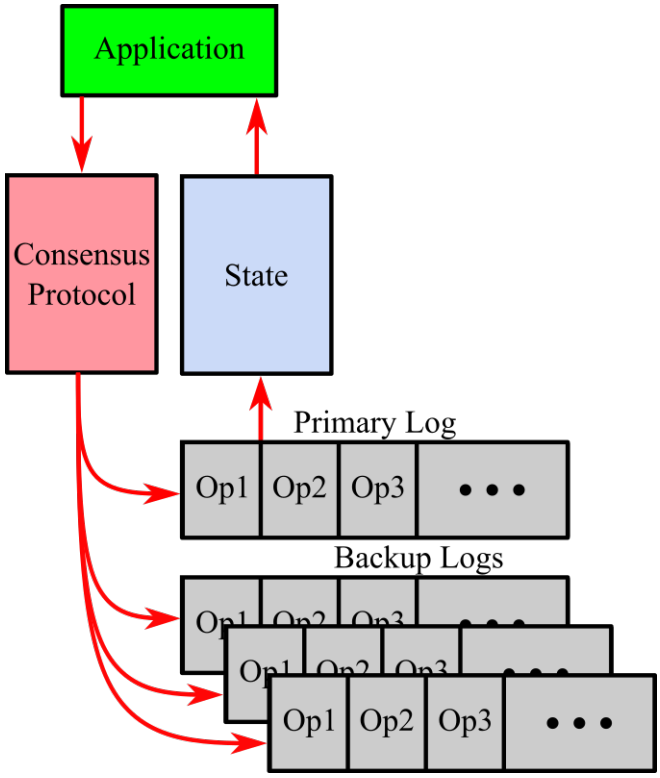

Figure 2-1: Replication in Raft. The application feeds updates to the consensus protocol, which coordinates the replication of the operations to all the logs. The replicas then read from the logs to construct the state, which the application can then use.

2.2 Operation and Memory-Based Replication

Another example of replicated state machine design was designed by VMWare with the goal of supporting fault tolerant virtual machines[29]. In that system, two machines are started running the same VM. The hypervisor on one machine, the primary, intercepts all input (network packets, keyboard input, etc.) to the VM, and records those operations in a log. This log is then replayed on the second virtual machine to reconstruct the state of the virtual machine. However, this had major performance drawbacks, and, like consensus protocols, prohibited multithreading.

The performance problems and lack of multithreading led VMWare to design a new class of systems for fault tolerance, which use live migration[13] instead of replicated state machines. Examples of systems that use this approach include Remus[16] and vMotion[7], which provide high availability and fault tolerance for virtual machines running in a cloud computing environment.

In Live Migration, while a VM is executing, the hypervisor captures a complete image of the memory of that virtual machine, and sends it to a backup machine which can then use that image to restore and resume execution of the virtual machine from the exact point in time that the image was captured.

This approach is illustrated in Figure 2-3. As all of memory is being replicated, live

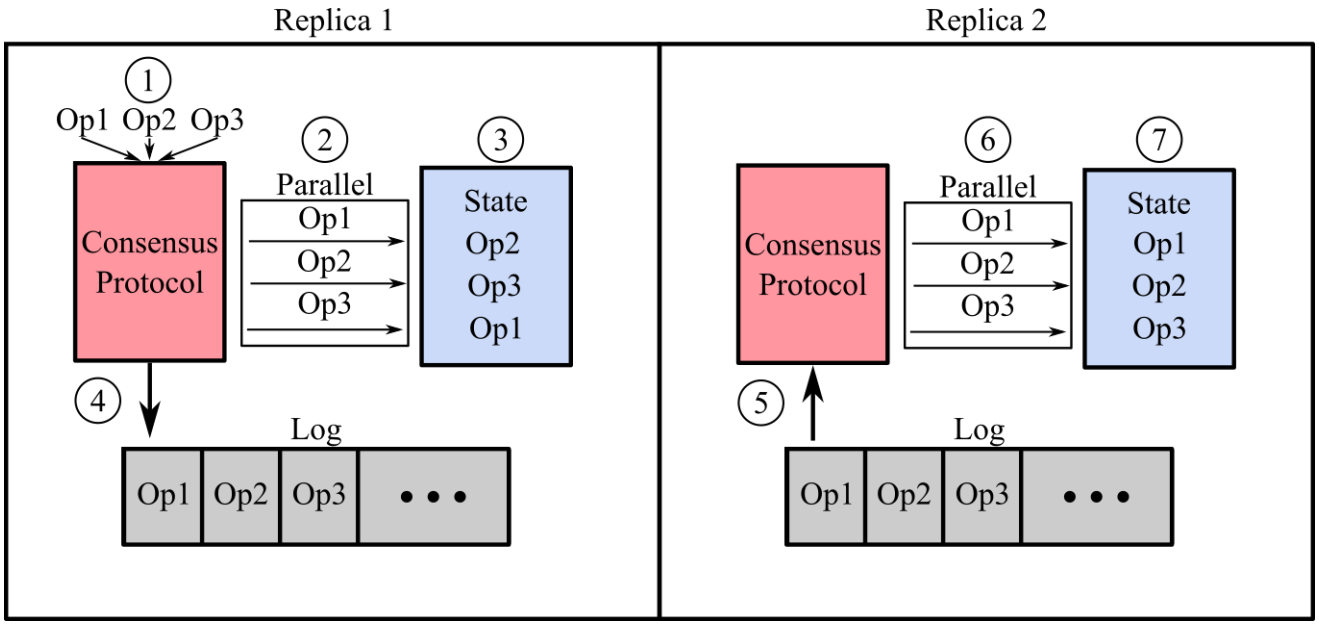

Figure 2-2: How Multithreading causes Failures in Raft

migration allows vMotion and its competitors to capture and restore virtual machines that have multiple threads. Also, because the virtual machine is captured rather than the application, it is transparent to applications.

The difference between replicated state machines and virtual machine live migration can be broadly categorized as operation-based replication and memory-based replication. Replicated state machines rely on the replication of a stream of operations that modify state, with each replica applying them in the same order to end up at the final state. Memory-based replication, however, copies the actual state of the application in the form of memory, rather than reconstructing it using operations.

CheckSync is an example of a memory-based replication scheme, and also operates as a primary and a backup. However, it differs from VM migration in how lightweight it is. While live migration does handle the problems with replicated state machines, it also requires that an application be running in a VM. This can be wasteful, as the application may be small, and live migration will capture the whole virtual machine no matter what. CheckSync deals with this by capturing only the application's state.

2.3 Multithreaded Fault Tolerance

Attempts have been made to address the lack of support for multithreading in replicated state machines as well. Sometimes through modifications to existing protocols[38], or wholly new replication schemes[18]. However, none of these approaches have seen widespread adoption, and are not transparent.

Another class of attempts to allow replicated operation approaches to provide fault tolerance even in the face of concurrent execution is the use of deterministic execution[15,

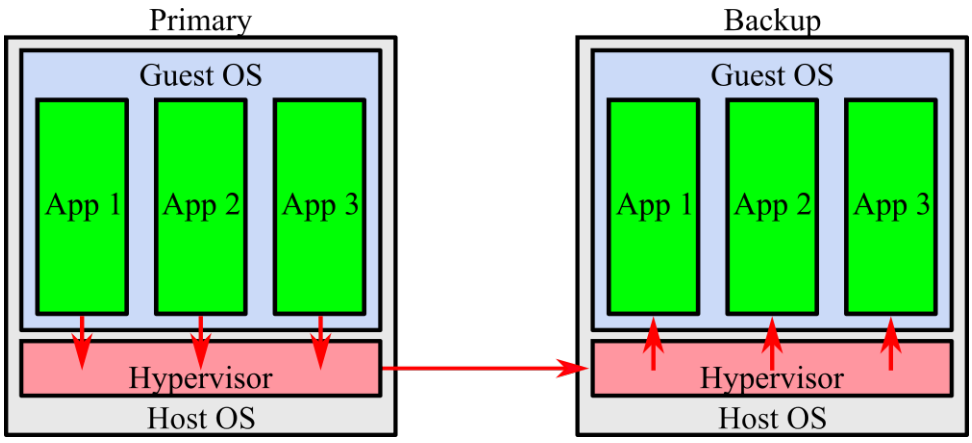

Figure 2-3: Virtual machine live migration

25, 8, 10]. These schemes create a consistent ordering of operations even if they happen in parallel, which addresses the core problem replicated state machines have with multithreading, but these schemes have considerable overhead for enforcing determinism. Again, such techniques have not seen much adoption.

2.4 Checkpointing

CheckSync works by taking checkpoints of the memory of a running application. Checkpointing has a long history in multiple areas of computer systems. One such area is in databases, where checkpointing is used for a variety of purposes [23], mostly as a way to improve recovery times when failure occurs. By checkpointing the operation log, less replay needs to be done and recovery times increase. In memory databases also make use of checkpoints, using it as a way to snapshot the state of the database in case of failure[27]. These uses of checkpointing are in a similar vein as CheckSync, but are designed specifically for databases rather than general purpose code.

Operating systems have also made use of checkpoint and restart patterns to improve recovery times in different contexts. One such example is improving update times[19]. Generally, these schemes are designed to allow applications on an operating system to tolerate hardware failures[11, 12]. There has also been some use of checkpointing for real-time operating systems to improve fault tolerance for such systems [28].

One of the more common environments where checkpoints are used to great effect is in high-performance scientific computation. For long running compute jobs, periodic checkpoints can be used to prevent the loss of large amounts of work in the face of a crash[32, 35]. These checkpointing schemes can support parallel execution as well[22, 31] Checkpoint support has even been built into the Charm++ runtime specifically to support such applications[37, 36], however these approaches are specifically designed to reduce the amount of work that needs to be redone after a failure. They are not concerned with maintaining high application availability, which makes them poorly suited for the kind of applications CheckSync seeks to support.

Chapter 3

Design

CheckSync tackles the problem of improving application availability by first *checkpointing* the application, and then *synchronizing* the checkpoint with a backup which can then restore from that checkpoint in the face of failure. CheckSync's design is intentionally lightweight and poses a small development load for the application programmer.

This chapter presents an overview of CheckSync's design (§3.1) and its components: the manager (§3.2) and the checkpointer (§3.3). The manager is responsible for initiating the checkpointing process, sending the checkpoints and keeping track of the state of the system. The checkpointer is responsible for taking and restoring from checkpoints. CheckSync has two different modes, one delivers timeline consistency while the other provides strong consistency (§3.4).

With the driving motivation to stay lightweight and keep work simple for developers, CheckSync cannot rely heavily on cooperation from application developers. However, we do assume that application developers write correct, thread-safe code as CheckSync provides no extra correctness properties to the system. If the application code is incorrect or inconsistent, CheckSync will not improve availability. Also, CheckSync places restrictions on what applications can do. These restrictions are discussed in §3.5.

3.1 Overview

CheckSync is a primary-backup system designed to deliver high availability for applications. It supports a single application written in Golang[5], and replicates the state of this application by periodically checkpointing its memory and sending that checkpoint to the backup. CheckSync uses Go to take advantage of runtime integration to when checkpointing. The primary runs the application, while the backup holds copies of the most recent checkpoints sent by the primary. The backup does not run the application unless a failover occurs.

CheckSync is designed to be deployed in a cluster environment to provide fault tolerance for components of a larger system. One example component that CheckSync supports is the MapReduce[17] coordinator.

Figure 3-1, illustrates the workflow for CheckSync, which can be described as follows:

(1) Starts a primary manager and backup manager on the primary and backup respectively
(2) The primary manager starts the application
(3) The primary manager checkpoints the application
(4) Failure occurs on the primary
(5) The backup manager detects failure after a lack of heartbeats
(6) The backup manager initiates the restore operation on the most recent checkpoint
(7) The application being replicated resumes execution

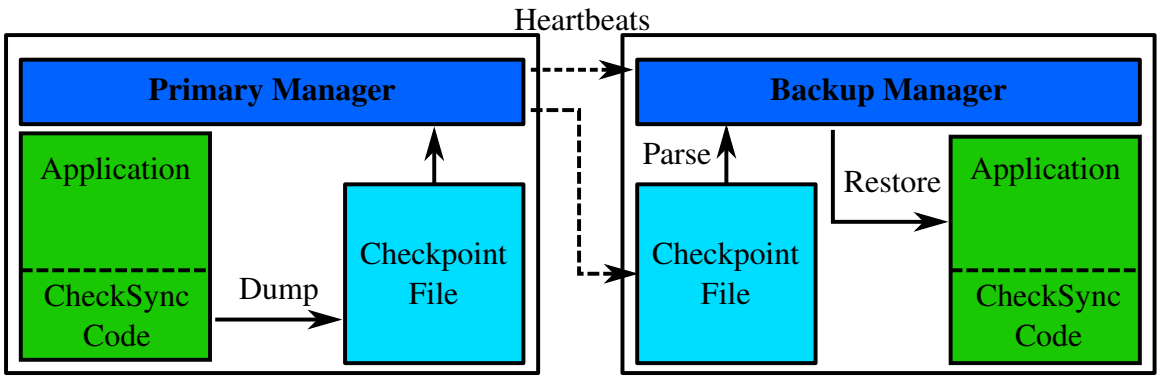

Figure 3-1: CheckSync design overview

3.2 Manager

The manager runs as a separate process independent of the application. There are two modes for the manager that determine its operation: primary manager, and backup manager. Both the primary and backup instances of the manager must be started before the system can actually start working, as they work in conjunction to keep the system running.

The primary side of the manager has two roles. First, it does the work of checkpointing the application process. The manager starts and periodically initiates the checkpointing of the application that CheckSync is being used to replicate. It configures parameters including the frequency of the checkpoints as well as the mechanism used to do the checkpointing, as discussed further in §3.3.

The second role of the primary manager is to replicate the checkpoints produced from the application to a location reachable by the backup. This is done by saving the checkpoint to a storage system like S3 or Azure, which ensures that the files are always available to the backup when the primary fails.

In addition to storing the checkpoints, the primary manager also sends periodic heartbeat messages to the backup to ensure that the backup is aware that it is still alive. These heartbeat messages and all other forms of communication between the two sides are simple RPCs using the gRPC framework.

The backup manager does very little work unless a failover occurs, during which it must take care of some configuration to get the system in proper order. This is discussed further in §3.2.1. Outside of failover situations, the backup monitors the status of the primary

via heartbeats messages. It also replies by informing the primary about the most recent checkpoint it received.

In a cluster deployment, the decision about which machine is the primary and which is the backup would be handled by a some external configuration agent, likely backed by a consensus protocol, to ensure that network splits don't create two primary machines.

3.2.1 Failover

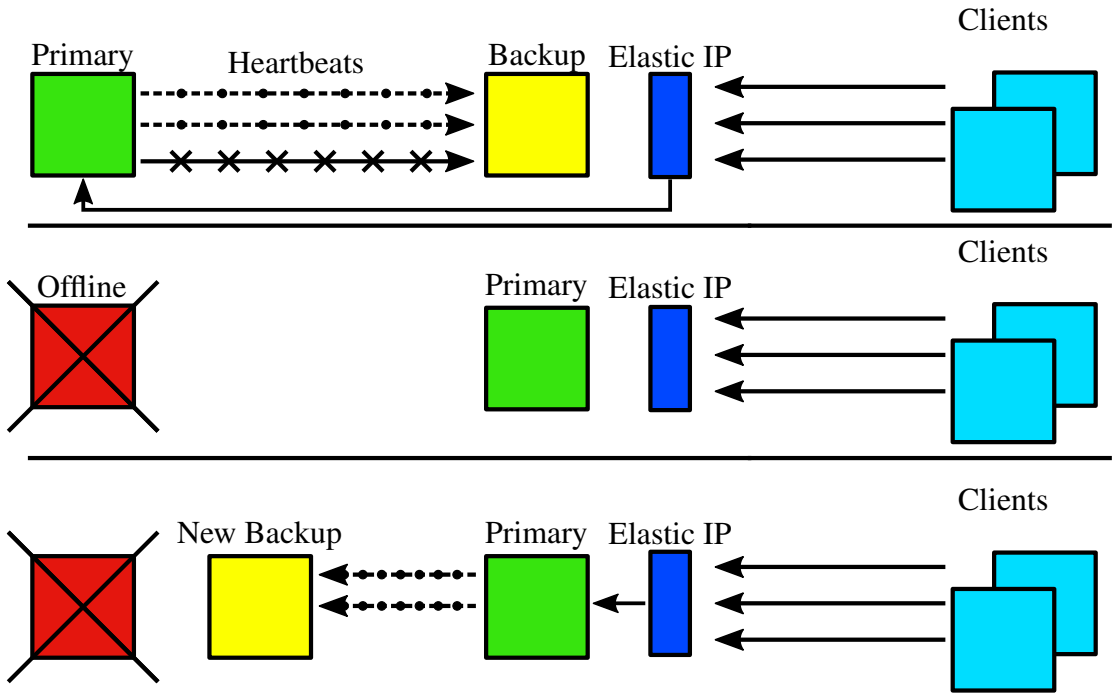

Figure 3-2: How CheckSync recovers from failure. Failure is detected due to a lack of heartbeats, the elastic IP is reassigned to the backup, which then becomes the new primary for clients to talk to.

Under normal operation, the application runs entirely on the primary, leaving the backup idle. When a failure occurs that causes the primary machine to go offline, the backup has to pick up from where the primary left off. The entire failover process is illustrated in 3-2.

First, it must detect that the primary is offline. This happens when the backup notices that it has missed five heartbeat messages, although this number can be configured for more rapid response to failure.

Then, the backup manager initiates the restoration process as described in §3.3.4. Once the restored application is up and running, the backup manager needs to make sure that any potential clients or other entities on the network communicating with the application can do so. To do this, it needs to update the cluster configuration information so that incoming traffic can reach the new primary.

The exact mechanism used to this varies depending on the cloud provider used to run CheckSync. In its test deployment, this was done by modifying an elastic IP address pointing at the application when a failover occurs. For the backup all this meant was running

a pre-configured AWS lambda function to modify the line in the cluster configuration that controls the destination of the elastic IP address.

The failover process also has consquences for clients communicating with the application. During the failover, the application will be offline. For this reason, it is important that the clients be constructed to attempt to reconnect to the application if their connections fail. Additionally, because CheckSync isn't strongly consistent (§3.4), the application must be able to handle duplicate client requests.

3.3 Checkpoint/Restore

CheckSync has two different approaches to checkpointing and restoration. The first, termed the "external" checkpointer uses a separate process that runs CRIU[3] to take the checkpoints and to restore from them. The second, the "internal" checkpointer lives inside a Go library and the runtime and uses runtime mechanisms alongside other tools to take the checkpoints. The restoration process for the internal checkpointer is a lighter-weight version of CRIU's restoration phase.

CRIU is an existing tool used to take and restore from checkpoints entirely in userspace. CRIU is not designed to provide high availability, and is instead often used to checkpoint containers such as Docker to improve startup/bootstrap times for creating new containers. However, as CRIU supports a number of features (see §3.5) that the internal method currently doesn't, it is an ideal tool for demonstrating the viability of checkpointing for achieving fault tolerance and availability.

Both the internal and external approaches have their own advantages. Using CRIU enables checkpointing of a wider variety of applications. Unlike CheckSync, CRIU is not specific to Go. Additionally, CRIU has the ability to take checkpoints even if an application is making a system call, in the midst of writing to a socket, and has open file descriptors. The internal checkpointer could be modified to support these features as well, but as currently implemented it cannot do so.

However, the internal checkpointer has its own advantages. First, because it lives in the runtime, it has access to more information about the application than CRIU does. While the current implementation doesn't take advantage of this, some simple extensions include checkpointing only the heap, stack, and data segments rather than the entire application. This would reduce both checkpoint size and overhead. Further extensions could take this to more extremes as well, such as tracking memory changes to dump out only the actual bytes that have changed.

Another advantage of the internal checkpointer is that with cooperation from the application it can deliver strong consistency (see §3.4).

As CRIU is extensively documented, the remainder of this section describes only the internal checkpointer.

The internal checkpointer is inspired by CRIU and is implemented as a small set of changes to the Go runtime, with the majority of the code living in a separate library. The internal CheckSync checkpointer operates entirely within the same process as the application.

3.3.1 Checkpointing

At a high level, the checkpoint process is divided into two phases, illustrated in Figure 3-3.

1 - **Suspension**. During this phase, CheckSync takes care to suspend the running application in a safe state so that it can be checkpointed without risk of memory being modified in the middle of the checkpoint.
2 - **Dump**. Here CheckSync captures the state of the application and outputs it as a collection of files.

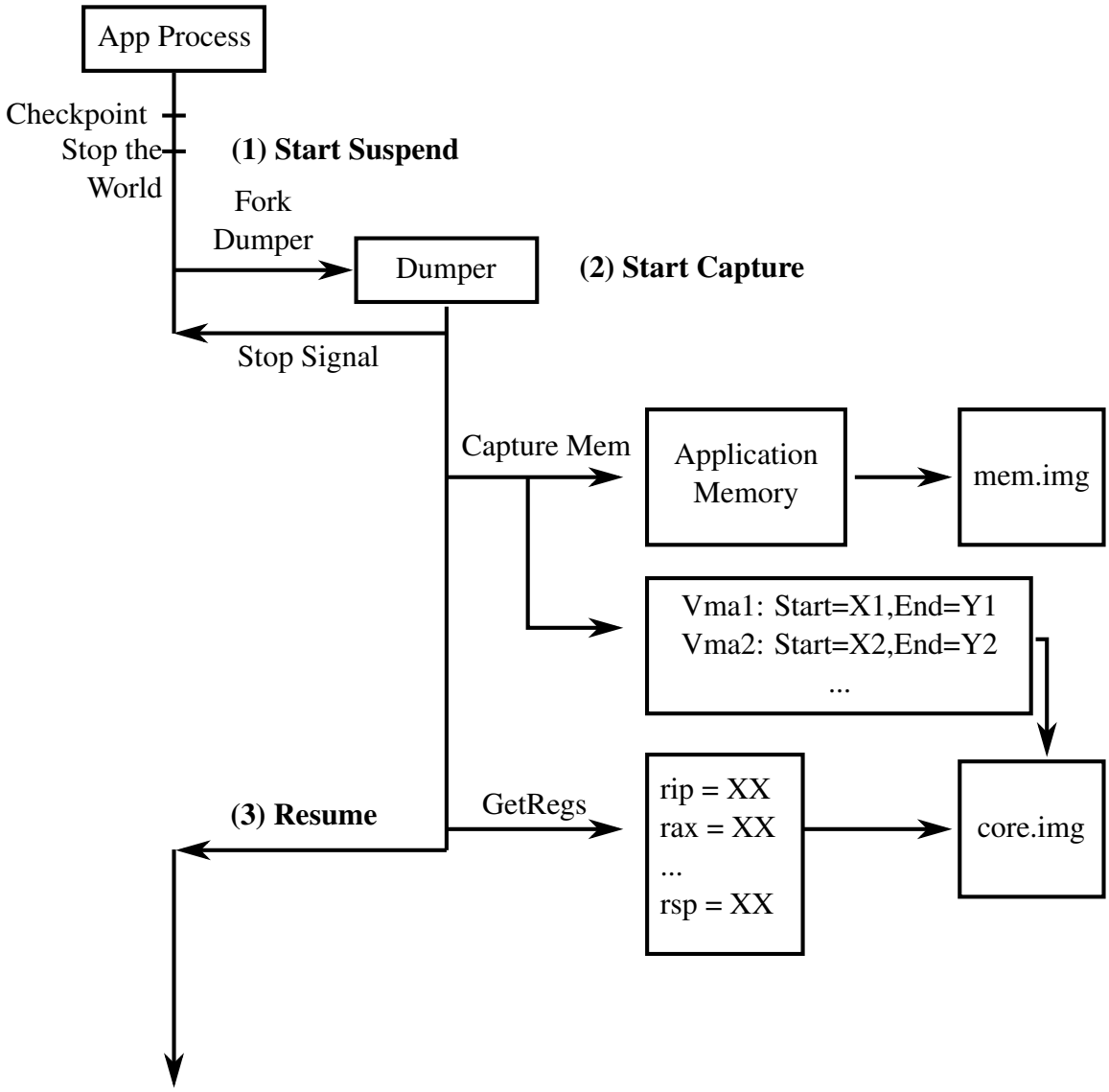

Figure 3-3: The checkpoint operation

3.3.2 Suspension

In order to safely take a checkpoint, CheckSync must ensure that none of the application's state will change in the midst of the checkpointing operation. Much like a garbage collector, this requires that CheckSync be able to stop the world, pausing the execution of all running threads.

To understand how the suspension step works, it is important to discuss goroutines and how the go runtime interacts with them. Goroutines are Go's implementation of user-level threads, and run on kernel threads. However, no goroutine is tied to a single thread. Instead, the Go runtime has an internal scheduler that decides which goroutine to run on which kernel thread.

In order to facilitate garbage collection, the Go scheduler can interrupt a goroutine to stop it's execution. The points at which this can occur are called preemption points, and CheckSync makes use of them to assist in the suspension of the application as well. There are two kinds of preemption points, synchronous ones inserted by the compiler, and asynchronous points computed by the runtime as the thread is executing.

Synchronous preemption points are inserted by the compiler and are locations in the goroutine's execution where it pauses to check if the scheduler has issued a request for preemmption. When discussing goroutines, the scheduler is *not* the OS scheduler, but rather the scheduler inside the runtime. The runtime scheduler will issue requests for preemption whenever the garbage collector needs to run, and CheckSync uses an identical mechanism to issue a request for preemption as well.

Synchronous preemption points don't cover all cases where a goroutine may need to be suspended, however, as there is no guarantee a goroutine ever reaches a preemption point. For this reason, the scheduler can issue a signal to a goroutine that forces it to suspend. Then, the scheduler checks if the goroutine is safe for the garbage collector to run. If it is, the thread stays suspended, and if not the scheduler lets it run for a bit before trying again.

An illustration of goroutines and preemption points is shown in fig. 3-4. Note that the red squares have been added for clarity's sake to represent preemption points.

3.3.3 Dump

Once all the other goroutines running in the application are suspended, the only one running is the one taking the checkpoint. Therefore, it is safe for CheckSync to capture the memory of the application, as it knows nothing will modify it any further. After dumping the memory, CheckSync dumps the contents of the registers at the time of the checkpoint.

Capturing the memory of the application is done using Linux's `proc` pseudo filesystem. Proc exposes all the information about a process' allocated virtual memory areas (VMAs), which CheckSync reads once the application has been suspended. This information is parsed from proc and stored as a table in memory. This table is then written out to the "core image file", which will eventually contain the register values as well.

CheckSync uses `ptrace` to capture the contents of the registers. As a process cannot trace itself, CheckSync forks a child process to facilitate both the dumping of memory as well as the registers. While CheckSync could invoke assembly code directly to do this, `ptrace` is more portable and adds less opportunity for bugs in CheckSync

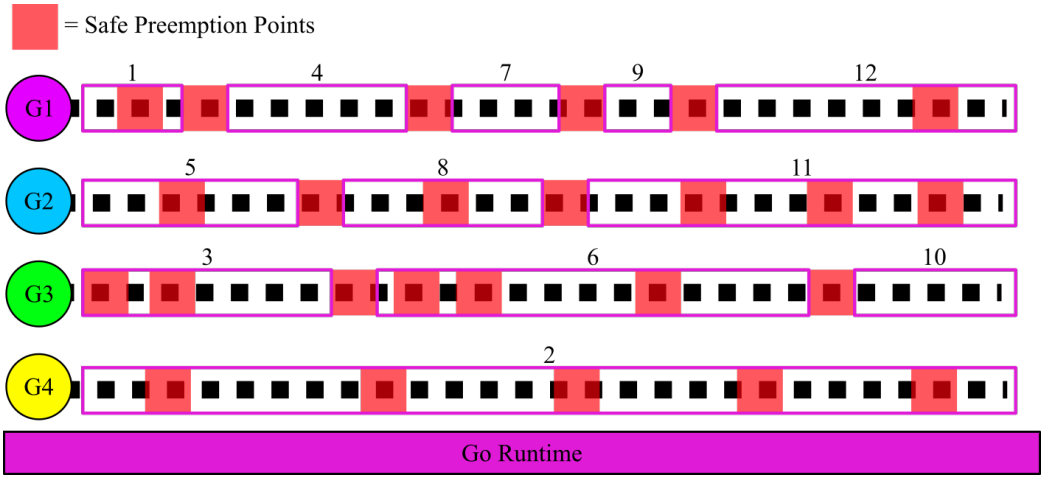

Figure 3-4: Goroutines and Preemption Points. The Go runtime can preempt any of the goroutines at a safe point, choosing another one to run.

The child process, termed the dumper, uses the `proc` filesystem to dump the contents of its parent's memory areas to disk as one single file. This is the "memory image file", which contains all virtual memory areas that are readable. VMAs that aren't readable, such as copy-on-write mappings and the pages used for virtual system calls aren't dumped, but their information is stored in the core image file to facilitate proper restoration. Finally, the dumper uses `ptrace` to capture and dump the values stored in its parent's registers to the core image file as well.

An illustration of the contents of both image files is shown in Figure 3-5. The memory image file (mem.img) contains the actual data from the application's memory (obtained during the dump step), and the core image file (core.img) contains all the data needed to reconstruct the memory space on the backup. Also included in the core file are the contents of the registers at the time the checkpoint was taken.

The parent of the dumper simply waits for the dumper to finish execution, at which point it starts the world again and allows the application to resume execution.

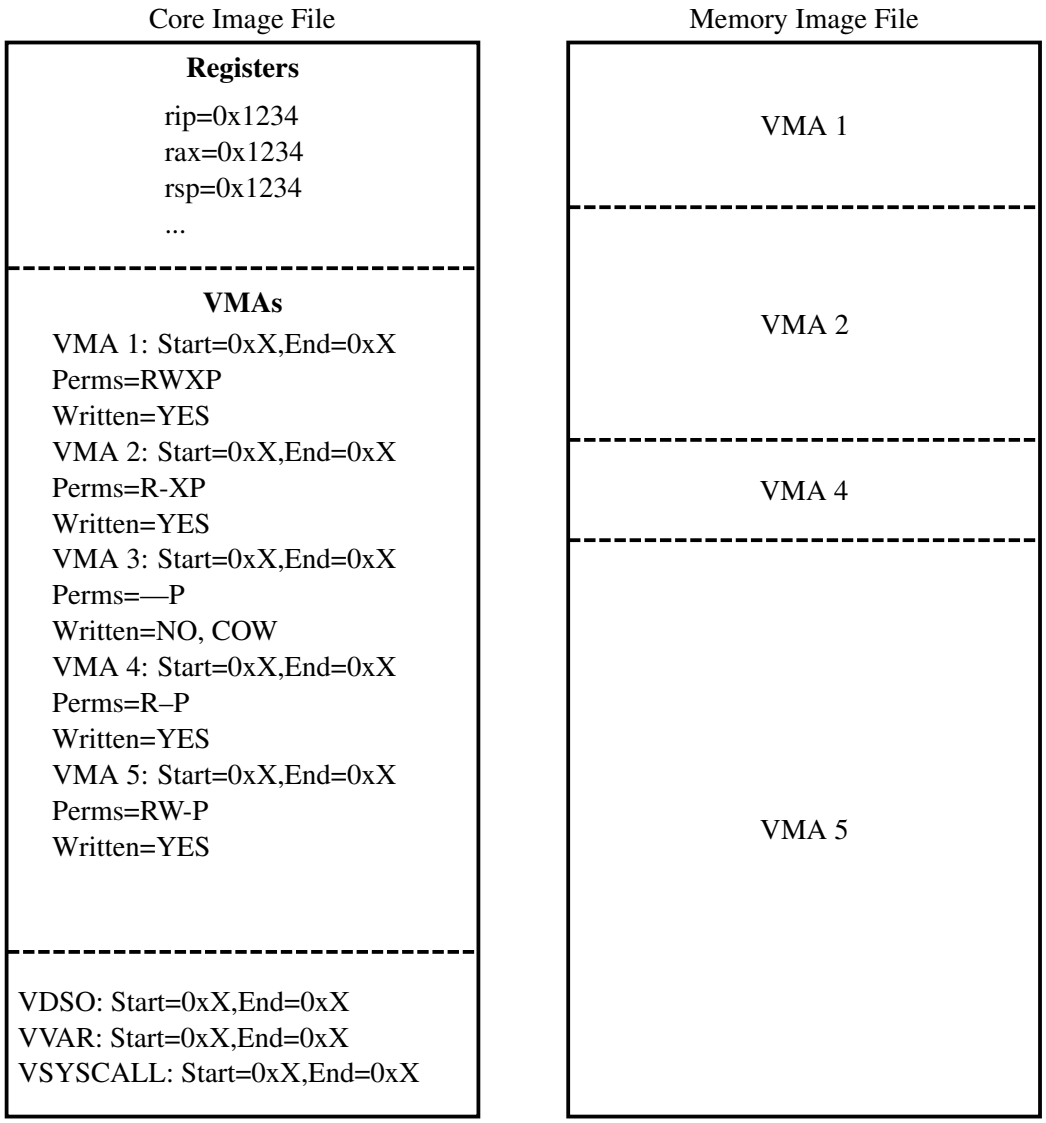

Figure 3-5: Checkpoint Image Structure. mem.img contains the raw bytes from each of the application's VMA's, and core.img contains all the extra information needed to map those areas to the correct location on the backup, as well as register contents.

3.3.4 Restore

There are a few things that must be restored for the application to work correctly. First, the memory of the application process must contain all the mappings that were present on the primary. Also, all the threads present on the primary must be recreated on the backup. Finally, the registers for all the threads have to be correctly restored as well.

Currently the internal checkpointer does not handle multithreaded code. However, extending it to do so may not be complicated as the Go runtime decouples goroutines from the underlying kernel threads that support them. The scheduler picks from available goroutines and maps them to available threads whenever it runs. CheckSync might be able to use this

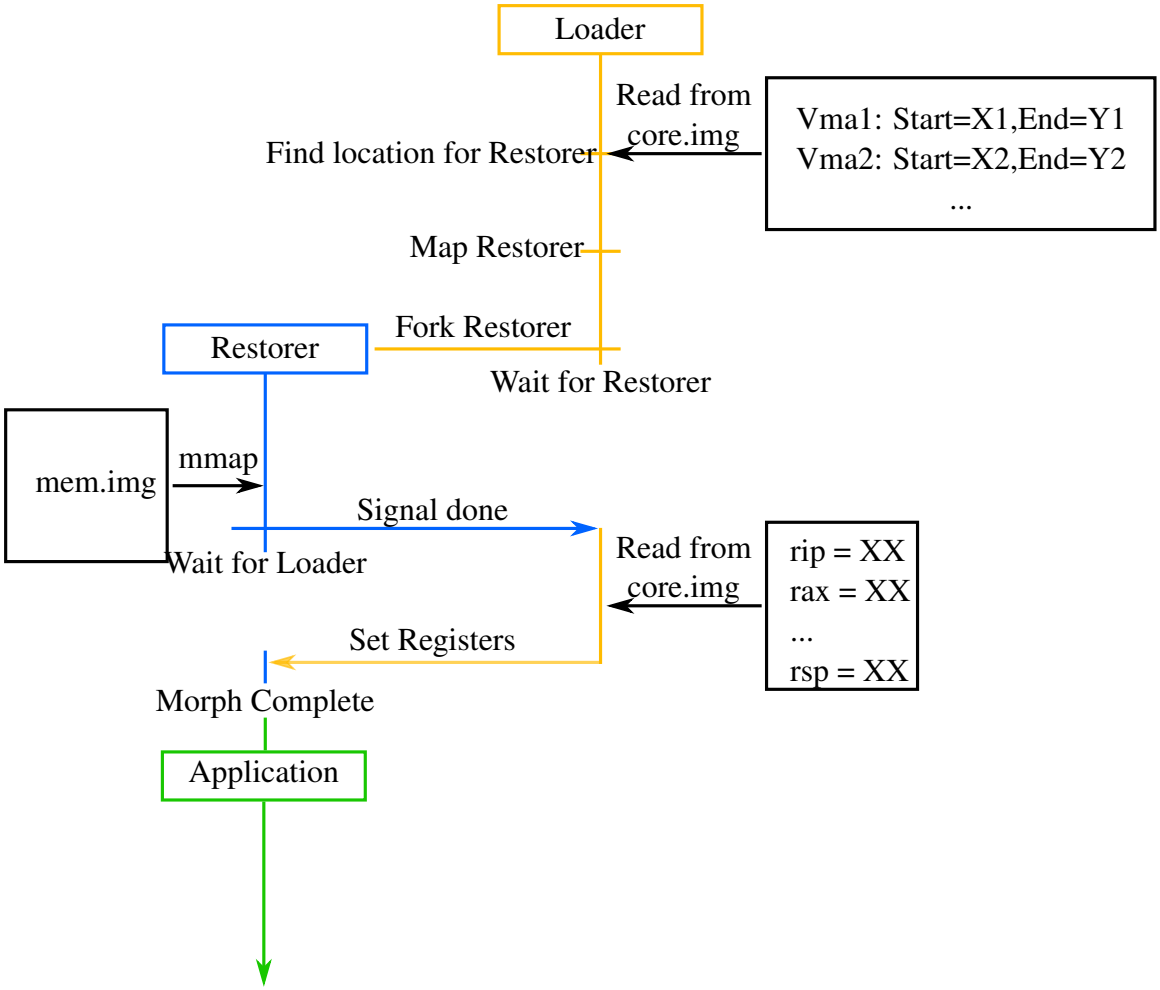

Figure 3-6: How CheckSync restores from a checkpoint.

ability to dynamically change the number of threads available to force the use of only one thread when a checkpoint is taken, and then recreate the original number of threads when the checkpoint finishes. This way CheckSync doesn't need to worry about threads.

CheckSync has two components that take care of restoring from a checkpoint: the loader and the restorer. Both are C programs that live on the backup and are invoked by the manager when a failure occurs. The manager starts by executing the loader, which then takes care of the rest of the process.

The first thing that needs to happen is for the loader to figure out what areas of memory are needed by the application being restored. It does this by reading in the core image file and looking at the table containing the information about all the VMAs. It also parses the proc filesystem to figure out its own VMAs as well.

It is precisely because these areas are likely to overlap that the restorer exists separately from the loader. The restorer is compiled with position independence enabled, allowing it to run no matter where in memory it gets located. Then, CRIU's compel utility [2] is used to turn the resulting binary file into a header file containing an array of the binary data for the restorer. This file also contains information about the locations of crucial functions

within the restorer. This array is referred to as the "restorer blob".

The loader takes the two lists of VMAs and the length of the restorer blob and computes a region of memory that intersects with neither its own mappings nor the application's and maps the restorer blob to this location. Additionally, it allocates space at the end of the restorer blob for a series of arguments that the restorer code will need. Most important in this is the information about the application's VMAs, as well as the loader's own VMAs.

Then, the loader forks off a child process and opens a pipe to communicate with it. This child process is responsible for jumping into the restorer. It does this by putting the address to the arguments on the stack and then jumping to the main function in the restorer. At this point, the restorer is running as the child process. The restorer is responsible for

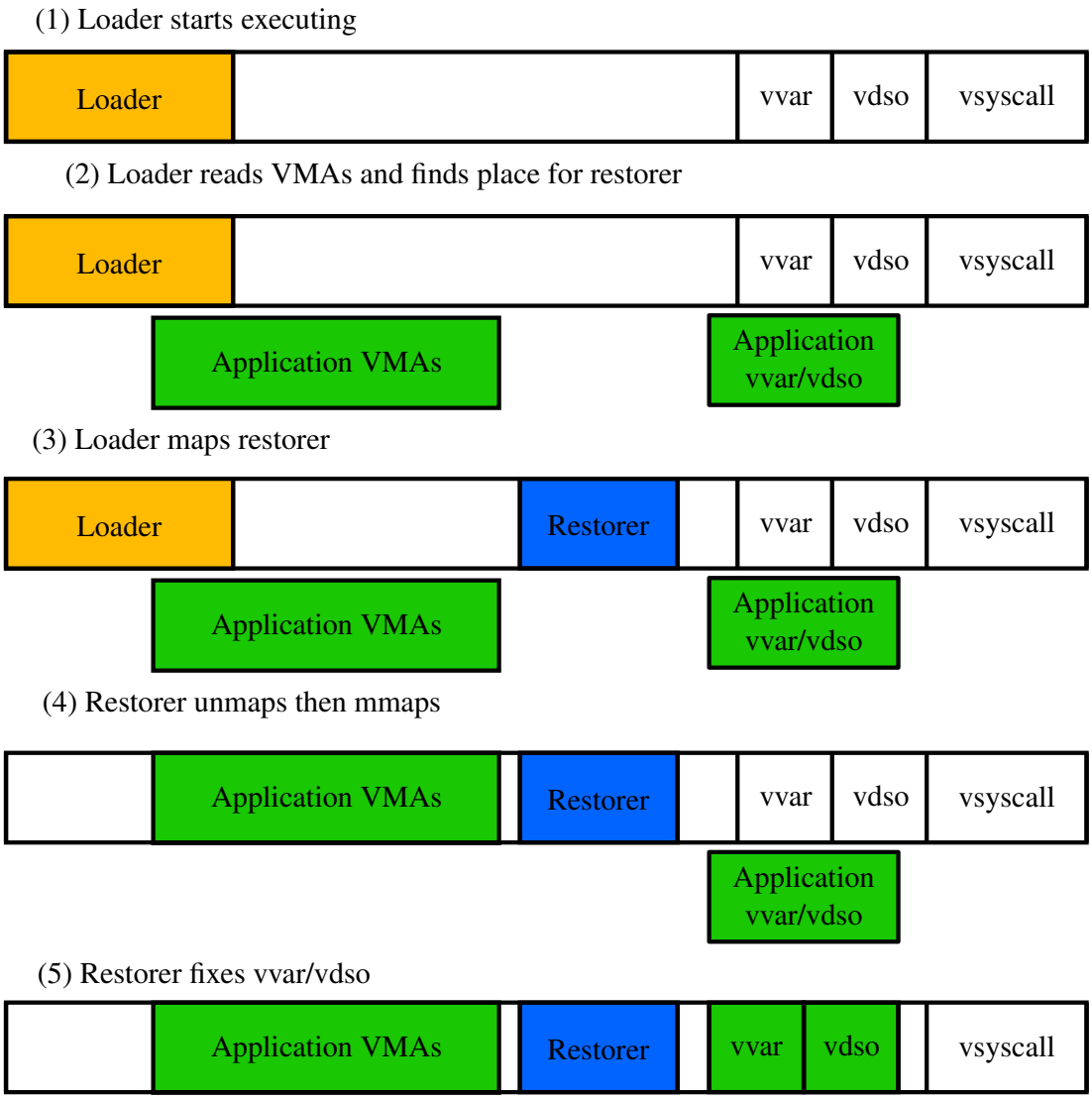

Figure 3-7: Memory layout on backup as restoration progresses

reconstructing the memory of the application. This is illustrated in fig. 3-7. It starts by unmapping all of the loader's mappings, as they are no longer necessary. Then, it looks at the list of VMAs needed by the application, opens the memory image file, and maps them

to the proper locations in memory. It also allocates space for any copy-on-write mappings not in the file, and finally concludes by remapping the virtual system call areas to their expected locations as well.

At this point the restorer uses the pipe to inform the parent process (still the loader) that it is done. The loader then finishes the restoration step by stopping the restorer and using `ptrace` to restore the state of the registers to what they were on the primary. Then, it detaches, at which point the restorer process has morphed entirely into the application process as it was captured on the primary.

Finally, the loader exits and the application resumes execution.

3.4 Consistency

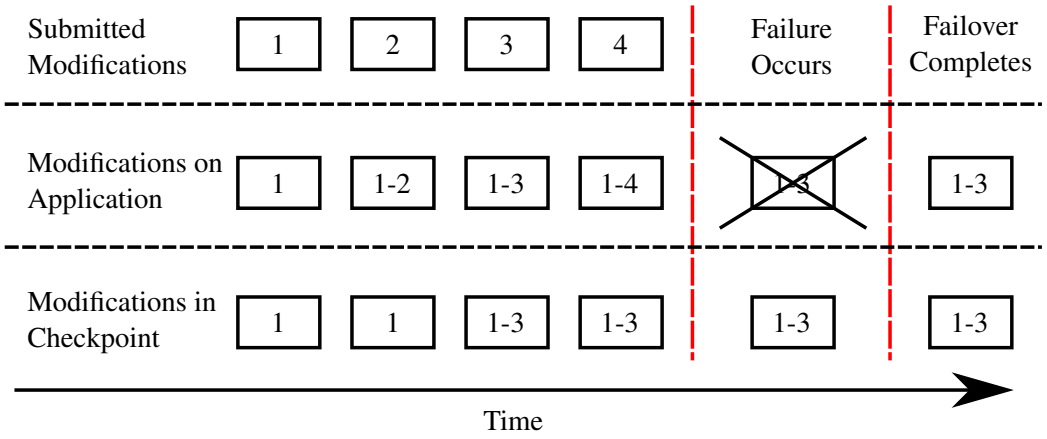

Figure 3-8: Timeline Consistency in CheckSync when failure occurs

CheckSync presents clients with a timeline-consistent view of the state of the application. We use the same definition of timeline consistency as is used in PNUTS[14]. This means that when a failover occurs, clients may be presented with an stale data, as the restored application will not reflect any changes that occurred between the last checkpoint and the failure. The stale data will reflect a valid prefix of the pre-crash operations. This is illustrated in Figure 3-8.

This is analogous to the consistency model presented by many existing systems designed to fulfill similar roles as CheckSync. While consensus protocols provide strong consistency rather than the weaker model used by CheckSync, systems like Redis[6], Memcached[24], and even VMWare's newer approach to fault tolerance [7] all provide weak consistency as well.

However, CheckSync does offer an alternative approach to checkpointing for applications that need strong consistency. Such applications do, however, need to know about and work with CheckSync. For these applications, CheckSync provides a blocking function call that allows them to start a synchronous checkpoint. If the application calls this whenever a state modification is made, then CheckSync will be strongly consistent.

To better facilitate this, CheckSync makes an optimization to the synchronous checkpointing process. Instead of capturing all of an application's memory, it instead relies on

the application passing it a pointer to the data structure that needs to be made strongly consistent. This shrinks the size of the checkpoint, reducing the amount of time the application is blocked for. However, as that data structure cannot be modified while the synchronous checkpoint is happening, modifications are made serial. Therefore, CheckSync loses the benefits of multithreading for writing threads. Reading threads can still execute in parallel.

The restoration process is also changed in this case. Instead of following the restore process, the application is just started directly. Then, a library function provided by CheckSync reads the checkpoint file and correctly restores the data structure.

This strategy of capturing and restoring only necessary structures could be made transparent. While not implemented currently, it would be possible for the internal checkpointer to sweep through an application's memory before checkpointing, and capture only the live data structures.

3.5 Application Support

Feature	**External Supported**	**Internal Supported**	**Internal Could Support**
Network I/O	Yes	No	Yes
File I/O	Yes	No	Yes
Fork/Exec	Yes	No	No
Devices	No	No	No
XServer Apps	No	No	No

Table 3.1: CheckSync's Checkpointing Schemes and the Features they Support

Table 3.1 lists the restrictions on the kind of applications that can be checkpointed using CheckSync. Generally, any operation supported by CRIU could be supported by the internal checkpointing method as well. The exception to this is fork/exec. As the internal checkpointer is inside the application process, the child created by fork/exec would not be in communication with the checkpointer. Therefore, the checkpointer would fail to checkpoint it.

Supporting File I/O for the internal checkpointer is possible. If the primary collected open file descriptors and the paths to the files they represent in the core image, and the file system on the backup was identical to the primary, then the loader could open all the files before restoration finished. An identical file system could be created using rsync[33], or by running on a networked storage system like Azure, Hadoop, or the like.

Supporting Network I/O requires cooperation from the Linux Kernel. The ability to mark a socket as `TCP_REPAIR`. The kernel then takes care of restoration of the socket on the backup itself. By capturing and restoring the queue and state of the socket, CheckSync could therefore ensure that the socket was reconstructed identically on the backup.

Supporting both devices and graphical applications that use XServer is currently not feasible. Hardware device drivers and the XServer both store application specific state

inside them, and CheckSync does not have access to this data. For this reason, it cannot replicate that state and therefore cannot support applications that rely on them.

Chapter 4

Implementation

Component	Lines of Code
Checkpoint Runtime Changes	< 100
Checkpoint Library Code	200
Checkpoint Loader	400
Checkpoint Restorer	300
Manager Code	500

Table 4.1: Lines of code for CheckSync's components

Table 4.1 lists the components of CheckSync and the lines of code used to implement each of them. The code CheckSync uses to take checkpoints lives partially in a modified version of the Go runtime, and partially in an external go library.

The code to perform the restore operation is divided into two components. First, the loader, which parses the various files that make up a single checkpoint, reads the results into memory, and then switches contexts to the restorer. Both it and the restorer are written in C.

The restorer is a piece of C code compiled with position independent flags, and then passed through the Compel utility. Compel takes care of formatting the resulting binary object file as an array of bytes and includes useful pointers to specific locations in the array. This is then included by the loader so that it can map the restorer into memory as described in the previous chapter.

The manager is written entirely in Go, using the grpc and the proto format for rpcs.

Chapter 5

Evaluation

We conducted three kinds of experiments with CheckSync to answer the following questions:

1. Transparency (§5.2): how much work does it take to make an application work with CheckSync?
2. Checkpoint Overhead for timeline (§5.3) and strong (§5.4) consistency: how much impact does the checkpointing phase have on application performance?
3. Failover (§5.5): how much downtime is experienced by an application when a failover occurs?

The results of these experiments show that CheckSync does deliver on its goal of providing applications with minor changes to the application. The only changes needed were small ones to handle reconnecting to the backup in case of failure. The performance of CheckSync is also satisfactory, incurring only a 30 percent loss in throughput when checkpoints are taken extremely frequently. This is an improvement over existing solutions which result in a more than 80 percent loss at the same frequency. Finally, we found that when failure does occur, CheckSync is able to recover in under a second, even when the checkpoint file is more than two gigabytes.

These measurements confirm that CheckSync accomplishes its goal. It provides transparent fault tolerance and high availability without a debilitating performance loss.

5.1 Experiment Setup

These evaluations were conducted in two conditions. First, locally on an 8-core desktop machine running Linux. We refer to this as the "local" test setup. Second, remotely using an Amazon Web Services' virtual private cloud. The primary and backup were run on identical, m5.xlarge EC2 instances. Linux was used there as well. We refer to this as the "remote" test setup.

All evaluations except those testing CheckSync's strong consistency offering were performed using the CRIU checkpoint/restore method. This is because the internal checkpointer currently doesn't support network connections, which are needed for the applications we looked at.

All experiments were run 50 times and the measured values varied little between runs. The only exceptions were the measurements on the remote setup, which were impacted by natural variation due to network latency and varying load on AWS. Even in that situation, values were quite consistent.

5.2 Applications Evaluated and Transparency

Table 5.1 shows that CheckSync can support a variety of applications with small changes to the actual application code. We selected three different applications to test this, each with unique characteristics, to evaluate the transparency provided by CheckSync. First, the MapReduce Coordinator [17]. Second, a key/value store, and third an application for computing the powers of large matrices. Table 5.1 shows the lines of code changed in these applications so that they can use CheckSync.

Application	Lines of Code	CheckSync Changes	Raft Changes
MapReduce	200	0 + 5	200+
K-V Server	100	0	50
Matrix Multiplication	130	0	N/A

Table 5.1: Lines of code changed for different applications. CheckSync required little-to-no changes, while making the conversion to Raft for the same applications required significant restructuring of the code.

5.2.1 Coordinator

The coordinator benefits from having CheckSync as it is a single point of failure in the MapReduce system. Providing fault tolerance for the coordinator therefore increases the overall reliability of the system. It is also an interesting case study for CheckSync as existing coordinators are written to be multithreaded. This makes it difficult to convert them to work with replicated state machines. Demonstrating that CheckSync *can* support the coordinator therefore sets it apart from those systems.

We used a MapReduce coordinator written for the first lab of 6.824[1]. The coordinator was run on the local setup and was configured to run a simple word counting job. After confirming that the output of the word count was correct, it was run on top of CheckSync. Even with a failure in the middle of the job the end result was still correct. A few lines of code were changed to make sure that the workers tried reconnecting to the coordinator when an RPC failed to go through. These changes are the +5 in table 5.1.

5.2.2 Key/Value Store

The second application evaluated was an in-memory key/value store in the style of Redis [6] and Memcached [24]. Such key/value stores are often used as caches in a larger system,

and benefit from having some sort of fault tolerance so that they never get too cold, even if a failure occurs.

The server maintains an internal hash table of keys and values, and exposes a get and put RPC for clients to use. Both gets and puts are executed in parallel, and bucket locks are used to ensure that concurrent client requests are handled safely.

No changes were required for it to work with CheckSync; this was due to CRIU. We expected to have to change client code similar to what was done for MapReduce; we thought they would have to be configured to attempt to reconnection on failure. However, CRIU's checkpointing implementation reconstructs the TCP connection on the backup, which meant that no client code had to be changed. We expect that for the internal checkpointer, a few lines would need to be modified so that clients would behave correctly.

5.2.3 Matrix Exponentiation

Finally, we wrote an application that performs repeated matrix exponentiation of a large matrix. This application also required no code changes to work with CheckSync. This application runs a long, continuous computation with no communication of any sort, similar to what a scientific computing application would do (§2.4).The characteristic of long, continuous computation sets it apart from the other two applications, which perform work only when requested.

The application benefits from using CheckSync because in the case of a failure, instead of starting from scratch, CheckSync enables the application to resume from the last checkpoint. This can save significant time and compute cycles. Also, this benefit of saving computation is not what replicated state machines provide. Replicated state machines provide high availability by replicating the work multiple times, and need a log of operations to do so. Applications like this one don't fit that model. That CheckSync supports this application with no changes demonstrates its versatility.

5.2.4 Using Raft

Verifying that CheckSync was able to support all three applications with only small changes shows the value in using CheckSync. We also tested the degree of the changes needed to make the coordinator and key/value store work with Raft. To do so, we used etcd's implementation of Raft[4], which has a good API for writing applications. Nevertheless, we found that both applications required non-trivial changes to work with it.

As noted earlier, the matrix exponentiation application doesn't get any benefit from using Raft. For this reason we felt it unfair to try and force it onto the consensus protocol.

For all applications, table 5.1 shows only the changes made to the server, and not the additional code needed to start and run the Raft instances that back the store.

The Key/Value store required less restructuring than the coordinator. All traces of multithreading had to be removed, and the code for the put operation was modified to add entries to the log rather than directly updating the hash table. A function was also added to read committed entries from Raft and apply those commits to the hash table. The get operation was unchanged.

```
func (c *Coordinator) handleCompleted() {
    for {
        job, ok := <-c.successJobs
        if !ok {
            break
        }
        switch job.JobType {
        case MapJob:
            id := job.JobId
            if _, exist := c.successJobSet[id]; !exist {
                c.successJobSet[id] = true
                if len(c.successJobSet) == len(c.inputFiles) {
                    c.mx.Lock()
                    if c.reduceCreated {
                        break
                    }
                    for j := 0; j < c.nReduce; j++ {
                        filenames := createReduceFilenames(c.rawFiles, j)
                        c.availableJobs <- ReduceJob(filenames, c.nReduce)
                    }
                    c.reduceCreated = true
                    c.mx.Unlock()
                }
            }
        case ReduceJob:
            for _, fileName := range job.FileNames {
                jobId := reduceJobId(job)
                c.successJobSet[jobId] = true
            }
            if len(c.successJobSet) == len(c.rawFiles)*(c.nReduce+1) {
                c.mx.Lock()
                close(c.availableJobs)
                close(c.successJobs)
                c.isSuccess = true
                c.mx.Unlock()
            }
        }
    }
}
```

Figure 5-1: The main coordinator function in the original MapReduce implementation.

```
func (c *Coordinator) handleCommits() {
    for commit := range c.commitC {
        if commit == nil {
            // signaled to load snapshot
            snapshot, err := c.loadSnapshot()
            if err != nil {
                log.Panic(err)
            }
            if snapshot != nil {
                if err := s.recoverFromSnapshot(snapshot.Data); err != nil {
                    log.Panic(err)
                }
            }
            continue
        }

        newlyCompleted := make([]Job, 0)
        newlyCreated := make([]Job, 0)
        for _, data := range commit.data {
            var job Job
            if isNewJob(data) {
                // This function puts the job on the queue of waiting jobs
                handleCreated(job)
            } else {
                // This performs the same functionality
                // the original handleCompleted
                // But instead of immediately adding new jobs
                // it proposes them
                handleCompleted(job)
            }
        }
    }
}
```

Figure 5-2: The modified coordinator function for Raft. Instead of relying on workers to tell us a job is complete, the coordinator instead waits on commits from Raft before marking them as complete. It also watches for new jobs having been created, putting them on the queue of available jobs once they are.

In the initial implementation of the coordinator, workers request jobs and then inform it when the jobs are completed. The coordinator answers request for new jobs in parallel, waiting in a main loop and watching for when any of the assigned jobs are completed.

Making the coordinator work with Raft required a restructuring of the way that jobs are stored and managed by the coordinator. Both the list of completed and outstanding jobs need to be made fault tolerant for the coordinator to become fault tolerant. This meant that any change to the jobs needed to be proposed to Raft, and no action could be taken on those changes until Raft committed them. This led to the modified function shown in fig. 5-2, as compared to the unmodified version shown in fig. 5-1.

This required modifying the other parts of the coordinator as well. As noted, handleCompleted was changed to propose new jobs rather than immediately create them. Also, all RPC handlers were made serial through the use of locks, and workers were forced to block until their job was committed by Raft. The job system was also changed to rely on a queue rather than a channel.

All of these changes combined required more than 200 lines of code to be changed, as noted in table 5.1.

5.3 Checkpointing

This section presents the experiments used to evaluate the overhead introduced by the checkpoint operation. The experiments also evaluated the size of the checkpoints on disk. They show that this overhead is within the bounds of the overhead introduced by similar operations on existing systems. However, as the size of the checkpoint grows, the operation does become expensive.

Keys	Checkpoint Size on Disk (MB)	Approximate Total Size of Keys
1000	8.16	0.07
10000	8.16	0.72
100000	27.08	7.20
1000000	289.19	72.00
10000000	2287.61	720.00

Table 5.2: The size on disk for different numbers of keys. As the key space grows, the disk size does as well.

To evaluate the overhead introduced by CheckSync, we used different checkpoint sizes. To accomplish this, we took the key/value store and populated it with various numbers of keys before running any experiments on it. The number of keys we picked and the resulting checkpoint sizes on disk are tabulated in table 5.2. Each key/value pair totalled 72 bytes. The extra size of the checkpoints comes from extra space on the heap and stack, the space taken up by the runtime, and statically linked libraries like libc.

We used a simple client that produced a constant stream of serial requests on a 90%-get-10%-put workload. The client submits a single request, waits for it to complete, and then submits the next request. This experiment allows us to evaluate the overhead introduced by checkpointing: whenever a checkpoint is taken the server will not process any requests.

5.3.1 Checkpoint Performance

Table 5.3 shows the time it takes for CheckSync to produce a checkpoint. This is the amount of time the application is suspended for, which therefore has the most impact on application performance.

The suspending and dumping columns correspond to the operations of the same time described in chapter 3. The cleanup step occurs after dumping is complete, and measure the work done to get the application to resume after everything has been written to disk. As expected, as checkpoint size grows, the time spent in the dumping stage grows to dominate the checkpoint time. In the future, this can be mitigated by taking incremental dumps rather than a new dump each time and taking advantage of the internal checkpointer. This is discussed in detail in chapter 6.

Not included in the table is the time spent sending the checkpoint to the backup. This happens in parallel with the application, and therefore has minimal impact on the application's performance. This time is entirely dependent on the size of the checkpoints, and the quality of the network. Taking incremental dumps will also help reduce this time.

Keys	Total	Suspending	Dumping	Cleanup
1000	15.11	2.12	12.57	0.27
10000	16.50	2.15	13.93	0.42
100000	24.80	2.14	22.24	0.31
1000000	441.87	84.25	356.90	0.73
10000000	2208.28	150.78	2051.07	6.42

Table 5.3: Time spent in different phases of the checkpoint process (in milliseconds). As checkpoint size grows, the time spent dumping dominates the time spent checkpointing as the disk becomes a bottleneck.

Figure 5-3 shows the number of requests completed over time for the key/value store with 1000 keys and checkpointing every 100ms. As can be seen in the graph, there are periods of time where no requests are processed. This represents time spent taking a checkpoint, as expected given the measurements in table 5.3.

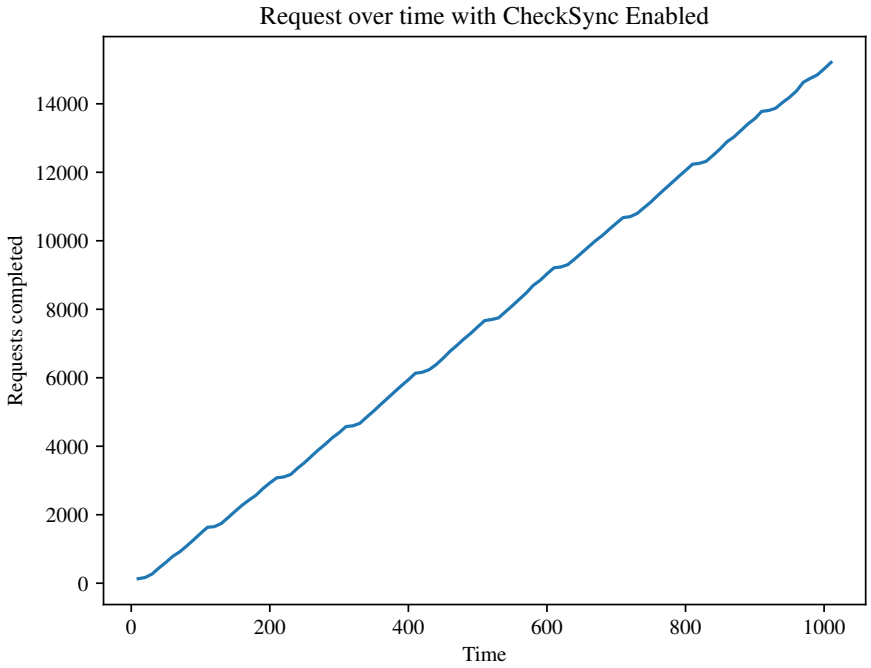

Figure 5-3: Requests over Time with 1000 keys and checkpointing every 100ms. The duration of the pauses in completion line up with the values in table 5.3

To determine whether the overhead introduced by CheckSync is reasonable, we looked at Redis, an industry-standard key/value store and measured how its performance is impacted by its save operation, which also suspends the server and writes to disk.

We constructed a client nearly identical to that used to measure CheckSync's impact, with the only difference being that it contacted a Redis server rather than a CheckSync one.

We compared the overhead introduced in each case. Redis is faster than our key/value store when save/checkpoint aren't used. This remains the case when save/checkpoint are turned on, however the relative amount of requests per second lost is much higher for Redis, especially at higher intervals of saving. The percentage of requests lost is tabulated in Table 5.4, which shows that when Redis saves once per second, it loses 13% of the performance without saving, while CheckSync loses only 2% in the same situation.

We also compared the key/value server with the Raft port of the same application. We used a three-node Raft cluster running locally, which achieves 2000 requests per second. This is slower than the throughput for CheckSync even when checkpoints are taken every 40 milliseconds for a single client. This is likely due to the extra replication factor Raft uses, as well as the fact that every modification incurs a penalty due to that replication. This differs from CheckSync as CheckSync's saving is periodic rather than occurring on every modification.

Interval (ms)	CheckSync Req/s	Percent Lost by CheckSync	Redis Req/s	Percent Lost by Redis
50	13378.36	29	10817.00	82
500	18508.00	2	46693.80	24
1000	18667.16	2	53495.80	13
1500	18788.92	1	54535.00	11
2000	18496.76	3	56111.20	8
None	18974.00	N/A	61174.00	N/A

Table 5.4: Comparing Redis' save overhead with CheckSync checkpointing. The percentage loss of requests per second in Redis is higher than CheckSync, especially at shorter intervals.

5.4 Strong Consistency

One advantage of Raft is that it provides strong consistency to applications. CheckSync's internal checkpointer also has this ability. Applications pass it the data structures that need to be strongly consistent and then CheckSync captures only the memory used for that data structure and replicates it.

We evaluated the cost of strong consistency with the key/value store. The server passes the underlying map structure that supports the key/value store to CheckSync anytime a client submits a Put operation. This required 10 lines of code to be changed in the server, still less than what was needed for Raft. Table 5.5 shows the time spent encoding the hash table as well as the resulting checkpoint size for these experiments.

Of particular note is the size of the checkpoints in this configuration. On a key/value store with one million key, the size of the synchronous solution was 3.9 times smaller than the periodic solution. This is because the periodic solution replicates the whole process rather than just the hash table. This is good evidence as to the value of the internal checkpointer: as noted earlier, if the internal checkpointer were improved to sweep the process' memory and checkpoint only live data structures, it would clearly decrease the size of the checkpoint.

Also, the encoding operation is almost 3 times faster than the dump operation in the periodic solution. Given the size difference between the two checkpoints this makes sense. Although, it does point to some room for optimization in the encoding step, as the dump for the external checkpointer is comparatively faster than this option given their relative checkpoint sizes.

We also measured the throughput of the key/value store with CheckSync's strong consistency option. We used the same 90% get workload to do so, again running on the local setup. Compared to a key/value store using etcd's Raft implementation and running on the same number of keys, CheckSync came out ahead, clocking 2500 requests per second as compared to 2000 from the Raft-backed store.

Keys	Checkpoint Size on Disk (Bytes)	Time Spent Encoding (ms)
1000	0.15	3.33
10000	0.85	5.91
100000	7.42	33.57
1000000	74.00	309.56
10000000	740.00	3463.14

Table 5.5: Checkpoint Size and Time Spent Encoding for Strong Consistency. Both encoding time and size are around a quarter of their values in the periodic checkpointing. This points to the efficacy of checkpointing on the level of data structures rather than all of memory.

Keys	Total	Restoring VMAs	Restoring Registers
1000	3.15	2.18	0.97
10000	5.87	4.95	0.92
100000	11.03	9.67	1.36
1000000	71.43	70.38	1.05
10000000	500.08	498.99	1.09

Table 5.6: Time Spent in Different Phases of the Restore Process (in Milliseconds)

5.5 Failover and Restoration

When the primary fails, there is a period of downtime present before the backup takes over. This length of this period has an impact on application availability, so keeping it low is important. In this section, we evaluate the downtime experienced by the key/value store when using CheckSync, showing that even with 10 million keys, the time to recover is only just over a second.

To do so, we ran CheckSync in both the local and remote setups. We used the local setup exactly as described in §5.3, except that part of the way through the client's run we induced a failure on the primary and measured the time it took for the client to return to normal throughput. After the backup detects 5 missing heartbeats, it triggers the failover process. The results of this are shown in fig. 5-4.

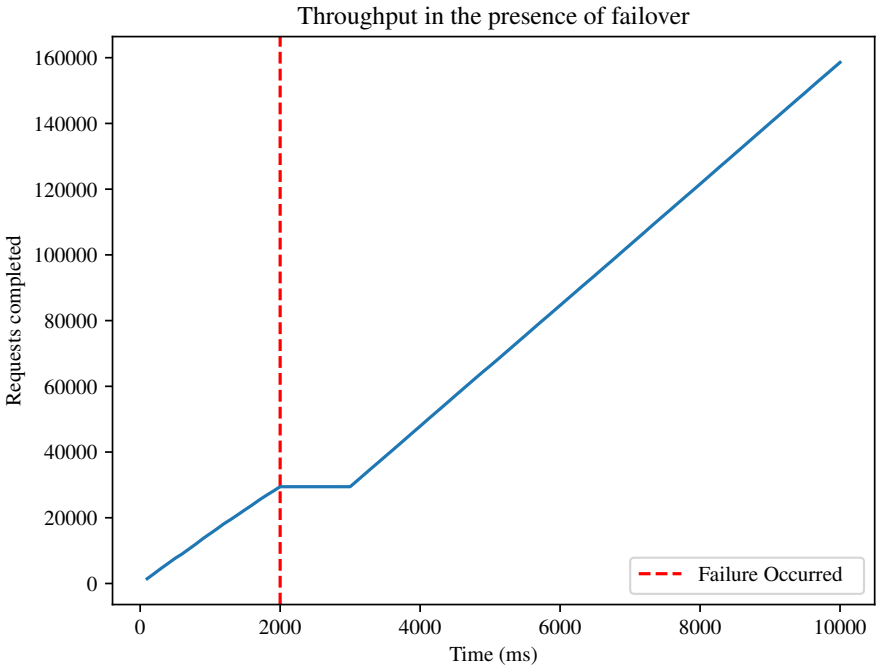

Figure 5-4: Impact of failover on application throughput. The pause in request completion matches with the timed results from table 5.6, demonstrating that downtime is reasonably short at under 1 second.

The reason for doing this on the local setup was to isolate the time spent in different parts of the restoration process for different checkpoint sizes. The breakdown is shown in table 5.6. As expected, restoring the VMAs eventually grows to dominate the time spent in restoration, as reading all that memory from disk becomes the bottleneck for the restore. Even with 10 million keys, though, restoration time stayed under one second.

The remote setup follows the description from chapter 3. We deployed the primary and backup in the same VPC on Amazon AWS, and assigned an elastic IP address to the primary. Then, we setup a lambda function to swap the elastic IP to point to it instead.

First, it carries out the restoration process exactly as it does in the local setup. Then, just before switching to the application completely, it triggers the lambda function to swap IPs so that the clients don't need to ask a clerk for the new location. Once the switch finishes, the backup resumes the application and completes the failover.

We deployed this with the key/value server on the remote setup, and evaluated the time spent in different phases of the process. The results are shown in table 5.7.

It should be noted that while these results are good, Raft's recovery time is faster than CheckSync. This is because Raft only needs to run a single election to recover from a failure, which doesn't take much time.

Keys	Total	Detecting Failure	Restoring VMAs	Restoring Registers	Swapping Elastic IP
1000	652.91	618.22	3.01	0.87	30.81
10000	595.02	558.01	6.02	1.11	29.88
100000	648.29	602.92	11.07	3.10	31.20
1000000	717.61	598.78	87.76	3.58	27.49
10000000	1132.09	570.18	528.63	3.07	30.21

Table 5.7: Time Spent in Different Phases of the Restore Process on Remote Deployment (in Milliseconds). Detecting failure time dominates the time spent here due to latency between primary and backup. However, even with a large checkpoint total time is just over 1 second.

5.6 Summary of Results

The experiments we conducted to evaluate CheckSync demonstrate four key results:

1.) CheckSync provides transparent fault tolerance. All three of the applications we evaluated CheckSync with required less than 10 lines of code to be changed in order to function with CheckSync. CheckSync successfully replicated and recovered from failure for all three applications.

2.) The overhead for taking a checkpoint is not unreasonable. Compared to Redis's save operation, CheckSync's checkpointing has a smaller impact on performance. This impact grows more significant the larger memory grows.

3.) CheckSync can achieve both strong consistency and smaller checkpoint costs with the internal checkpointer than the external one. This demonstrates the viability of the internal checkpointer and points to potential future optimizations even for the periodic checkpointing.

4.) CheckSync is highly available. The time spent recovering from failure is just over one second even with ten million keys in storage.

Chapter 6

Extensions and Future Work

6.1 Decreasing Checkpoint Size

One of the main advantages of integrating CheckSync's checkpointing code into the Go runtime is the access to the information the runtime has about the running application. For instance, the runtime knows about every structure in use by the application, which would potentially enable CheckSync to capture and replicate only those structures from the primary to the backup. Other potential uses of the runtime that we hope to investigate in the future include making use of the garbage collector to replicate only live data.

CheckSync's goal in using this extra information would generally be to reduce the size of the state being checkpointed. This could significantly increase the uptime of the application by reducing the amount of time that it needs to be frozen while CheckSync captures and dumps its state.

6.2 Incremental Checkpointing

Outside of reducing the size of checkpoints on disk, another optimization to the checkpointing step would be to use incremental checkpoints. It is unlikely that *all* of memory is changed between checkpoints, and CheckSync could use this to its advantage. One option would be to track memory changes (likely using the runtime) and only dump pages that were changed. Another angle might be to keep the previous checkpoint in memory and take a diff between it and the current one. By dumping only the diff, the amount of state being replicated for each checkpoint would decrease.

6.3 Fast Restoration

Another potential use of the runtime integration would be to speed up the restoration process. By making use of the fact that the runtime is identical on primary and backup, it may be possible to further modify the bootstrap process used by Go in ways that reduce the overhead of loading in the full checkpoint. For instance, the CheckSync runtime might be able to setup a skeleton of the application without even looking at any files on disk, only

making use of the checkpoint files to fill in specific gaps in memory. This could improve the delay experienced when a failure occurs.

6.4 Checkpoint Storage in Deployment

One other area for future investigation is improving the storage of checkpoints. As we envision CheckSync being deployed in cloud computing environments, it is natural that the files making up the checkpoint be stored in such a system as well. Doing so would eliminate the need for actually replicating the checkpoint from the primary to the backup, and would introduce a higher level of fault tolerance due to the guarantees provided by storage systems like Amazon S3. This could improve both the performance and availability of CheckSync in future deployments.

Chapter 7

Conclusion

CheckSync is a system designed to provide transparent fault tolerance for application developers. It can take a Go application and periodically checkpoint its state, replicate that state, and resume from it in the case of failure, without any cooperation from the application itself. CheckSync has two different checkpointing libraries, one using Go that demonstrates the feasibility of runtime cooperation in taking checkpoints, and another that uses CRIU. System configuration and maintenance is handled by managers running on the primary and backup. CheckSync was evaluated in both a local and remote deployment, demonstrating its viability as a method for achieving fault tolerance. In the future, we hope to make better use of CheckSync's integration with the runtime to reduce checkpoint overhead.

Bibliography

[1] 6.824: Distributed systems. https://pdos.csail.mit.edu/6.824/.

[2] Compel. https://criu.org/Compel.

[3] CRIU - Checkpoint/Restore in User Space. https://www.criu.org.

[4] etcd. https://etcd.io/.

[5] Golang. https://golang.org/.

[6] Redis. https://redis.io/.

[7] vsphere vmotion. https://www.vmware.com/products/vsphere/vmotion.html.

[8] Gautam Altekar and Ion Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 193–206, New York, NY, USA, 2009. Association for Computing Machinery.

[9] Joel Bartlett, Jim Gray, and Bob Horst. Fault Tolerance in Tandem Computer Systems. In Algirdas Avižienis, Hermann Kopetz, and Jean-Claude Laprie, editors, *The Evolution of Fault-Tolerant Computing*, pages 55–76, Vienna, 1987. Springer Vienna.

[10] Claudio Basile, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Active Replication of Multithreaded Applications. *IEEE Trans. Parallel Distrib. Syst.*, 17(5):448–465, may 2006.

[11] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. Application-Level Checkpointing for Shared Memory Programs. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 235–247, New York, NY, USA, 2004. Association for Computing Machinery.

[12] Tuan Cao, Marcos Vaz Salles, Benjamin Sowell, Yao Yue, Alan Demers, Johannes Gehrke, and Walker White. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 265–276, 2011.

[13] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, page 273–286, USA, 2005. USENIX Association.

[14] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[15] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable Deterministic Multithreading through Schedule Memoization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 207–221, USA, 2010. USENIX Association.

[16] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174, USA, 2008. USENIX Association.

[17] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[18] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: Replication at the Speed of Multi-Core. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[19] Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, Taesoo Kim, and Pavel Emelyanov. Instant OS Updates via Userspace Checkpoint-and-Restart. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 605–619, Denver, CO, jun 2016. USENIX Association.

[20] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.

[21] P A Lee, T Anderson, J C Laprie, A Avizienis, and H Kopetz. *Fault Tolerance: Principles and Practice*. Springer-Verlag, Berlin, Heidelberg, 2nd edition, 1990.

[22] K Li, J F Naughton, and J S Plank. Real-Time, Concurrent Checkpoint for Parallel Programs. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, PPOPP '90, pages 79–88, New York, NY, USA, 1990. Association for Computing Machinery.

[23] Jun-Lin Lin and Margaret H Dunham. A Survey of Distributed Database Checkpointing. *Distributed and Parallel Databases*, 5(3):289–319, 1997.

[24] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, page 385–398, USA, 2013. USENIX Association.

[25] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. *SIGPLAN Not.*, 44(3):97–108, mar 2009.

[26] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, USA, 2014. USENIX Association.

[27] Kun Ren, Thaddeus Diamond, Daniel J. Abadi, and Alexander Thomson. Low-overhead asynchronous checkpointing in main-memory database systems. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1539–1551, New York, NY, USA, 2016. Association for Computing Machinery.

[28] Santiago Rodríquez, Antonio Pérez, and Rafael Méndez. A new checkpoint mechanism for real time operating systems. *SIGOPS Oper. Syst. Rev.*, 31(4):55–62, October 1997.

[29] Daniel J Scales, Mike Nelson, and Ganesh Venkitachalam. The Design of a Practical System for Fault-Tolerant Virtual Machines. *SIGOPS Oper. Syst. Rev.*, 44(4):30–39, dec 2010.

[30] Fred B Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, dec 1990.

[31] Johny Srouji, Paul Schuster, Maury Bach, and Yulik Kuzmin. A transparent checkpoint facility on NT. In *2nd USENIX Windows NT Symposium (2nd USENIX Windows NT Symposium)*, Seattle, WA, August 1998. USENIX Association.

[32] X. Tang, J. Zhai, B. Yu, W. Chen, W. Zheng, and K. Li. An efficient in-memory checkpoint method and its practice on fault-tolerant hpl. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):758–771, 2018.

[33] Andrew Tridgell, Paul Mackerras, et al. The rsync algorithm. 1996.

[34] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, page 7, USA, 2004. USENIX Association.

[35] Ruini Xue, Wenguang Chen, and Weimin Zheng. Cprfs: A user-level file system to support consistent file states for checkpoint and restart. pages 114–123, 01 2008.

[36] Gengbin Zheng, Chao Huang, and Laxmikant V Kalé. Performance Evaluation of Automatic Checkpoint-Based Fault Tolerance for AMPI and Charm++. *SIGOPS Oper. Syst. Rev.*, 40(2):90–99, apr 2006.

[37] Gengbin Zheng, Lixia Shi, and L V Kale. FTC-Charm++: An in-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, CLUSTER '04, pages 93–103, USA, 2004. IEEE Computer Society.

[38] Li Zhuang Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou. Paxos Made Parallel.