A Distributed Snapshot Protocol for Virtual Machines

by

Gang Peng

B.Eng., Zhejiang University, 2002
M.Eng., Zhejiang University, 2005

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies
(Computer Science)

The University Of British Columbia
September 7, 2007
© Gang Peng 2007
Abstract

The distributed snapshot protocol is a critical technology in the areas of disaster recovery and computer security of distributed systems, and there have appeared a huge number of projects working on this topic since the 1970's. Recently, with the popularity of parallel computing and disaster recovery, this topic has received more and more attention from both academic and industrial researchers. However, all the existing protocols have several common disadvantages. First, existing protocols all require several modifications to the target processes or their OS, which is usually error prone and sometimes impractical. Second, all the existing protocols are only aiming at taking snapshots of processes, not whole entire OS images, which constrains the areas to which they can be applied.

This thesis introduces the design and implementation of our hypervisor level, coordinated non-blocking distributed snapshot protocol. Superior to all the existing protocols, it provides a simpler and totally transparent snapshot platform to both the target processes and their OS images. Based on several observations of the target environment, we simplify our protocol by intentionally ignoring the channel states, and to hide our protocol from the target processes and their OS, we, on one hand, exploit VM technology to silently insert our protocol under the target OS, and on the other hand, design and implement two kernel modules and a management daemon system in the control domain. We test our protocol with several popular benchmarks and all the experimental results prove the correctness and the efficiency of our protocol.
Contents

Abstract .................................................. ii

Contents .................................................. iii

List of Tables .......................................... v

List of Figures .......................................... vi

Acknowledgements ....................................... vii

1 Introduction ........................................... 1

2 Related Work ......................................... 6
   2.1 Properties of distributed snapshot protocols .......... 6
   2.2 Classification ....................................... 8
      2.2.1 Coordination Strategy ............................. 8
      2.2.2 Snapshot Granularity ............................. 14
   2.3 Summary ............................................... 16

3 Design and Implementation ............................. 17
   3.1 Requirements ......................................... 17
   3.2 Design ................................................ 20
      3.2.1 Assumptions ...................................... 20
      3.2.2 Algorithm ........................................ 21
   3.3 Implementation ....................................... 24
      3.3.1 Implementation Overview ......................... 24
      3.3.2 Kernel Modules ................................... 28
      3.3.3 Management Daemon System ....................... 31

4 Evaluation ............................................. 34
   4.1 Configuration ........................................ 34
   4.2 Iperf ................................................. 36
      4.2.1 P2V Bandwidth ................................... 36
      4.2.2 V2V Bandwidth ................................... 40
   4.3 NAS MPI benchmarks .................................. 42
      4.3.1 Micro Benchmarks ................................. 44
      4.3.2 Macro Benchmarks ................................. 46
<table>
<thead>
<tr>
<th>Contents</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5 Conclusion</td>
<td>49</td>
</tr>
<tr>
<td>5.1 Conclusion</td>
<td>49</td>
</tr>
<tr>
<td>5.2 Future Work</td>
<td>50</td>
</tr>
<tr>
<td>Bibliography</td>
<td>51</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Physical Machine and Virtual Machine Configuration 35
List of Figures

2.1 The Correctness Definition ............................................. 7
2.2 The Domino Effect ....................................................... 10
2.3 The Coordinated Protocols ............................................. 12

3.1 The SecondSite Project .................................................. 18
3.2 The Algorithm of Our Protocol ........................................ 22
3.3 The Xen Overview ....................................................... 25
3.4 The Change of the Ethernet Header ..................................... 26
3.5 The Implementation Overview ......................................... 27
3.6 The Bridge Mechanism .................................................. 29
3.7 The Kernel Modules ..................................................... 31
3.8 The Management Daemon System ...................................... 32

4.1 The Test Bed Configuration ............................................. 35
4.2 The P2V Network Bandwidth (disable and enable) .................... 37
4.3 The P2V Network Bandwidth (20 second interval) .................... 38
4.4 The P2V Network Bandwidth (50 second interval) .................... 39
4.5 The P2V Network Bandwidth (100 second interval) ................... 40
4.6 The V2V Network Bandwidth (disable and enable) .................... 41
4.7 The V2V Network Bandwidth (20 second interval) .................... 42
4.8 The V2V Network Bandwidth (50 second interval) .................... 43
4.9 The V2V Network Bandwidth (100 second interval) ................... 44
4.10 bt.A.4 runtime ............................................................ 45
4.11 sp.A.4 runtime ........................................................... 45
4.12 Repeating bt.A.4 100 times runtime ................................... 47
4.13 Repeating sp.A.4 100 times runtime ................................... 47
Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Norm Hutchinson, for his jokes, instructions, suggestions, help and kindness. For me, he is not only a great academic supervisor, but also my spiritual mentor.

I also feel so honored to work with Dr. Andrew Warfield and all the other SecondSite project members, Dr. Michael Feeley, Brendan, Geoffrey, Dutch, Gitika and Anoop. These brilliant guys open my eyes and teach me a lot.

I would also like to thank all the DSG members, Dr. Charles Krasic, Dr. Alan Wagner, Kan, Mike Tsai, Mike Wood, Andrei, Ani, Le, Shuang, Abhishek for their help and kindness along the course.

At last, I would like to thank my families, especially my fiancee Jing Dai. Without their support and encouragement, I cannot finish anything.
Chapter 1

Introduction

The Design of distributed snapshot protocols is a hot research topic and it plays a key role in the areas of disaster recovery and computer security of distributed systems [21] [27] [33]. Recently, with the popularity of parallel computing, this topic has received even more attention from both academic and the industrial researchers [14] [23] [19]. Although the number of existing snapshot protocols is huge and they differ a lot from one to another, there are several common requirements on them. The most important requirement of distributed snapshot protocols is correctness [14], which means these protocols have to guarantee they can find a set of globally consistent [15] snapshots from the taken snapshots. Transparency is also a key requirement. The snapshot targets should be unaware of the underlying snapshot protocols as much as possible. Another critical requirement is efficiency. Distributed snapshot protocols should not incur heavy overhead on the snapshot targets. For example, they should not tremendously show down the network or greatly increase the run time of snapshot targets.

Roughly, we can apply two criteria to classify the existing snapshot protocols [22]. According to the coordination strategy, we can classify existing snapshot protocols into uncoordinated protocols, coordinated protocols and communication-induced protocols. As its name hints, uncoordinated protocols need no coordination when taking snapshot, but they do need recovery methods to meet the correctness requirement. Compared with other kinds of protocols, they are simple in theory and very easy to implement. Unfortunately, they are
vulnerable to the domino effect [32], which makes them impractical in most cases. Unlike uncoordinated ones, coordinated protocols carefully coordinate before taking snapshots in order to make sure that the latest set of snapshots are always globally consistent. Although they are guaranteed to be correct and immune from the domino effect, they generate heavier overhead than uncoordinated protocols. And the communication induced protocols try to combine the advantages of coordinated and uncoordinated protocols by taking random snapshots and forced snapshots. However, they turn out to be inefficient in practice [10] [26].

Another classification criterion is snapshot granularity. According to it, the existing snapshot protocols can be classified into system level protocols, application level protocols and mixed level protocols [14]. System level protocols take snapshots by directly copying the raw memory data of target processes. Although system level protocols may consume a lot of space for snapshots due to their copying all the memory data of the processes, they are totally transparent to these processes. Application level protocols are trying to use a cleverer way to take snapshots by only recording some key variables of the target processes. They need not copy all the memory data of the processes, but they are not transparent at all to target processes, which means the target processes need to be modified to work with application level protocols. The mixed level protocols are something in the middle. They apply different snapshot methods to different types of variables of the processes, but they are also not totally transparent.

Although the distributed snapshot protocol has been well studied since 1970's, it is still facing new challenges from all kinds of systems especially when the virtual machine (VM) technology gets more and more popular recently. The SecondSite project [17] is such a system which incurs several unique requirements to the distributed snapshot protocol. The SecondSite project is
aiming at providing a novel disaster recovery solution to Web sites and other similar network service providers, with the help of VM technology. Its basic idea is continuously taking and storing distributed snapshots of the entire OS images of the target virtual machines which function as the platforms of Web site server programs. When the Web site crashes for any reason, the latest taken snapshots will be immediately restored on a backup Web site and the backup Web site will replace the crashed main Web site to provide services until the main Web site recovers. The project requires the distributed snapshot protocol to efficiently and transparently take snapshots of the entire OS images of target virtual machines, which cannot be satisfied by any known distributed snapshot protocol.

To meet the requirements of the SecondSite project and fix the problems of existing snapshot protocols, we design and implement a hypervisor level, coordinated non-blocking distributed snapshot protocol. As a component of the SecondSite project, our protocol supports continuously taking distributed snapshots of the OS images of target virtual machines. Basically, our snapshot protocol functions as follows. First, a snapshot initiator takes a snapshot of itself, and then updates its own epoch number and piggybacks this new epoch number on every outgoing message. When others receive a message, they will compare the epoch number in the message with their own epoch numbers. If the incoming epoch number is newer than theirs, they will block their network connections, take snapshots, update their epoch numbers and then unblock their connections. And if it is not, they will process the message as usual.

Though the design of our snapshot protocol seems very similar to the classic coordinated non-blocking distributed snapshot protocol [15], we still make several improvements according to our observations of the target environment. First, we intentionally ignore the channel states [15] because we notice that
most network connections are now TCP based, which means the ignored channel states will be automatically recovered by the TCP protocol. Second, we choose to piggyback the epoch number on all the outgoing messages, other than send out special coordination messages, to make our protocol functional in both the FIFO and non FIFO network environments. Third, our protocol does not require a special snapshot coordinator. In fact, each snapshot target can function as the snapshot initiator.

To implement our protocol as an efficient and transparent snapshot platform for target virtual machines, we create two kernel modules in the virtual machine control layer and a management daemon system. The two kernel modules finish most of the necessary functionality, such as piggybacking/extracting epoch numbers and blocking/unblocking network connections. Finally, the management daemon system is in charge of managing the snapshot related information and operating the kernel modules according to this information.

To evaluate our snapshot protocol, we test it with several well recognized benchmarks, including iperf, NAS, MPI benchmarks, etc. We use iperf to show our protocol's influence on the network bandwidth between target OS, as well as the bandwidth between target OS and remote clients. And we use the NAS MPI benchmarks to verify the correctness of our protocol and quantify the performance overhead of our protocol on the computation intensive and the network intensive applications. The experiment results prove the correctness of our snapshot protocol and shows that our protocol incurs negligible performance overhead specially with low snapshot frequency.

The rest of the thesis is organized as follows. Chapter 2 discusses the technical background of distributed snapshot protocols, introduces the classification of existing snapshot protocols and analyzes the merits and demerits of existing protocols. Chapter 3 details the design and implementation of our distributed
snapshot protocol. And in chapter 4, we test our snapshot protocol with several popular benchmarks and analyze the experiments. Chapter 5 gives a conclusion and illustrates future work.
Chapter 2

Related Work

Distributed snapshot is not a new topic at all. As early as the 1980's, Chandy and Lamport published their milestone paper [15] in this area, and after that, there appeared a huge number of projects working on this topic [21] [27] [33] [35]. Recently, with the popularity of parallel computing and disaster recovery, this topic has received more and more attention from both academic and industrial researchers [14] [23] [? ]. Although all the previous distributed snapshot projects are concerned with taking snapshots of processes, not entire OS images, their ideas and the theories behind their concrete protocols are still very helpful for designing our own protocol for taking snapshots of OS images.

2.1 Properties of distributed snapshot

protocols

The most important property of a distributed snapshots protocol is correctness [14], which means the captured distributed snapshots should be globally consistent. Chandy and Lamport [15] give the definition of globally consistent snapshots. Informally, we can describe globally consistent snapshots as snapshots that do not contain a message which is logged as being received by some process but not recorded as being sent out by any other process. Figure 2.1 shows an example of globally consistent snapshots and globally inconsistent snapshots respectively. Figure 2.1(a) is an example of globally consistent snapshots. In
Figure 2.1(a), P0's snapshot taken at time A records message m1 as having been sent out, while P1's snapshot taken at time B has no record of receiving m1. Although the message m1 is missing, this pair of snapshots is still globally consistent. Figure 2.1(b) shows a set of globally inconsistent snapshots. In P3's snapshot taken at time D, the message m2 is recorded as being received, but P2's snapshot at time C has no record that the message m2 has been sent out, which violates the definition of globally consistent snapshots.

Besides correctness, there are several other important properties of distributed snapshot protocols, including transparency and efficiency [14]. Transparency concerns the extent to which application developers need to be involved in the distributed snapshot protocol. Some protocols [7] are totally transparent to the applications working on them, which means the application developers need not change the source code of their applications at all to work with these distributed snapshot protocols. But some others protocols [8] are heavily application-involved and require the application developers to manually modify the applications' source code to perform such tasks as marking some variables and setting up global barriers, and recompile their applications. Also, there
are some protocols in the middle [14], which provide semi-automatic tools for developers to use to change their applications.

Efficiency is another important property of distributed snapshot protocols. Efficiency concerns the overhead caused by the protocols, and it is mainly composed of two parts, runtime overhead and space overhead. Runtime overhead is the extra runtime directly or indirectly caused by the distributed snapshot protocols and the space overhead is the amount of storage space used to store all the snapshots.

Generally speaking, we hope the distributed snapshot protocols could be totally transparent to upper level applications, so the applications could remain unchanged, but the transparency is tightly connected with the efficiency. In most cases, we are just looking for the best trade-off of these two properties.

2.2 Classification

Although the number of projects on distributed snapshot is huge, mainly we can use just two criteria, coordination strategy and snapshot granularity [22], to classify these projects.

2.2.1 Coordination Strategy

Based on how distributed snapshots are coordinated to achieve a globally consistent state, we can classify distributed snapshot protocols into three categories, uncoordinated protocols, coordinated protocols and communication induced protocols [28].

Uncoordinated Protocols

The basic idea of uncoordinated protocols is to allow the processes to take snapshots whenever they want, which means the processes can take snapshots
Chapter 2. Related Work

when the amount of information needing to be saved is small [35]. Because the processes take snapshots without any coordination, there is no guarantee that the latest snapshots of each process are globally consistent. As a result, the processes need to keep all the snapshots they have ever taken and we need some method to find a set of snapshots which are globally consistent. There are two methods that can be used to find a consistent set of snapshots. One is based on rollback dependency graphs [11], and the other is based on checkpoint graphs [35]. In fact, these two methods are equivalent, and they always generate the same set of snapshots [28].

Uncoordinated protocols have two obvious advantages. First, because uncoordinated protocols allow the processes to take snapshots when the amount of state information is small, the size of each snapshot would be smaller compared with other protocols. Second, since uncoordinated protocols need not coordinate all the processes, they usually generate a lower runtime overhead. But on the other hand, they have several serious disadvantages, for example, although some snapshots are useless, uncoordinated protocols still need to maintain all the snapshots the processes have ever taken because the latest snapshots of each process may not be globally consistent, and we may need some snapshots to form a globally consistent set, which a waste of storage space. Another more deadly problem of uncoordinated protocols is they are vulnerable to the domino effect [32], which means that when we are trying to find a set of globally consistent snapshots, we may have to go back to the initial snapshots of all the processes. Figure 2.2 shows an example of the domino effect. When the process P1 encounters a failure, it needs to restore to snapshot D (because it is the latest snapshot of process P1), and this restoration will invalidate the sending of message m2. As a result, the process P0 needs to restore back to snapshot C, which will invalidate the sending of message m1 and cause P1 to restore back to
B. For the same reason, P0 will restore back to A. At last, both the processes restore back to their initial snapshots. The domino effect problem is unsolvable for uncoordinated protocols, because the key reason behind the domino effect is the lack of coordination between the processes, which is also the key property of the uncoordinated protocols themselves.

![Figure 2.2: The Domino Effect](image)

**Coordinated Protocols**

Given the serious problems of uncoordinated protocols, people have determined that coordination between the processes is very important and necessary, which leads to the coordinated snapshot protocols. As their name hints, coordinated protocols are based on the coordination between the processes. The coordination could be some special control messages or some control information piggybacked on the normal messages. According to whether the protocol blocks communication between the processes, coordinated snapshot protocols can be classified into two categories, blocking ones and unblocking ones [28].

The blocking coordinated snapshot protocols use a straightforward strategy to take coordinated snapshots of the processes. Tamir and Sequin [33] introduced a blocking snapshot protocol, when the processes receive the snapshot command from a coordinator, they pause for a while and flush all the commu-
nication connections and inform the coordinator that they are ready to take snapshots; after receiving all the "ready" replies from the processes, the coordinator would broadcast a "commit" command to all the processes and the processes would each take a snapshot accordingly. Because the connections are flushed when taking the snapshots, we are sure that the snapshots taken are globally consistent. The only problem of blocking coordinated snapshot protocols is efficiency. Because they would block the communications between the processes every time when we want to take the snapshots, they could greatly slow down the whole system especially when the frequency of taking snapshots is high and the target application is network bound. As a result, people move to unblocking coordinated snapshot protocols, which would generate much less overhead during execution.

Unblocking coordinated snapshot protocols try to implement the coordination between the processes without blocking the communication between the processes. The most famous example is Chandy and Lamport's paper [15]. According to this paper, when the processes receive the marker, which functions as the snapshot request, from the coordinator, they would immediately take snapshots and rebroadcast out this marker to other processes before they send out any other messages, if this is the first marker they have received. As Figure 2.3(a) shows, when process P0 receives the marker from the coordinator, because this is the first time it received this marker, it would take a snapshot immediately and then broadcast this marker to other processes before it begins to send out normal messages. Similarly when P1 receives the marker from P0, it would take a snapshot immediately. Notice, when P1 receives the marker from the coordinator, because it has received the marker before, it would do nothing about it. This method is based on the assumption that the data channels are First-In-First-Out (FIFO). Otherwise, the snapshots may not be globally consis-
Chapter 2. Related Work

tent. For instance, in Figure 2.3(b), when the data channels are not FIFO, the message \( m1 \) may be sent out later than the marker but arrive earlier than the marker, which makes the snapshots contain some messages which have not yet been sent out. If the data channels are not FIFO, solution is to piggyback the marker on every message sent out by the processes [21] [27]. When a message arrives, before the processes act on the received message, they first determine whether the message contains a marker newer than the one they received last time. If it does, the processes would take snapshots before acting on this message, and piggyback this new marker on every out-going message. If the arriving message contains a marker older than the current marker of the process, this message would be treated as a part of the "state of the channel" [15].

![Figure 2.3: The Coordinated Protocols](image)

Coordinated protocols complicate the generating of snapshots, but simplify the recovery [34]. Unlike uncoordinated protocols, they need no special recovery methods to discover globally consistent sets of snapshots and they are immune to the domino effect. Moreover, since the latest snapshots of each process are always globally consistent, there is no need to keep multiple snapshots for every process, which saves a lot of storage space, although the size of each snapshot may be bigger than that of the uncoordinated protocols.
Communication Induced Protocols

Communication induced protocols [31] [18] [24] try to combine the advantages of uncoordinated protocols and coordinated protocols. They have two kinds of snapshots, local snapshots and forced snapshots [20]. On one hand, like uncoordinated protocols, communication induced protocols allow the processes to take local snapshots whenever they want. On the other hand, to avoid useless snapshots [10] and globally inconsistent state [15], the processes are induced to take forced snapshots when they detect there exist dangerous patterns which would incur useless or inconsistent snapshots. To detect these patterns, the processes need to communicate with each other about their current states, including their snapshot logs and messages record. Unlike coordinated protocols, they do not send out special coordination messages, but only piggyback coordination information on each out-going message. In [30], Netzer and Xu introduced the Z-path and Z-cycle theory to describe this “dangerous patterns detection” idea.

Theoretically, communication induced protocols have several obvious advantages over other kinds of protocols [13]. Compared with uncoordinated protocols, they are immune from the Domino effect. Compared with coordinated protocols, they give the processes considerable autonomy to decide when to take snapshots, which usually means smaller snapshots, and thus should incur less runtime overhead since they don't need global and strict coordination. Unfortunately, according to [10] [26], communication induced protocols turn out to be inefficient in practice. This is mainly because of the number of forced snapshots. The performance overhead is heavily dependent on the number of processes, the processes characteristics, and how the processes communicate with each other. It is also observed that the number of forced snapshots increases almost linearly with the number of processes.
2.2.2 Snapshot Granularity

Based on the granularity of snapshots, we can classify the distributed snapshot protocols into three categories, the application level protocols, the system level protocols and the mixed level protocols [14] [22].

System Level Protocols

System level protocols function as an infrastructure, which is totally invisible to the upper level target applications. They implement snapshots by directly copying the raw memory data of the applications to the stable storage devices, and the target applications can work with system level snapshot protocols without any modification. There are several ways to implement system level protocols. For example, Duell [7] and Barak [7] describe protocols that are modules in the operating system, and [23] and [25] are user level libraries, which can be linked into the target applications. The key advantage of this method is the target applications need no modification to work with them, making them the only choice when we cannot access the source code of the applications for some reason, but their problem is also obvious. When taking snapshots, they just naively copy all the raw memory data of the target applications to the storage devices, which means the snapshot size is completely dependent on the applications. If the application occupies a huge amount of memory, 1 Gigabyte for example, the size of each corresponding snapshot is also 1 Gigabyte.

Application Level Protocols

Application level protocols [8] are designed to take snapshots in a clever way. People notice that in some situations, there is no need to copy all the memory data of the processes. Sometimes, we only need to record some key variables, since the other parts of the processes would remain unchanged during execution.
Chapter 2. Related Work

This method would greatly reduce the size of the snapshots, but it also has a serious problem, we must manually modify the applications. For example, programmers need to mark some variables and setup global barriers. This kind of manual modifications is very dangerous and sometimes impossible.

Mixed Level Protocols

Mixed level snapshots were firstly introduced by Bronevetsky [14], and try to combine the advantages of application level protocols and system level protocols. As its name hints, its key idea is to apply application level protocols or system level protocols to different types of variables of the processes. This is based on the assumption that processes obey a fixed strategy to lay out different types of variables, such as the global variables, which are visible to the whole process, and the local variables, which are only accessible in a shared library. This fixed strategy makes it possible to use special compiling and linking commands to separate the different types of variables into isolated regions of memory, and then we can apply application level protocols or system level protocols to each region according to the characteristics of the variables in the region. For example, for the region containing the global variables, we can use application level protocols to take snapshots of these variables since it is very easy for the programmer to mark them in the source code; and for the local variables within some shared libraries, system level protocols are the only choice because application programmers usually have no access to the source code of these libraries.

Mixed level protocols are very "clever" snapshot methods, which combine the merits of application level protocols and system level protocols, and also provide a strategy to design a series of snapshot protocols according to the different characteristics of the applications [29]. Experiments show that the runtime overhead caused by the mixed level protocols is negligible and the snapshot size
is much smaller compared with the system level protocols [29].

2.3 Summary

As discussed above, there are already many distributed snapshot protocols, but none of them meet the requirements of the SecondSite project. As a result, we decide to design and implement our own distributed snapshot protocol which supports efficiently and transparently taking snapshots of virtual machines.
Chapter 3

Design and Implementation

This chapter firstly introduces the requirements of our distributed snapshot protocol, and then talks about the design and the implementation of our protocol. Generally speaking, to satisfy the requirements of the SecondSite project [17], our distributed snapshot protocol is designed and implemented as a hypervisor level, coordinated unblocking distributed snapshot protocol. Different from all the previous related protocols which are concerned with taking snapshots of processes, our protocol is aiming at taking snapshots of entire OS images using virtual machine technology. To implement our protocol, we designed and implemented a management daemon system and two kernel modules, `sch_cfifo` and `ebt_cptarget`.

3.1 Requirements

Before describing the requirements of our protocol, it is necessary to introduce the SecondSite project a little bit since our distributed snapshot protocol is a component of the SecondSite project and all the requirements of our protocols come from this project.

The general idea of existing disaster recovery solutions is taking snapshots of the server programs and copying these snapshots to a backup server. In the event of system failures, the server programs would restart from the most recently taken snapshot images to recover to the latest states. The serious
drawback of this method is the server programs or the underlying OS must be modified to work with the snapshot protocols, which is error prone and sometimes impossible.

![Diagram](image)

Figure 3.1: The SecondSite Project

The SecondSite project is aiming at providing a novel disaster recovery solution to Web sites and other network service providers, such as data centers and computing power providers. As Figure 3.1 shows, the main site is composed of several physical machines, each of which has several virtual machines running on it. The server programs and their OS are running on these virtual machines. The basic idea of the SecondSite project is periodically taking snapshots of the virtual machines of the main site and replicating these snapshots to the remote storage servers. When the main site crashes for any reason, these snapshots would be immediately restored in the backup site from the snapshots kept in the storage server and the network traffic router would redirect all the network traffic to the backup site. Recovery is thus totally transparent to the service users. In the SecondSite project, our distributed snapshot protocol is used to
take the distributed snapshots of the virtual machines in the main site.

Compared with existing disaster recovery solutions, the SecondSite project has several advantages. It requires no modifications to the server programs or their OS and the recovery process is totally transparent to the service users. However, these attractive advantages place several strict requirements on our distributed snapshot protocol, including correctness, efficiency and transparency.

Correctness is the fundamental requirement on our snapshot protocol. We have to guarantee that the taken snapshots are globally consistent and immune to the domino effect, which makes a coordinated protocol the only possible choice.

Another important requirement is efficiency. As mentioned above, in the SecondSite project we would take very frequent distributed snapshots and copy these snapshots to the remote storage servers. To avoid incurring too much runtime overhead in the original service, our snapshot protocol must be very efficient. Obviously, blocking coordinated protocols are not suitable here, because they would block all the network traffic every time a snapshot is taken. This behavior makes them very inefficient, especially when the snapshots are taken frequently. According to the analysis above, the unblocking coordinated snapshot protocols are the most suitable choices.

The last requirement is transparency. As introduced in the second chapter, the snapshot protocols can be classified into two categories, application level protocols and system level protocols. The application level protocols are not transparent at all and the system level protocols are only transparent to the target processes, not their OS. However, our snapshot protocol is required by the SecondSite project to be totally transparent to both the processes and the OS. Because it is different from all the previous related protocols, we decide to give our protocol a new name, and call it a hypervisor level protocol.
According to the requirements and analysis mentioned above, our snapshot protocol is designed and implemented as a hypervisor level, coordinated unblocking distributed snapshot protocol.

3.2 Design

3.2.1 Assumptions

Before introducing the design of our protocol in detail, it would be helpful to describe several important assumptions about the context of our snapshot protocol that greatly impact our design and implementation.

We first assume that all the target virtual machines are within the same subnet. We make this assumption based on the objects of the SecondSite project, which are web sites and other similar network service providers. In most cases, their servers are geographically located near each other for performance, security, and other reasons. Therefore, it is reasonable to assume all these servers are within the same subnet. This assumption simplifies the piggybacking of coordination information on messages, since in this case, the source MAC address in the Ethernet header is useless and can be used to hold the coordination information. If this assumption does not hold, we have to either piggyback the coordination information in the reserved bits of the message header, or hide the coordination information in the content field of every message. For example, we can hide coordination information in the first 4 bytes of the content field of each message. Unfortunately, these alternatives have several serious disadvantages. The reserved bits provide very limited capacity to hold the coordination information and it is dangerous to put information in these bits since other protocols may use them for their own reasons. And hiding the coordination information in the content fields of messages is also error prone, because the message for-
mats are greatly different for different protocols, and this operation will split a message into two when the original message uses up the entire content field and leaves no space for the coordination information.

The second assumption we made is that all the connections between target virtual machines are reliable, which means either they are TCP based, or they have their own lost message resent mechanisms. This assumption is based on the observation of the current network protocols, most of which are based on TCP, such as FTP, HTTP, SSH, TELNET, SMTP, etc. [5] and others have their own lost message recovery mechanisms built in. This assumption simplifies the design of our protocol, because all the lost messages will be resent, we can ignore the channel state [15] during the design of our protocol without worrying about losing messages during recovery.

The last assumption is we assume the connections between virtual machines are non-FIFO. This assumption is made for the reason of protocol robustness. As a result, our snapshot protocol can be used in both FIFO networks and non-FIFO networks.

3.2.2 Algorithm

As mentioned above, our environment requires that our protocol is a hypervisor level, coordinated unblocking distributed snapshot protocol. In fact, our design is greatly inspired by the existing coordinated unblocking protocols, such as [15] [21] [27]. In our protocol, we assign each virtual machine an epoch number, indicating the latest snapshot ever taken on this virtual machine. When the virtual machines communicate with each other, they piggyback their epoch numbers on every message. When receiving a message, the virtual machine first checks whether the epoch number piggybacked on the message is newer than its own epoch number before it really acts on the received message. If it is,
the virtual machine immediately blocks all the connections with other virtual machines, takes a snapshot, updates its epoch number and then finally unblocks the connections. If it is not, the message will be sent directly to the destination virtual machine. Our algorithm is shown in Figure 3.2:

**Snapshot initiator**

- Take a snapshot
- Increase the local epoch number by 1
- Piggyback the new epoch number on all outgoing messages

**Normal virtual machines**

- Check epoch number of all incoming messages
- If the received epoch number is newer than my epoch number
  - Block all the connections; take a snapshot, update my epoch number to the received one and unblock connections
  - Piggyback my epoch number on all outgoing messages

Figure 3.2: The Algorithm of Our Protocol

Although our protocol is very similar to some existing protocols, compared with these existing protocols, our protocol still has several distinguishing characteristics.

First, our protocol is aiming at transparently taking snapshots of entire OS images, while all the other snapshot protocols are only concerned with taking snapshots of target processes. This goal requires us to implement our protocol underneath the target operating systems, which leads us to the virtual machine
technology. Second, because we assume the data connections are not FIFO, our protocol has to piggyback the coordination information on all the outgoing messages, rather than simply sending out several special coordination messages. In our protocol, the piggybacked coordination information is the epoch number. Third, we intentionally ignore the channel state [15]. This decision is based on two things. First, the channel state has no impact on the correctness of the distributed snapshot protocols [15]. Second, we assume all the connections are reliable, which means all the missing messages caused by ignoring channel state will be retransmitted by their sender according to the TCP protocol or their own lost message resent mechanisms. At last, we do not require a special snapshot coordinator. Any virtual machine can function as the snapshot initiator, which makes our protocol more robust.

Another important concept worth mentioning here is the snapshot group. During the design, it was observed that we can easily extend our snapshot protocol to provide the disaster recovery service to multiple sites simultaneously. To achieve this goal, we introduce the concept of snapshot group. The snapshot group is a group of guest domains belonging to the same site. Our snapshot protocol only needs to guarantee that the snapshots of the guest domains within the same snapshot group are globally consistent. To support snapshot groups, we only need to design a management daemon system to record the member information of each snapshot group and ask it to setup the kernel modules according to the information of each snapshot group.
3.3 Implementation

3.3.1 Implementation Overview

Since our snapshot protocol is a component of the SecondSite project, the implementation of our protocol is heavily dependent on the implementation of the SecondSite project. Therefore, it is very necessary to first introduce the implementation of the SecondSite project. The SecondSite project is aiming at providing a novel disaster recovery solution with the help of virtual machines. The virtual machine system chosen by the SecondSite project is Xen [12], which is the most popular open source virtual machine system. The overall structure of Xen is shown in Figure 3.3. As Figure 3.3 shows, domain 0 is a special control domain which is in charge of the management of other guest domains. This includes creating, destroying and modifying other guest domains. At the same time, domain 0 functions as a network gateway for the guest domains. All network messages heading to the guest domains first go through domain 0, and are then dispatched by domain 0 according to their destinations.

Having discussed the implementation background of our snapshot protocol, we can move to the implementation of our protocol. However, before implementing our protocol, we have to answer several critical questions about our implementation.

The first question is how to piggyback the epoch number on each message. Initially, we planned to steal several reserved bits from the IP headers or the Ethernet headers of the messages, but later we found out it was dangerous to piggyback our epoch numbers in these reserved bits because some other protocols may also use these reserved bits for their own reasons. Moreover, these reserved bits provide very limited capacity to piggyback our epoch numbers (they are only 3 or 4 bits long at most). Our solution to this problem is based on the first assumption we mentioned above. We assume that all the target virtual machines
are within the same subnet and the messages between virtual machines do not go through any device which will rewrite the source MAC addresses, such as gateways. Under this assumption, for the messages between virtual machines, their source MAC addresses are "useless" and will remain unchanged during the message transmission. Therefore, we decide to piggyback the epoch numbers in the source MAC address of the Ethernet header. This method is superior in several respects to our original idea. First, the source MAC address, being 48 bits long, is much wider than the reserved bits. Second, it is much easier to piggyback our epoch numbers in source MAC address than hide the epoch numbers in the reserved bits. In fact, the **Etables** program has provided the Source Network Address Translation (SNAT) functionality, with which we can easily change the source MAC address. In our implementation, we do not use up all the 48 bits to hold the epoch numbers. We only use half of them (24 bits)
for piggybacking the epoch numbers, and we use the remaining 24 bits to hold MAC address IDs, which are the representatives of the real MAC addresses and will be translated back to the real MAC addresses when the messages arrive at their destination virtual machines. Figure 3.4 shows the changes of the source MAC address field during message transmission. Obviously, we need to create and manage the mapping table between the MAC address IDs and the real MAC addresses. In our implementation, we extend the management daemon system to accomplish this job.

The second question is where to implement our snapshot protocol. Our distributed snapshot protocol is required to be totally transparent to the processes and the OS running in guest domains, so we have to implement our protocol underneath the guest OS. We already know that domain 0 is the special control domain of the Xen system and more importantly, all the network messages heading to the guest domains will go through domain 0 first, all of which makes the domain 0 a perfect choice to implement our snapshot protocol.

The last question we need to answer before implementing our snapshot pro-
tocol is how to block and later unblock all the connections of a guest domain. According to our algorithm, when a virtual machine receives a message with a newer epoch number, all the connections of the virtual machine must be blocked before we can begin to take a snapshot of the virtual machine, and after finishing the snapshot and updating the local epoch number, we need to unblock these blocked connections. We implement these blocking and unblocking functionalities by adding a new kernel module, `sch_cfifo`, in the traffic control (tc) [9] module of the domain 0 kernel. This `sch_cfifo` module checks the epoch numbers contained in each incoming message and determines whether it is newer than the epoch number of the target virtual machine. If it is, the `sch_cfifo` module blocks all the messages to this virtual machine until it receives an unblock command.

![Figure 3.5: The Implementation Overview](image)

After answering the above questions, the implementation of our snapshot protocol becomes clear to us. As Figure 3.5 shows, the implementation consists of several kernel modules in the domain 0 kernel and a user level management daemon system. The kernel modules are used to piggyback the epoch number on every outgoing message, map MAC addresses to MAC IDs, extract the epoch
number from each incoming message, block/unblock the network connections and communicate with the management daemon system. The management daemon system is in charge of the snapshot information management and interacts with the kernel modules. For example, the daemon system records the snapshot group information and also manages the mapping table between the real MAC addresses and the MAC IDs. The kernel modules and the user level management daemon system will be discussed in detail in the following sections.

3.3.2 Kernel Modules

To implement our snapshot protocol, two kernel modules, ebt_cptarget and sch Cfio, and their corresponding user level control programs are designed and implemented. These two kernel modules are the most important components of the implementation and provide most of the necessary functionality of the snapshot system, including piggybacking/extracting epoch number, blocking/unblocking network connections and interacting with the user level management daemon system.

Before introducing the implementation of these kernel modules, it would be helpful to first explain the basic network topology of Xen and how Xen manages the network traffic heading to domain 0 and other guest domains. As Figure 6 shows, each network device in a guest domain is directly connected to a virtual interface (vif) network device in domain 0 through a "virtual crossover cable" [6]. The name of the vif device, vifX.Y, indicates the guest domain ID and the network device ID. For example, device vif3.0 is connecting to the first network device of the guest domain whose domain ID is 3. Domain 0 treats these vif devices as normal network devices and supports standard methods to operate these devices, such as setting up Network Address Translation (NAT), using traffic control tools [9] to limit network bandwidth, and building the routing
table via the `route` command. Xen itself provides several mechanisms, such as bridging and routing, to manage the network traffic, and bridging is the default option among them. Our distributed snapshot protocol is implemented based on the bridging mechanism.

![Bridge Mechanism Diagram](image-url)

Figure 3.6: The Bridge Mechanism

Figure 3.6 describes the basic idea of the bridging mechanism. In bridging, all the vif devices are directly connected to the *Linux Ethernet bridge*. The *Linux Ethernet bridge* is a kernel utility which mimics the behavior of real hardware bridges and provides more powerful functionalities [3]. When messages arrive in domain 0 from vif devices, they will be first sent to the *Linux Ethernet bridge* and the *bridge* will forwarded them to different interfaces according to their destination MAC addresses. For example, when domain 1 sends out a message from its `eth0` device, this message will first arrive in the vif device `vif1.0` and then be sent to the *Linux Ethernet bridge*. After receiving this message, the *bridge* will route this message according to its destination MAC address.
To implement our design targets, piggybacking/extracting epoch numbers and blocking/unblocking network connections, we resort to two Linux utilities, `Ebtables` and `tc`. `Ebtables` [1] is a powerful tool for the Linux Ethernet bridge, which provides several useful abilities, such as filtering messages according to their MAC addresses, altering the messages' source or destination MAC address and counting passing messages. Although we can directly use the Source MAC address NAT (SNAT) ability to piggyback the epoch number on each outgoing message, we need to implement a new module, `ebt_cptarget`, which extracts the epoch numbers from source MAC addresses of arriving messages and restores the real source MAC addresses from the MAC IDs. Another tool we use to block/unblock network connections is `tc`. `tc` is a network traffic control tool in the Linux kernel, which provides functionalities such as slowing down network traffic, simulating physical congestion, adjusting the order of sending out messages, etc. It works on the messages already queued on each network device, and to implement the blocking/unblocking functions, we add a new module, `sch_cfifo`, into `tc`.

The functionalities of these two kernel modules and their relationship are shown in Figure 3.7. When a message heading to a guest domain arrives, it will go through the `ebt_cptarget` module first. The `ebt_cptarget` module first extracts the epoch number from the source MAC address field of the Ethernet header and stores this epoch number in the `nfmark` area of the message's `sk_buff` structure, then it restores the real source MAC address from the MAC ID contained in the message. After these operations, the message is then sent to the Linux Ethernet bridge, which forwards the messages to different network devices according to their destination MAC addresses. After this routing, the messages arrive in the `sch_cfifo` module, which contains a FIFO buffer pool. The arriving message will be put into this buffer pool if the pool is not full.
Then the `sch_cfifo` module takes a peek at the first message in the buffer pool and check whether the epoch number contained with this message is newer than that of the destination guest domain. If it is not, this message will be sent to the destination guest domain immediately, and the `sch_cfifo` module will proceed to check the next message in the buffer pool. If the epoch number is newer, the `sch_cfifo` module will stop dequeuing messages from the buffer pool, which means that it blocks all the messages heading to this virtual machine, and notifies the upper level management daemon system. This blocking would continue until the management daemon system finishes the snapshot process and asks the `sch_cfifo` module to unblock.

![Diagram](image)

**Figure 3.7: The Kernel Modules**

### 3.3.3 Management Daemon System

The management daemon system is the management unit of the implementation. It is in charge of managing all the snapshot related information. For example,
it manages the mapping table between the real MAC addresses and the MAC IDs and it also records the member information of each snapshot group.

As shown in Figure 3.8, the management daemon system is a distributed system and consists of three components, \texttt{cp-command}, \texttt{central_server} and \texttt{local_daemon}. \texttt{cp-command} is a command program, by which users send out all kinds of commands to the other components, such as creating, modifying and deleting snapshot groups. The \texttt{central_server} is the control unit of this management daemon system. It receives and analyzes the commands from \texttt{cp-command}, then distributes translated commands to the involved \texttt{local_daemon} on each physical host to complete each user command. The \texttt{local_daemon} is the functional unit of the management daemon system, which is running on every domain 0. On one hand, it receives the commands from the \texttt{central_server}, takes actions accordingly and sends results back to the \texttt{central_server}. On the other hand, it interacts with the two kernel modules and the snapshot pro-
gram. It sets up the two kernel modules according to the information from the
central_server, and forwards notifications from the sch_cfifo module to the
snapshot program when a newer epoch number appears. When the snapshot is
finished, it commands the kernel modules to unblock the network connections.
Chapter 4

Evaluation

To give a thorough test of our design and implementation, we apply several popular benchmarks on our protocols. We choose iperf to evaluate the bandwidth impact caused by our protocol, and we use NAS MPI benchmarks to verify the correctness of our protocol and quantitate the overhead incurred by our protocol. The experimental results prove the correctness of our protocol and also show that the overhead caused by our protocol is negligible especially with low snapshot frequency.

4.1 Configuration

Before analyzing the experimental results, it is necessary to introduce the configuration of our experimental environment.

As Figure 4.1 shows, there are two physical machines in our configuration which are connected by a one gigabit Ethernet network. Each of the physical machines has two virtual machines running on it, and all the virtual machines and physical machines are connected with each other based on the bridging mechanism provided by Xen. Table 4.1 shows the configuration details of the virtual machines and the physical machines.

Besides introducing the experimental environment configuration, it is also necessary to explain how these virtual machines take snapshots and how they coordinate with each other to achieve globally consistent snapshots, which has
great impact on our experimental results. In our experiments, the virtual machine g3 functions as the snapshot initiator, which periodically starts the snapshot procedure of itself and all the other three virtual machines. To avoid generating huge delay between the snapshots of the same set globally consistent snapshots, the snapshot initiator g3 keeps broadcasting messages to other virtual machines. As a result, after g3 finishes taking a snapshot and updating its epoch number, all the other virtual machines will immediately receive the messages, containing the new epoch number, from g3 and will take snapshots accordingly. In fact, these normal virtual machines are taking snapshots almost at the same time, because in such a small network environment, the broadcast messages from g3 arrive in other virtual machines almost at the same time.
4.2 Iperf

iperf [2] is a benchmark used to measure the TCP bandwidth of networks with different parameters, which is able to report several meaningful figures of networks, such as bandwidth, datagram loss rate, delay, etc. We apply iperf to evaluate the impact caused by our protocol on the network bandwidth. And to thoroughly test its influences, we measure the bandwidths between target virtual machines (V2V bandwidths) as well as the bandwidth between a physical machine and a target virtual machine (P2V bandwidth) under different snapshot frequencies.

4.2.1 P2V Bandwidth

The first bandwidth we measure is the P2V bandwidth between the physical machine c2 and the virtual machine g6. To accurately test the impact of our protocol, we measure the P2V bandwidth under different snapshot frequencies.

Figure 4.2 shows the P2V bandwidth when we enable and disable our protocol. Enabling our protocol basically means that all the messages will be piggybacked with epoch numbers and when these messages arrive in their destinations, the piggybacked epoch numbers will be extracted and their source MAC addresses will be restored. But we do not take snapshots in this case. According to Figure 4.2, the average bandwidth between c2 and g6 is around 505 Mb/s when disabling our protocol, and the average bandwidth drops to 485 Mb/s while enabling our protocol. The average bandwidth decreasing is less than 5 percent, which is very impressive.

Figure 4.3 shows the network bandwidth between c2 and g6 when taking snapshots every 20 seconds. According to this figure, when we take snapshots, the network bandwidth will drop to zero, because the network connections of the target virtual machines are blocked during the process of taking snapshots.
When the snapshots are finished, the network bandwidth will recover to normal because the blocked network connections will be unblocked after the snapshots are finished. From Figure 4.3, we can see that the network blocking time (when the network bandwidth is 0) is around 6 seconds, which is out of our expectation, since usually we only need about 3 seconds to take a snapshot of a single virtual machine. The reason for this long time blocking is the order of these virtual machines being taken snapshots. Because the normal virtual machines, g4, g5 and g6, take snapshots almost at the same time, the physical machine c2 needs longer time (around 6 seconds) to finish taking snapshots of both g5 and g6. As a result, there is a 6 second long network blocking about every 20 seconds. Because the snapshots happen very frequently, the average P2V bandwidth drops to 291 Mb/second.

Figure 4.4 and Figure 4.5 show the P2V bandwidth between c2 and g6 when
the snapshot intervals are 50 seconds and 100 seconds respectively. In fact, these
two figures are very similar to Figure 4.3, which has been discussed above. In
Figure 4.4 and Figure 4.5, the blocking time is still around 6 seconds and the
bandwidth will recover after the snapshots are finished. However, due to the
longer snapshot intervals, the average bandwidths shown here are much high
than that shown in Figure 4.3. In Figure 4.4, the average bandwidth recovers
to 410 Mb/second. And in Figure 4.5, the average bandwidth is further restore
back to 449 Mb/second, which only decreases about 10 percent compared with
the original 505 MB/second bandwidth. Based on these observations, we can
predict that the decreasing of average bandwidth caused by our protocol will
keep diminishing when the snapshot interval gets bigger.
Figure 4.4: The P2V Network Bandwidth (50 second interval)

Figure 4.5: The P2V Network Bandwidth (100 second interval)
4.2.2 V2V Bandwidth

Another bandwidth we test is the V2V bandwidth between the virtual machine g4 and the virtual machine g5. Figure F shows the V2V bandwidth between g4 and g5 when we enable and disable our protocol. As it shown, the average V2V bandwidth is about 809 Mb/second when disabling our protocol, and the average bandwidth drops to 798 Mb/second if we enable our protocol. The average bandwidth drop is less than 2 percent.

Figure 4.6: The V2V Network Bandwidth (disable and enable)

Figure 4.7 shows the bandwidth between g4 and g5 when snapshots happen every 20 seconds. Because we take snapshots very frequently, the average bandwidth dramatically drops to 467 Mb/second. In Figure 4.7, there are several interesting observations worth mentioning.

First, we find out the network blocking time is still around 6 seconds. Although g4 only needs about 3 seconds to finish its snapshots, g5 does need around 6 seconds to finish his, because the two virtual machines g5 and g6,
which are running on the same physical machine c2, are usually taking snapshots together and c2 needs about 6 seconds, not 3 seconds, to finish taking snapshots of both virtual machines. As a result, the network blocking time still appears to be 6 seconds.

Second, we notice there are always some bandwidth vibrations before each network blocking. In fact, these bandwidth vibrations are caused by the virtual machine g3, although g3 seems unrelated with the bandwidth between g4 and g5. As mentioned above, the snapshot initiator g3 always takes snapshots right before the other virtual machines take their. Moreover, our snapshot program consumes a lot of memory and computing power when taking snapshots, which at the same time reduces the hardware resources that virtual machine g4 can use, and further impacts the bandwidth between g4 and g5. This explains why there are always bandwidth vibrations before the network blocking in Figure 4.7.
Third, the intervals between network blockings are uneven. For example, in Figure 4.7, the interval between the first network blocking and the second one is only 13 seconds, and the interval between the second one and the third one is around 26 seconds. These uneven intervals are caused by the way that our snapshot program takes snapshots. Like the method mentioned in [16], our snapshot program repeatedly scans and copy the main memory of target virtual machine until the final round, which means the amount of time that our snapshot program consumes to finish taking snapshots varies in different situations. When the virtual machines are still, our snapshot program may be able to finish a snapshot very quickly. But if the target virtual machines are active, our snapshot program may need a longer time to finish taking snapshots. In our experiment, our snapshot initiator g3 needs uncertain amount of time to finish its snapshot, which causes the intervals between network blockings to be uneven.

Figure 4.8 and Figure 4.9 show the bandwidths between g4 and g5 when the snapshot interval is 50 seconds and 100 seconds respectively. These two figures clearly show us the bandwidth vibrations before each network blocking. And with the increasing of the snapshot interval, the average bandwidth decreasing caused by our protocol keeps dropping. For example, when we take snapshots every 50 seconds, the average bandwidth is around 660 Mb/second, and when the snapshot interval becomes 100 seconds, the average bandwidth increases to 740 Mb/second.

4.3 NAS MPI benchmarks

The second kind of benchmarks we applied on our protocol is the NAS MPI benchmarks [4], which are a set of parallel benchmarks designed to evaluate the performance of supercomputers. In our experiments, we use the NAS MPI
Figure 4.8: The V2V Network Bandwidth (50 second interval)

benchmarks to verify the correctness of our protocol and evaluate the overhead caused by our protocol.

To verify the correctness of our protocol, we run the NAS MPI benchmark programs in our experimental environment, and take several snapshots during their executions. Then, we randomly pick one set of snapshots and restore our virtual machines to these snapshots. By checking whether the NAS MPI benchmarks resume running from these snapshot points and comparing their results with those generated in the ordinary environments, we draw a conclusion that our protocol is completely correct.

To evaluate the overhead caused by our protocol, we measure the runtime of NAS MPI benchmarks under different situations.
4.3.1 Micro Benchmarks

First, we execute the NAS MPI benchmarks only once and measure the runtime of these benchmarks under different situations. By comparing the runtime generated in different cases, we can quantitate the overhead caused by our protocols and further prove the efficiency of our protocol. As Figure 4.10 and Figure 4.11 show, we first measure these benchmarks’ runtime when they are executed on 4 physical machines (the configuration of these physical machines is described in Table 4.1). Then, we run these benchmarks in our experimental environment and record their runtime without enabling our protocol. By comparing these two kinds of runtime, we find out the runtime generated in our test environment is almost twice as long as that generated in 4 physical machines environment, which is reasonable because in our test environment, there are only 2 physical machines.
Chapter 4. Evaluation

bt.A.4

Figure 4.10: bt.A.4 runtime

sp.A.4

Figure 4.11: sp.A.4 runtime
Then, we enable our protocol and measure the runtime. Comparing with the disabled case, the runtime increase is less than 2 percent. After this, we start taking snapshots and evaluate the runtime with different snapshot intervals, including 30 seconds, 50 seconds and 100 seconds. From Figure 4.10 and Figure 4.11, we find out when the snapshot interval is 30 seconds, the runtime dramatically increases, but when we increase the snapshot interval, the runtime overhead caused by our protocol begins to drop. When we take snapshot every 100 seconds, the runtime increase is already less than 15 percent comparing with the “disabled” situation. And we are sure if we continue increasing the snapshot interval, the runtime overhead caused by our protocol will keep dropping.

4.3.2 Macro Benchmarks

In fact, taking snapshots every 50 seconds or even 100 seconds is not a wise way to use our protocol to protect the real world MPI applications which usually run for hours and even days. In this case, we are supposed to take snapshots every half and hour or every hour, so when system crashes for any reason, the virtual machines can recover to the latest set of snapshots and only lose at most one hour’s work. To test our protocol’s performance under this case, we repeat the NAS MPI benchmarks 100 times and record their runtime under different situations. As Figure 4.12 and Figure 4.13 show, we first measure their runtime when executing them on 4 physical machines environment, and then comparing to those generated in our test environment while disabling our protocol. We find out our test environment doubles the benchmarks’ runtime, which is reasonable since we only have two physical machines in our test environment, but in the another case, there are 4.
**Chapter 4. Evaluation**

**bt.A.4 (repeat 100 times)**

<table>
<thead>
<tr>
<th>Physical Machines</th>
<th>Virtual Machines (disabled)</th>
<th>Virtual Machines (enabled)</th>
<th>Virtual Machines (1800s interval)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6375 s</td>
<td>11075 s</td>
<td>11360 s</td>
<td>11394 s</td>
</tr>
</tbody>
</table>

Figure 4.12: Repeating bt.A.4 100 times runtime

**sp.A.4 (repeat 100 times)**

<table>
<thead>
<tr>
<th>Physical Machines</th>
<th>Virtual Machines (disabled)</th>
<th>Virtual Machines (enabled)</th>
<th>Virtual Machines (1800s interval)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8218 s</td>
<td>15439 s</td>
<td>15737 s</td>
<td>15779 s</td>
</tr>
</tbody>
</table>

Figure 4.13: Repeating sp.A.4 100 times runtime
Then we measure the runtime in our test environment with our protocol enabled and taking snapshots every half an hour. According to Figure 4.12 and Figure 4.13, when we take snapshots every half an hour, the runtime increase is less than 3 percent comparing with the "disabled" case, which shows the high efficiency of our protocol. Based on this observation, we predicate that if we taking snapshots every hour, this negligible runtime overhead caused by our protocol will drop further.
Chapter 5

Conclusion

5.1 Conclusion

This thesis presents the design and implementation of a hypervisor level coordinated distributed snapshot protocol as a part of the SecondSite project. Different from all the existing distributed snapshot protocols, our protocol is aiming at transparently taking snapshots of entire virtual machine OS images. Comparing with the known snapshot protocols, our protocol is simpler in design because we make several assumptions based on our target environment. And to make our protocol totally transparent to the target virtual machines, we choose to implement our protocol in the control domain of the Xen virtual machine system. We create two kernel modules, sch Cfifo and ebt_cptarget, and their corresponding user level programs which function as the key components of our protocol and provide most of the necessary functionality of our snapshot protocol, including piggybacking and extracting epoch numbers, and blocking and unblocking network connections. And to invoke these two modules and maintain the snapshot related information, we implement a user level daemon system which is the management unit in our implementation.

We apply several popular benchmarks on our protocol to verify the correctness of our protocol and evaluate the overhead caused by it. A huge amount of experimental results prove the correctness of our protocol and show that our protocol only incurs negligible overhead on these benchmarks especially with
big snapshot intervals.

\section*{5.2 Future Work}

Although our protocol has basically met the requirements of the SecondSite project, there are still several directions for future work. First, we plan to accelerate our snapshot program. Now, our snapshot program needs around 3 seconds to finish taking a snapshot of a 128 MB virtual machine, which is not satisfactory especially when there are several virtual machines taking snapshots at the same time. Second, we need more experiments to thoroughly test our protocol. Although our protocol has been evaluated with several popular benchmarks, we are planning to apply more benchmarks, such as the SpecWeb benchmark and the dkftpbench, to measure the performance of our protocol with different kinds of workload. Another interesting future work direction is to extend our snapshot protocol to work with the new functionalities of the SecondSite project. We are now working on extending the SecondSite project to provide several novel security services, such as "1 day" patch. To work with these new services, we need to extend our snapshot protocol correspondingly.
Bibliography


[16] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *NSDI’05: Proceedings of the 2nd conference on*


