Distributed Checkpointing of Virtual Machines in Xen Framework

Thesis Submitted In Partial Fulfillment Of The Requirements
For The Degree Of

Master of Technology
in
Computer Science and Engineering

by **Sankalp Agarwal**

Roll No: 03CS3023

under the guidance of

Prof. Arobinda Gupta

Department of Computer Science and Engineering

Indian Institute of Technology

Kharagpur

May 2008

Certificate

This is to certify that the thesis titled **Distributed Checkpointing of Virtual Machines in Xen Framework** submitted by **Sankalp Agarwal** to the Department of Computer Science and Engineering in partial fulfillment for the award of the degree of **Master of Technology** is a bonafide record of work carried out by him under my supervision and guidance. The thesis has fulfilled all the requirements as per the regulations of this Institute and, in my opinion, has reached the standard needed for submission.

Prof. Arobinda Gupta

Dept. of Computer Science and Engineering

Indian Institute of Technology

Kharagpur 721302, INDIA

May 2008

Acknowledgments

In the beginning, I thank my supervisor, Prof. Arobinda Gupta for all his guidance and support. I would thank him for withstanding all my shortcomings, answering all my doubts, and being super-patient with me. He has been much more than an advisor or a teacher to most of the students associated with the Department. His humility and knowledge is worth admiring and learning from. I must also thank Prof. A. Das for the temperament and zeal he has shown as my advisor. His constant encouragement enabled me to learn and grow.

I thank Prof. A. Pal, Prof. N. Ganguly, Prof. R. Kumar, Prof. R. Mall and Goutam Biswas Sir for their constructive criticism of the project during the evaluations and otherwise.

In the Department, it has been a treat learning under teachers like Prof. P. Dasgupta, Prof. A. Bishnu, Prof. S. P. Pal, Prof. A. Basu & Prof. D. Sarkar. I would like to thank all the professors especially Prof. N. Ganguly, Prof. S. Sarkar, Prof. P.P. Chakraborty & Prof. I. Sengupta for taking up the initative of building the alumni network of the department.

In the Software Lab, I would like to thank Prasun Sir, Shibobroto Sir, Atanu Sir and Prasenjit Sir for all they have done for us students.

I still remember the wonderful time I spent at an Intern at LabOS, EPFL, Switzerland where I met and worked with leading researchers. I fondly remember and thank Prof. Willy Zwaenepoel and Aravind Menon for their guidance.

I am grateful to Mukesh, Tathagata, Mayank, Piyush, Udit, Joydeep, Arpit, Umang, Dani, Pankaj, Puspesh, Nitin, Pranith, Swapandeep, Vikas, Mayank Varsh-

ney for being best depmates ever.

Immense thanks to my wingmates Anuj, Akshay, Anshuman, Arun, Bineet, Deepak, Kapil, Nandlal, Piyush, Surendra, Shonam for making my stay at Kharagpur so very memorable. Saket, Abhishek, Tushar, Ramprasad, Ashish, Gaurav and other juniors at Azad Hall of Residence have been great sources of fun. Thank you guys for all these years.

Lastly, thank you, Maa and Aayush. Its been very painful staying away from you two, but the stay was bearable only because of your support.

Sankalp Agarwal

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

May 2008

Abstract

A computing system consists of a multitude of hardware and software components that are bound to fail eventually. In many systems, such component failures can lead to unanticipated, potentially disruptive failure behavior and to service unavailability. In distributed system as the size of system grows, so does the probability that some component may fail. Recovering from such failures is notoriously difficult and is important in the design and development of reliable systems, applications & protocols. Checkpointing and rollback recovery is a widely used scheme for dealing with failures at the application level. Extension of these distributed checkpointing protocols from application level to the virtual operating systems in XEN framework is not implemented so far. The aim of this work is to provide users/applications with a checkpointing library along with few implemented basic checkpointing algorithm, synchronous & asynchronous, which would enable them to checkpoint a group of VMs that are communicating among themselves.

Contents

List of Figures iv

List of Algorithms v

1 Introduction 1

1.1 Motivation 2

1.2 Problem Definition 3

1.3 Contributions 4

1.4 Thesis Outline 4

2 Background & Related Work 5

2.1 Virtual Machines 5

2.1.1 Full Virtualization 7

2.1.2 Paravirtualization 7

2.1.3 Virtualization on the OS level 7

2.2 Xen 8

2.3 Checkpointing and Recovery 14

2.4 Checkpointing and Recovery in Xen 16

3 Distributed Checkpoint and Recovery of VMs in Xen 20

3.1 Architecture of Xen Network Split Device Driver 20

3.1.1 Transmission of packets 22

3.1.2 Reception of Packets 23

3.2 Synchronous Checkpoint and Recovery 24

3.2.1 System Model 24

3.2.2 Checkpoint Algorithm 25

3.2.3 Recovery Algorithm 26

3.2.4 Proof of Correctness 26

3.2.5 Complexity 27

3.2.6 Plan for Testing 27

3.3 Asynchronous Checkpointing and Recovery 29

3.3.1 System Model 30

3.3.2 Checkpoint Algorithm 30

3.3.3 Recovery Algorithm 30

3.3.4 Complexity 33

3.3.5 Proof of Correctness 33

3.3.6 Plan for Testing 34

3.4 Implementation of Checkpoint and Recovery Library in Xen 35

3.4.1 Functions 35

3.4.2 Plan for Testing 37

3.5 Challenges Faced 37

4 Conclusion and Future Work **39**

4.1 Roads Travelled 39

4.2 Contributions 40

4.3 Future Directions 40

Bibliography **42**

List of Figures

2.1 Xen Architecture 9

2.2 Xen runs in ring 0, Guest Domain runs in ring 1, User applications run in ring 4 12

2.3 The Split Device Driver Model in Xen 13

List of Algorithms

1 Rollback Recovery Algorithm 32

Chapter 1

Introduction

Virtualization is a widely used technique in which a software layer mutliplexes lower-level resources among higher level software applications and systems. Examples of virtual systems include a vast body of work in the area of operating systems[14, 17, 16, 13, 8, 4] , high level language virtual machines such as those for Java and .NET and more recently, Virtual Machine Monitors (VMMs). Hardware Virtualization, which is old technology of 1960's, is experiencing a revival in both industry and research communities in form of VMM. VMM virtualizes entire software stack including operating system and application, via a software layer betwen hadrware and OS of the machine. VMMs offer a wide range of benefits including application and full system isolation, OS based migration, distributed load balancing, performance isolation, security, checkpointing and recovery, non-native application execution and suport for multiple or customized operating systems. This use of virtualization can improve reliability, flexibility and recovery time after OS crashes. From now on, whenever we refer to virtualization, it refers to Hardware Virtualization using VMMs and Virtual Machine (VM) will refer to instance of virtualized OS.

Distributed systems have become popular because of several advantages they

offer over centralized ones. They enable many applications, including client-server systems, transaction processing, World Wide Web, scientific computation, etc. The vast computing potential of these systems is often hampered by their susceptibility to failures. Many techniques have been developed to add reliability and high availability to distrubted systems. One of them is Checkpointing and Recovery. Checkpoint is saved local state of a processor. All the local checkpoints, one from each site, collectively form a global checkpoint. A global checkpoint is called a consistent set of checkpoints, if every message recorded as received in a local checkpoint is also recorded as sent in another local checkpoint that constitute the same global checkpoint.

There are 2 approaches towards distributed checkpointing and recovery, synchronous checkpointing and asynchronous checkpointing. The synchronous checkpointing is to ensure that all processes keep local checkpoint in stable storage and coordinate their local checkpoint action such that global checkpoint is guaranteed to be consistent. When a failure occurs, processors roll back and restart from their most recent checkpoints. In asynchronous approach each processor takes local checkpoints independently and a consistent global state is constructed using the checkpoints during recovery.

1.1 Motivation

Numerous systems have been designed which use virtualization to subdivide the ample resources of a modern computer. Some require specialized hardware, or cannot support commodity operating systems. Some target 100% binary compatibility at the expense of performance. Others sacrifice security or functionality for speed. Few offer resource isolation or performance guarantees; most provide only best-effort provisioning, risking denial of service. Xen is a VMM produced by Univer-

sity of Cambridge Computer Laboratory and released undger GNU General Public License. Xen allows multiple commodity operating systems to share conventional hardware in a safe way with minimal performance and functionality loss. Xen VMM has become one of the popular hypervisor due to performance benefits it offers over other similar systems like VMware, VirtualPC etc. It is emerging as an open standard for virtualization on x86 systems. Xen VMM works on IA-32, x86-64, IA-64 and PowerPC 970.

System level virtualization provides several advantages like load balancing, fault tolerance etc. However, Xen is still new, work on checkpoint/restart of a VM in Xen is in progress. Design of Xen is targeted at hosting up to 100 VM instances simultaneously on a mordern server. Distributed applications running on Xen still don't have support for distributed checkpointing and recovery of domains. Thus, the aim of project is to add Distributed Checkpoint and Recovery Mechanism to Xen. Exact details will follow in next section.

1.2 Problem Definition

In distributed system, as the size of the system grows so does the probability that some component will fail. Recovering from such failures is notoriously difficult and is important in design and development of reliable systems, applications and protocols. Checkpointing and rollback recovery is a widely used scheme for dealing with failures at the application level. Extension of these distributed checkpoinitng protocols from application level to the virtual operating systems in XEN framework is not implemented so far. We aim to provide users/applications with a checkpointing library along with few implemented basic checkpointing algorithms, which would enable them to checkpoint a group of VMs that are communicating among themselves.

1.3 Contributions

In this work, I have successfully created the following:

1. Checkpointing and Recovery Library
2. A Synchronous Checkpointing & Recovery Algorithm using the library
3. A Asynchronous Checkpointing & Recovery Algorithm using the library
4. Proper Documentation

1.4 Thesis Outline

The rest of the thesis is structured as follows. Section 2 describes the related work done in XEN and Checkpointing and Recovery Algorithms. Section 3 describes the Checkpointing and Recovery Library and its development. Section 4 provides future directions and conclusions. This is followed by bibliography.

Chapter 2

Background & Related Work

This chapter provides a background by looking into some of the central concepts of virtualization and checkpointing. First the concept of VMs is presented describing different approaches to virtualization and types of VMs. Next Section then gives a brief overview of Xen from an architectural point of view. Then we look at the work on checkpointing and recovery and final section is about implementation of checkpointing in Xen.

2.1 Virtual Machines

An abstraction of a common computer is defined as the three layers: hardware, operating system and application software. Each of these layers may be emulated having other layers or even computer architectures as platforms. The concept of such emulation is called a VM. A VM is an emulation of lower layers of a computer abstraction on behalf of higher layers to a certain extent, the higher layer mechanisms are given the illusion that they are running on a certain lower layer mechanism, yet they are actually running on a virtual instance of this mechanism.

As a VM can be implemented on any layer in this abstraction while providing another virtual layer, many combinations exist in possible VM solutions; however, the most common are software based and provide higher level runtime environments or virtualization of hardware or operating systems. Higher level runtime environments provide an abstraction on which programs compiled for the particular VM may run. A virtualization of operating system services may be performed between the layers of the operating system and the higher level software, giving the higher level software the necessary execution environment to perform its tasks. Virtualization of hardware usually is performed running as an application above the operating system; thereby a virtual instance of arbitrary hardware architecture lies on top of the actual hardware, on which an operating system may run.

Hardware level VMs have many applications. Some allow virtualization of architectures so that programs designed for a particular architecture may be executed in a VM on a different architecture; some allow architectures to be developed virtually, so that these may be tested and evaluated, and software may be developed for them, before realization. Moreover, most hardware level VMs that are implemented in software, allow several virtual instances of architecture to run on a single architecture, providing a platform for running multiple operating systems on a single computer.

The execution of a virtual machine (VM) implies that one or more virtual systems are running concurrently on top of the same hardware, each having its own view of available resources. The level in the software hierarchy where the virtualization occurs influences the transparency and performance overhead. The system-level virtualization incorporates a management facility called a Hypervisor, which oversees VMs on a host machine.

There are different approaches to providing a virtual architecture.

2.1.1 Full Virtualization

Virtual machines emulate some real or fictional hardware, which in turn requires real resources from the host (the machine running the VMs). This approach, used by most system emulators, allows the emulator to run an arbitrary guest operating system without modifications because guest OS is not aware that it is not running on real hardware. The main issue with this approach is that some CPU instructions require additional privileges and may not be executed in user space thus requiring a virtual machines monitor (VMM) to analyze executed code and make it safe on-the-fly. Hardware emulation approach is used by VMware products, QEMU, Parallels and Microsoft Virtual Server.

2.1.2 Paravirtualization

This technique also requires a VMM, but most of its work is performed in the guest OS code, which in turn is modified to support this VMM and avoid unnecessary use of privileged instructions. The paravirtualization technique also enables running different OSes on a single server, but requires them to be ported, i.e. they should "know" they are running under the hypervisor. The paravirtualization approach is used by products such as Xen and UML.

2.1.3 Virtualization on the OS level

Most applications running on a server can easily share a machine with others, if they could be isolated and secured. Further, in most situations, different operating systems are not required on the same server, merely multiple instances of a single operating system. OS-level virtualization systems have been designed to provide the required isolation and security to run multiple applications or copies of the same

OS (but different distributions of the OS) on the same server. OpenVZ, Virtuozzo, Solaris Zones & FreeBSD Jails are examples of OS-level virtualization.

2.2 Xen

A few commercial VMMs exist today, such as VMware and VirtualPC. They provide a true x86 VM platform with performance losses that are small enough to make them feasible in some applications, yet large enough to make them infeasible for high performance purposes.

Xen is a novel VMM which allows multiple commodity operating systems to share conventional hardware in a safe way with minimal performance and functionality loss. The Xen VMM was originally intended to be an integral part of a UK research project, Xenoserver. The Xenoserver project aims to provide a wide-area distributed computing platform on which members of the public can submit code for execution. Later Xen emerged as a separate entity.

Design principles of Xen are[5]:

- Support for unmodified application binaries is essential, or users will not transition to Xen. Hence all architectural features required by existing standard ABIs must be virtualized.
- Supporting full multi-application operating systems is important, as this allows complex server configurations to be virtualized within a single guest OS instance.
- Paravirtualization is necessary to obtain high performance and strong resource isolation on uncooperative machine architectures such as x86.

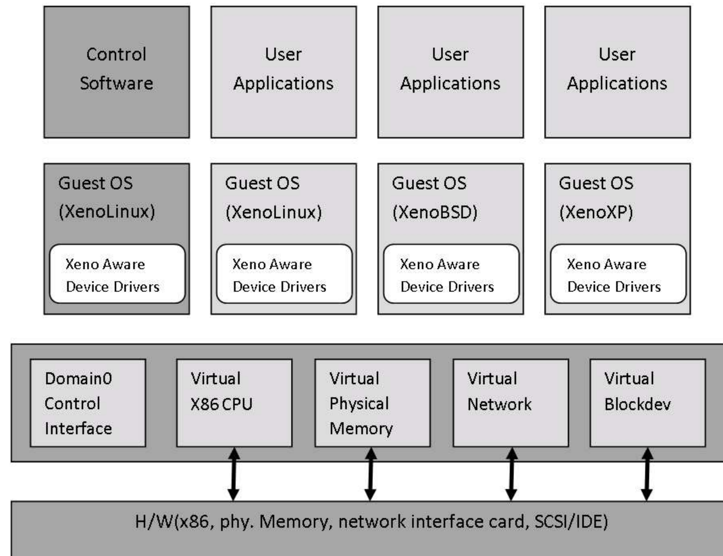

Figure 2.1: Xen Architecture

- Even on cooperative machine architectures, completely hiding the effects of resource virtualization from guest OSes risks both correctness and performance.

A Xen virtual environment consist of several items that work together to deliver the virtualization environment which are as follows:

1. Xen Hypervisor
2. Domain 0
3. Domain Management and Control (Control Software + Domain0 Control Interface)
4. Domain U (Dom U)

The basic organization of these domains is shown in Figure 2.1.

Xen Hypervisor

The Xen Hypervisor is the basic abstraction layer of software that sits directly on the hardware below any operating systems. It is responsible for CPU scheduling and memory partitioning of the varoius virtual machines running on the hardware device. The hypervisor not only abstracts the hardware for the virtual machines but also controls the execution of virtual machines as they share the common processing environment. It has no knowledge of networking, external storage devices, video or any other common I/O functions found on a computing system.

Domain 0

Domain0, a modified Linux kernel, is a unique virtual machine running on the Xen hypervisor that has special rights to access physical I/O resources as well as interact with the other virtual machines running on the system. All Xen virtualization environments require Domain0 to be running before any other virtual machine can be started. Two drivers are included in Domain0 to support network and local disk requests from DomainU or guest domains; the Network Backend Driver and the Block Backend Driver. The Network Backend Driver communicates directly with the local networking hardware to process all virtual machines requests coming from Domain U guests. The Block Backend Driver communicates with the local storage disk to read and write data from the drive based upon Domain U requests.

Domain U

DomainU refers to all paravirtualized virtual machines running on a Xen hypervisor. They run modified Linux operating systems, Solaris, FreeBSD, and other UNIX operating systems. There are also fully virtualized machines like running Windows

etc. which are known as HVM Guests. For our work, we will concentrate only on paravirtualized virtual machines.

The DomainU guest virtual machine is aware that it doesnot have direct access to the hardware and recognizes that other virtual machines are running on the same machine. The DomainU Guest virtual machine is not aware that it is sharing processing time on the hardware and that other virtual machines are present. DomainU contains 2 drivers for network and disk access, Network Frontend Driver and Block Frontend Driver.

Domain Management and Control

A series of Linux daemons are classified as Domain Management and Control by the open source community. These services support the overall management and control of the virtualization environment and exist within the Domain0 virtual machine.

Brief Working

The idea behind Xen is to run guest operating systems not in ring 0, but in a higher and less privileged ring. Running guest OSes in a ring higher than ring 0 is called "*ring de-privileging*". The default Xen installation on x86 runs guest OSes in ring 1, termed Current Privilege Level 1 (or CPL 1) of the processor. It runs a virtual machine monitor (VMM), the "hypervisor", in CPL 0. The applications run in ring 4 without any modification[2].

However, some instructions in IA-32 instruction set are problematic in running in ring 1. These instructions can be problematic in two senses. First, running the instruction in ring 1 may cause General protection exception (GPE), which also may be called a general protection fault (GPF). For example, running HLT immediately

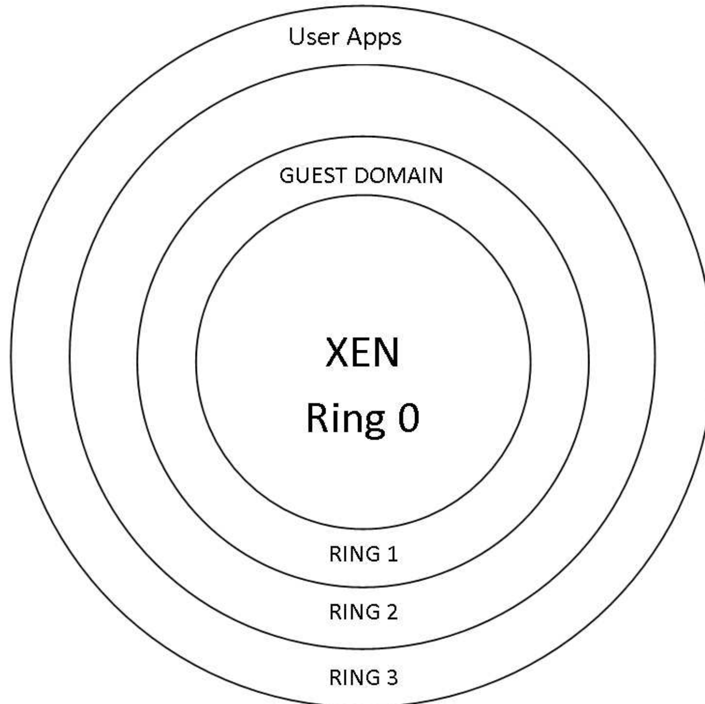

Figure 2.2: Xen runs in ring 0, Guest Domain runs in ring 1, User applications run in ring 4

causes a GPF. Some instructions, such as CLI may cause a GPF if a certain condition is met. That is, a GPF occurs if the CPL is greater than the IOPL of the current program or procedure and, as a result, has less privilege. Another problem occurs with instructions that don't cause GPF but still fail. Many Xen articles use the term "fail silently" to describe such cases. Xen handles these problems via a provision called hypercall. A hypercall is Xen's analog to Linux system call. A system call is an interrupt (0x80) called in order to move from user space (CPL3) to kernel space (CPL0). A hypercall is also an interrupt (0x82). It passes control from ring1, where guest domains are running to ring0, where Xen runs[1] . To provide *safe hardware isolation*, Xen uses Virtual Split Drivers. Domain 0 is the only one which has direct access to hardware devices, and it uses original Linux drivers. But domain 0 has another layer, the backend, which contains netback and blockback virtual drivers[3].

Similarly, the unprivileged domains have access to a frontend layer, which consist of netfront and blockfront virtual drivers. The unprivileged kernel issues I/O

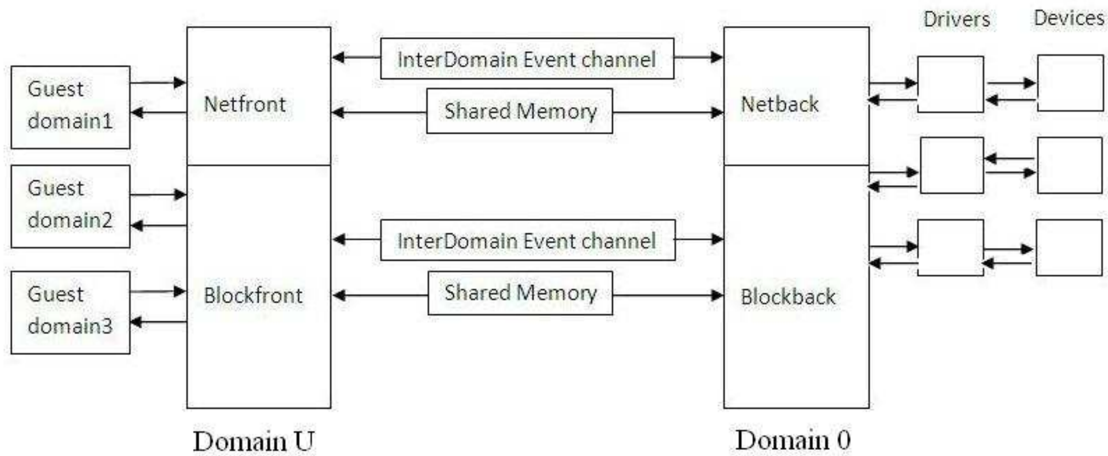

Figure 2.3: The Split Device Driver Model in Xen

requests to the frontend in the same way that I/O requests are sent to ordinary Linux kernel. However, because frontend is only a virtual interface with no access to real hardware, these requests are delegated to the backend. From there they are sent to the real devices. In xen domains use Event Notifications and Grant Tables to communicate between each other.

Event notifications in Xen travel between domains via Event channels. An event in Xen is equivalent to a hardware interrupt. They essentially store one bit of information, the event of interest is signaled by transitioning this bit from 0 to 1. Event notifications can be masked by setting a flag; this is equivalent to disabling interrupts and can be used to ensure atomicity of certain operations in the guest kernel.

Xen's Grant Tables provide a generic mechanism to memory sharing between domains. This shared memory interface underpins the split device drivers for block and network IO. Each domain has its own grant table. This is a data structure that is shared with Xen; it allows the domain to tell Xen, what kind of permissions other domains have on its pages. Entries in grant tables are identified as grant references.

A grant reference is an integer, which indexes into grant table.

Thus these two features provide an asynchronous notification mechanism between domains.

2.3 Checkpointing and Recovery

A system consists of a set of hardware and software components. Failure of a system occurs when the system doesn't perform its services in the manner specified. An *erroneous state* of the system is a state which could lead to a system failure by a sequence of valid state transitions. A *fault* is an anomalous physical condition which may be caused due to design errors, manufacturing problems etc. An *error* is that part of the system state which differs from its intended value [18, 15, 11].

A *system failure* occurs when the processor fails to execute. It is caused by hardware problems or software errors. In case of system failure, the processor is stopped and restarted in correct state. When the nature of the errors and damage caused by faults can be accurately and completely assessed then it is possible to remove the errors and enabling the system to move forward. This technique is called *forward error recovery.* On the other hand, if it is not possible to foresee the errors or remove the errors in the system's state, then the system's state can be restored to a previous error-free system state. This technique is known as *backward error recovery*[18] .

Backward error recovery can be achieved by 2 methodologies, operation-based and state-based[11] . In *operation based approach*, we store the state of the system in sufficient detail so that a previous state can be restored by reversing all the changes made to the state. In *state based approach*, complete state is saved when a recovery point is established and recovering a system involves reinstating its saved state and

resuming the execution from that state[18, 6] . The process of saving state is referred to as checkpointing or taking a *checkpoint*. The process of restoring a process to a previous state is called as *rolling back* of the process.

Distributed Systems have become popular because of several advantages they have over centralized ones. They provide enhanced performance and increased availability. One way of realizing enhanced performance is through concurrent execution of many processes, which cooperate in performing a task. An important requirement of distributed systems is the ability to tolerate failures. The probability that some component in a distributed system will fail increases with the increase in the size of the system. Checkpointing in distributed systems is much more complicated. Each processor saves its state at the local stable storage. These recovery points are called *local checkpoints*. All the local checkpoints, one from each site, collectively form a *global checkpoint*. A global checkpoint is called a *consistent* set of checkpoints, if every message recorded as received in a local checkpoint is also recorded as sent in another local checkpoint that constitute the same global checkpoint.

There are 2 approaches towards distributed checkpointing, synchronous checkpointing and asynchronous checkpointing [21, 19, 12, 10, 9, 6]. The *synchronous checkpointing* is to ensure that all processes keep local checkpoint in stable storage and coordinate their local checkpoint action such that global checkpoint is guaranteed to be consistent. When a failure occurs, processes roll back and restart from their most recent checkpoints. While crash recovery is easy and simple in this case, additional messages are generated for each checkpoint, and synchronization delays are introduced during normal operations. If there are no failures, then above approach places an unnecessary burden on the system in form of additional messages and message delays. Similarly, when a processor rolls back and restarts after a failure, a number of additional proce‘sses are forced to roll back with it. The processes indeed roll back to a consistent state, but not necessary to maximum consistent

state. In the *asynchronous approach*, each processor takes local checkpoints independently and a consistent global state is constructed using these checkpoints during recovery. To help in crash recovery and minimize the amount of computation done during a rollback, all incoming messages are logged (stored on a stable storage) at each processor. 2 approaches are available for message logging namely, pessimistic message logging and optimistic message logging. In pessimistic logging, each process has to block wait for the processor state of a non deterministic event before the effects of that event can be seen by outside world or other processes. It simplifies recovery but hurts failure free operation. In optimistic logging, the process doesn't block, and determinants are spooled to stable storage asynchronously. Optimistic logging reduces failure-free overhead, but complicates recovery.

In message passing system, recovery via rolling back is complicated because transmission and reception of messages introduce interprocess dependencies during failure free operation. Upon a failure of one or more processes of the system, these dependencies may force some of the processes that didn't fail to rollback, creating what is known as *rollback propagation*. Under some senarios, rollback propagation may extend back to the initial state of the computation, losing all work performed before a failure. This situation is known as *domino effect*.

2.4 Checkpointing and Recovery in Xen

The Xen Hypervisor provides mechanisms that allow users to take checkpoints of the VMs. The hypervisor is responsible for the checkpoint and restart of a virtual machine. However, the hypervisor works in concert with the host OS to access resources to actually carry out the process, i.e., writing checkpoint to disk, send/receive data on network. Therefore the checkpoint/restart mechanism for virtual machines is composed of two parts[20]:

1. Hypervisor checkpoint mechanism
2. The checkpoint manager and resource manager that run the host OS

The implemented hypervisor checkpoint mechanism consists of 2 parts[7]:

Checkpoint Mechanism at Guest/DomU

1. Hypervisor asks the guest for help by writing a suspend message to a location in xenstore on which the guest has a watch.
2. Guest disconnects itself from devices
3. Guest unplugs all starting from CPU
4. Interrupts are disabled
5. Page tables of all the processes are pinned into RAM
6. Prepares a suspend record ? the Xen_start_info structure, with the address of store and console pages converted to PFN, so that restore can rewrite them on restore
7. Makes a suspend hypercall that doesn't return

Checkpoint Mechanism at Domain 0

1. Serialize guest configuration
2. Wait for Xen to announce that the domain has suspended
3. Map guest memory in batches and write it out with a header listing the PFNs in the batch
4. Write out VCPU state for each VCPU, with MFN to PFN fix-ups

Along with the checkpoint mechanism at the hypervisor, the implementation checkpoint manager, resource manager is proposed[20]. In later section we discuss briefly about the disk checkpointing.

Checkpoint Manager

The checkpoint coming from the Hypervisor is a raw checkpoint saving the entire VM image. Before storing a checkpoint, it may be interesting to modify this checkpoint. The checkpoint manager is responsible for preparing the checkpoint for storage, to include modification like compression, transfer to remote place, etc.

Resource Manager

Once a checkpoint is ready for storage, the system has to access a hardware resource. The Resource Manager (RM) is responsible for these aspects of the system. The storage may be local (e.g., local disk, local memory) or remote (e.g., remote disk, remote memory) to the VM that is being checkpointed. CM takes care of managing the checkpoints and resource manager abstract the storage method for the CM

Resource Manager is composed of multiple components, each of them being dedicated to a specific resource access (for instance local vs. remote, memory vs. disks). Whenever a checkpoint is received, in order to identify the correct component that can store the checkpoint, the RM sends a request to all the components. If a checkpoint's characteristics match component requirements, the component saves the checkpoint. This enables dynamic management of resources since RM components may be activated/ deactivated according to the resource availability

Disk Checkpointing

Disk checkpointing is yet not implemented in Xen, however, a solution with stackable file system (UnionFS) is proposed.

Once the VM's disk and memory state have been recorded a full rollback mechanism is possible without potential for inconsistency during checkpoint/restart.

Chapter 3

Distributed Checkpoint and Recovery of VMs in Xen

Running large distributed applications on VMs make its very important that the states of these VMs can be restored to most recent state possible in the event of failure. For this purpose, we have built up a distributed CnR library to provide users/applications to write their own checkpoint and recovery algorithm at system level. Before we go on to the library, we look at the packet transmission and reception strategy inside Xen.

3.1 Architecture of Xen Network Split Device Driver

The network Xen driver is a split driver i.e. it has a portion in a privileged domain handling the physical device called backend and a frontend in the unprivileged domain acting as a proxy upon the backend. The backend and frontend communicate using shared event channels and ring buffers, in an architecture known as the Xenbus.

From now on, netfront refers to the frontend interface, netback refers to the backend interface.

The network interface uses two "descriptor rings", one for transmit, the other for receive. Each descriptor identifies a block of contigous machine memory allocated to the domain. The transmit ring carries packets to transmit from the guest to the backend domain. The return path of the transmit ring carries messages indicating that the contents have been physically transmitted and the backend no longer requires the associated pages of memory.

To receive packets, the guest places descriptors of unused pages on the receive ring. The backend will return received packets by exchanging these pages in the domain's memory with new pages containing the received data, and passing back descriptors regarding the new packets on the ring. This zero-copy approach allows the backend to maintain a pool of free pages to receive packets into, and then deliver them to appropriate domains after examining their headers. If a domain doesn't keep its receive ring stocked with empty buffers then packets destined to it may be dropped. This provides some defence against receive livelock problems because an overloaded domain will cease to receive further data. Similarly, on transmit path, it provides the application with feedback on the rate at which packets are able to leave the system.

Flow control on rings is achieved by including a pair of producer indexes on the shared ring page. Each side will maintain a private consumer index indicating the next outstanding message. In this manner, the domains cooperate to divide the ring into two message lists, one in each direction. Notification is decoupled from the immediate placement of new messages on the ring; the event channel will be used to generate the notification when either a certain number of outstanding messages are queued, or a specified number of nanoseconds have elapsed since the oldest message was placed on the ring.

How actually this process takes place is covered in Section and Section .

3.1.1 Transmission of packets

Network Stack sends the socket buffer to the device driver which calls hard_start_xmit function. In case of Xen it is network_start_xmit. In this function, frontend calculates the number of fragments corresponding to the socket. It then acquires a transmission lock. Transmission lock provides mutual exclusion for sending the packet to the backend domain via request descriptors. Frontend then request for a slot in transmit ring and enqueues the request to the backend to transmit the socket buffer. Check for Generic Segmentation Offloading takes place and if TCP segmentation offloading is enabled then another request descriptor is queued in trasmit ring to convey details about the same. If a socket buffer has fragments then frontend sends them via more request descriptors. Finally, it notifies the backend via irq on event channel. Frontend then checks whether there are any pending responses on the transmit ring. Finally, it releases the transmission lock and update the network statistics about number of bytes and packets transmitted.

The backend on receiving the irq adds the network interface to the schedule list and schedules a tasklet to handle the requests from the frontend. The net_tx_action tasklet picks up the network interface (netif) next in schedule and checks for requests in the transmit ring for that interface. The credits for scheduling the netif are updated. Check for GSO is made followed by creating a local copy of the ring request descriptors for the fragments. Now backend starts reconstructing the socket buffer. Metadata for constructing the socket buffer has arrived via request descriptors and the data is received using pages shared via grant table mechanism. Then socket buffer is queued in a transmission queue. Backend continues in this loop till no interface requests scheduling or its buffers are filled with the maximum requests it can handle. Now it dequeues the socket buffer from transmission queue, performs

check for appropriate permission for grant references. It than copies the header of the packet to socket buffer and updates some more fields and finally trasferring the socket buffer to upper layers.

3.1.2 Reception of Packets

The socket buffer arrives at the backend in netif_be_start_xmit function. The socket buffer is then enqueued in receive queue and a tasklet is scheduled for its further processing. This tasklet calls net_rx_action function. In this function, sockets are dequeued from receive queue. Then they are mapped to shared pages whose information is stored in the request descriptors in receive ring using grant table mechanism. After this they are queued in another receive queue, rxq. Once we have no more socket buffers to process, we dequeue packets from the rxq and responses are enqueued inside the receive ring. Finally a notification is sent to the frontend over event channel via irq.

In the frontend, the irq is handled by netif_int function which schedules device for polling to handle reception of packets on finding unconsumed responses inside the receive ring. Inside the netif_poll function, 3 types of queues exist. They are tmpq (temporary queue), rxq (receive queue) and errq (error queue). First the socket buffers are read by analyzing the response descriptors and then they are enqueued in tmpq. In case of error, they are enqueued in errq. Then socket is dequeued from the tmpq and checked for GSO optimization. Then its fragments are attached to it by reading response descriptors corresponding to them. Now we update fields of the socket buffer and enqueue it in rxq. All the socket buffers in errq are now freed as we don't need to do anything with them anymore. Once we have successfully received the socket buffers and queued them in rxq, we need to send them to the higher network layers. for further processing. Therefore, we dequeue the socket buffers, find out the which protocol handler they need to be transmitted to and on basis of

that processing continues further. After sending the socket buffer to higher layers we allocate more shared pages for use by the backend and send their identification via requests in receive ring.

3.2 Synchronous Checkpoint and Recovery

Once, we have seen how the packet gets transferred from domainU to domain0, we can now now modify it to get a control over the packets transmission and reception at system level.

As, we have discussed before, a synchronous checkpointing algorithm coordinates among the participants to take a consistent global checkpoint. We now describe its implementation on VMs instead of processes in following subsections.

3.2.1 System Model

The system is assumed to consist of various virtual machines in Xen Framework. All virtual machines and domain0 of all machines have a secondary storage system. The secondary storage system is assumed to be stable storage i.e. it doesn't lose information in the event of system failure. No halting failures occur in the system. The virtual machines communicate by exchanging messages via communication channels. Channels are FIFO in nature. End to end protocols are assumed to cope with message loss due to communication failure. Communication failures do not partition the network. There is no shared memory or clock between all VMs.

The checkpointing algorithm takes two kinds of checkpoints on stable storage, permanent and tentative. A permanent checkpoint is a local checkpoint at a VM and is part of a consistent global checkpoint. A tentative checkpoint is a temporary

checkpoint that is made a permanent checkpoint on the successful termination of the checkpoint algorithm. Process rollback only to their permanent checkpoint.

3.2.2 Checkpoint Algorithm

Checkpoint algorithm assumes single initiator, as opposed to multiple initiators concurrently invoking the algorithm to take checkpoints. The algorithm is modified version of synchronous checkpointing algorithm proposed by Koo and Touegaccording to architecture of Xen. The algorithm has 3 phases.

1. *First phase:*The checkpoint initiator which is domain0 sends request to all virtual machines to send information about their domain0. Each Virtual Machine then contacts corresponding domain0 to check whether it is ready to take a checkpoint as hypervisor in domain 0 can only take the checkpoint. If domain-0s are ready then they send confirmation to the guest domains who want to take checkpoints. The guest domains then send back the domain-0 information to the initiator.

2. *Second phase* :The initiator then requests all the domain0s corresponding to virtual machines to take tentative checkpoints. Each domain informs initiator whether it succeeded in taking a tentative checkpoint. If domain0 fails to take a checkpoint, it replies "no" which could be due to several reasons, depending upon the underlying virtual machine. If initiator learns that all the processes have successfully taken tentative checkpoints, initiator decides that all tentative checkpoints should be made permanent; otherwise initiator decides that all the tentative checkpoints should be discarded.

3. *Third phase*: The initiator informs all the domain0s of the decision it reached at the end of the first phase. A domain-0 on receiving message from initia-

tor, will act accordingly. Therefore either checkpoint is taken for all virtual machines or for no virtual machine

The algorithm requires that every process, once it has taken a tentative checkpoint, not send messages related to underlying computation until it is informed of initiator's decision.

3.2.3 Recovery Algorithm

Recovery in case of synchronous checkpointing algorithm is easy as each set of permanent local checkpoints combine together to form a consistent global checkpoint. The algorithm has 2 phases:

1. The failed domain sends message to all the other domains about its failure and initiate recovery algorithm.
2. All domains request their corresponding domain0 to shutdown them and start from the last local permanent checkpoint.

3.2.4 Proof of Correctness

A set of permanent checkpoints taken by checkpoint algorithm is consistent because:

1. Either all or none of the VMs is checkpointed
2. A set of checkpoints will be inconsistent if there is a record of a message received but not of event sending it. This will not happen as no VM sends message after a tentative checkpoint of that VM is taken by corresponding hypervisor until the receipt of the initiator's decision, by which time all domains would be checkpointed by their corresponding hypervisors.

The algorithm for the recovery actually recovers the distributed application running inside the VMs as we start the distributed system of VMs from its last consistent global checkpoints, which in turn ensures the consistent global checkpoint for the application.

3.2.5 Complexity

In this section, we find out the message complexity of each of checkpointing and recovery algorithm. The first phase of checkpointing algorithm takes $4N$ in first phase, $2N$ in second phase and N in the third phase where N is the number of domains participating in the checkpointing process. Hence, the total message complexity in worst case is $7N$. In case of recovery, if a domain fails then, another $N-1$ messages are sent for informing other domains to recover from the failure.

3.2.6 Plan for Testing

To get an accurate and comprehensive view of working of synchronous checkpointing algorithm implemented above, rigorous testing of algorithm is essential. In general, testing of distributed applications is difficult because of their non-reproducibility of events, complex timing of events, and complex states. In this section, we discuss our testing plan to check the implementation of synchronous checkpointing algorithm.

Our plan is as follows:

1. **Correctness of Implementation**

   The correctness of implementation is checked by running a distributed application on various VMs, checkpointing them, making a VM to fail and then recovering back the application via the consistent set of checkpoints. For our testing, we have considered two types of applications:

(a) a simple message passing(client-server) application coded using POSIX sockets.

(b) a distributed application for example where computation may spread over various VMs.

In both the cases, we would take a series of permanent checkpoints over suitable time intervals and show that we save system resources by restoring the execution of algorithm from intermediate consistent state rather than starting the applications from initial state.

2. **Fault Tolerance**

   We need to check the ability of algorithm to perform under failures like communication failure, message delays in network. We implemented our algorithm over TCP so that tramsmission and reception errors along with communication failures are tackled by the TCP layer itself. Message Delays are produced by introducing timeout at every send and receive of message by using timeval data structure and setsockopt system call provided under Linux APIs. Arrival of late messages is tackled by keeping data stuctures that change with timeout.

3. **Privileges**

   Checkpoint application need *root privileges* at every host. These privileges were checked by using geteguid system call on Linux. The effective user id of root is 0.

4. **Other Tests**

   Size of each checkpoint file is large which is approximately equal to size of the physical memory we allocate to each virtual machine. Check to ensure that enough disk space is available to store the checkpoint.

3.3 Asynchronous Checkpointing and Recovery

Synchronous checkpointing simplifies recovery, but it has disadvantages like additional message exchange, synchronization delays and unnecessary overhead in cases of no failure. Several asynchronous distributed checkpointing algorithms have been proposed. We choose an optimistic asynchronous distributed checkpionting algorithm given by Juang et. al [?]. We choose this algoirthm because of 2 reasons, *firstly*, we don't need to append any information to the messages so that all the distributed applications written without checkpoint and recovery support can benefit from this algorithm as all the logging, as we will see, can be offsetted to the kernel. *Secondly*, we don't need to store processor state (in our case state of VM) for each message.

The Algorithm assumes two types of log storage are available for logging in the system, namely, volatile log and stable log. Properties of the volatile log are

1. Accessing the volatile log takes less time than stable log.
2. Contents of volatile log are periodically flushed to stable storage and cleared.

The important point to note here is that checkpoint of a guest domain is taken by Domain0 in stable storage because of its large size. Hence, processor state which in this case is the checkpoint file can't be present in volatile storage of either the guest domain or domain0. Also, taking checkpoint after every event is not possible, we take checkpoint at regular intervals in the guest domain and maintain the logs in kernel space of guest domain (which are later flushed to stable storage) while the checkpoint file is present in stable storage.

3.3.1 System Model

The system is assumed to consist of various virtual machines in Xen Framework. All virtual machines and domain0 of all machines have a secondary storage system. The secondary storage system is assumed to be stable storage i.e. it doesn't lose information n the event of system failure. No halting failures occur in the system. The virtual machines communicate by exchanging messages via communication channels. Channels are FIFO in nature. End to end protocols are assumed to cope with message loss due to communication failure. Communication failures do not partition the network. The message delay is arbitrary, but finite. There is no shared memory or clock between all VMs.

3.3.2 Checkpoint Algorithm

In our implementation, we record a tuple (s_j, r_j) for every participant domain in volatile storage where s_j represent number of messages sent to j^{th} domain and r_j represents the number of messages received from j_{th} domain. When we take a checkpoint, dom0 creates a unique checkpoint id and sends it to the corresponding guest domain. The guest domain stores this unique id along with current stats of (s_j, r_j) tuples in stable storage.

Since, the checkpoint procedure is asynchronous, the guest domain (with the help of domain0) takes checkpoints at regular intervals without communicating with the other participant domains.

3.3.3 Recovery Algorithm

Once a domain crashes, it is brought alive by the domain0. It then sends message to all the other domains to initiate recovery algorithm.The recovery algorithm stops

recording (s_j, r_j) as it will be updated after recovery anyways. The checkpointing algorithm stops taking checkpoints.

Recovery Algorithm uses following datastructures:
$RCVD_{i \leftarrow j}(CkPt_i)$ represents the number of messages received by domain i from domain j, per the information stored in the checkpoint $CkPt_i$.
$SENT_{i \leftarrow j}(CkPt_i)$ represents the number of messages sent by domain i to domain j, per the information stored in the $CkPt_i$

Recovery Algorithm has 2 phases.

1. In **first phase**, the domains stops the logging of messages and finds out the recovery point. The pseudocode for this phase is given in Algorithm 1. This algorithm runs at each domain i. In this algorithm, we first decide the last recovery point for each domain. For the crashed domain it is the last checkpoint stored on the stable storage, for others, it is the current state. Now we send rollback messages to the all the domains which consist of number of messages sent by i during the execution of some distributed application. Each domain waits for these messages to arrive from other domains and store them into a queue. Then we pick up messages at the head of queue and check if the number of messages received by i from other domain j is greater than the value stored in rollback message. If this is the case, this implies presence of orphan messages, hence we have to rollback to checkpoint before the present checkpoint. This is repeated for times equal to number of domains that are participating in recovery procedure.

2. In **second phase**, the domains ask their domain0 to start them from the recovery point calculated in phase1. The domain0 then restarts the guest domain at the requested state stored as a checkpoint.

Algorithm 1 Rollback Recovery Algorithm

```
if d_i is the crashed domain then
  REC_i ← the last checkpoint in the stable storage
else
  REC_i ← the latest event that took place in d_i
end if
for k ← 1 to |V| do
  for  each domain d_i do
    compute SENT_{i→j}(REC_i)
    send a rollback SENT_{i→j}(REC_i) message to d_j
  end for
  repeat
    wait for a rollback(c) message
    m ← rollback(c) message received
    put m into processing queue
  until a rollback message from each domain is received
  while processing queue ≠ Φ do
    let m = rollback(c)be a message in processing queue
    delete m from rollback processing queue
    compute the RECEIVED_{i←j}(REC_i) if m came from p_j
    if RECEIVED_{i←j}(REC_i) > c then
      find the latest checkpoint CkPt such that
      RECEIVED_{i←j}(e) <= c
      REC_i ← CkPt
    end if
  end while
end for
```

3.3.4 Complexity

In worst case, the Phase 1 of algorithm takes $\frac{n^2(n-1)}{2}$ messages to find the recovery points for all the domains while the phase 2 takes n messages to start domains from their recovery points where n is the number of domains participating in the distributed application.

3.3.5 Proof of Correctness

Lemma 3.3.1. *At the end of each iteration of the recovery procedure in phase 1, at least one domain will rollback to its final recovery point unless the current recovery points are consistent.*

Proof. to ensure the consistency of the domain states, we let domains to rollback. During the first iteration, the processor that rolls back finds the correct recovery point. At subsequent iterations, it is impossible with respect to state of any of other domain. □

Theorem 3.3.1. *At the end of the recovery procedure in phase 1, all the domains roll back to the optimum consistent recovery points.*

Proof. A domain rolls back to a state only because the state immediately following that was started by message m that transitively depends on an unlogged state of a failed domain. Thus, the rollback points for all domains are optimum. From Lemma 3.3.1, it is clear that the domain states are consistent. □

3.3.6 Plan for Testing

To get an accurate and comprehensive view of working of asynchronous checkpointing and recovery algorithm implemented above, rigorous testing of algorithm is essential. In this section, we discuss our testing plan to check the implementation of asynchronous checkpointing algorithm.

Our plan is as follows:

1. **Correctness of Implementation**

   The correctness of implementation is checked by running a distributed application on various VMs, checkpointing them, making a VM to fail and then recovering back the application via the consistent set of checkpoints. For our testing, we have considered two types of applications:

   (a) a simple message passing(client-server) application coded using POSIX sockets.

   (b) a distributed application for example where computation may spread over various VMs.

   In both the cases, we would take a series of permanent checkpoints over suitable time intervals and show that we save system resources by restoring the execution of algorithm from intermediate consistent state rather than starting the applications from initial state.

2. **Fault Tolerance**

   We need to check the ability of algorithm to perform under failures like communication failure, message delays in network. We implemented our algorithm over TCP so that tramsmission and reception errors along with communication failures are tackled by the TCP layer itself. Message Delays are produced by introducing timeout at every send and receive of message by using timeval

data structure and setsockopt system call provided under Linux APIs. Arrival of late messages is tackled by keeping data stuctures that change with timeout.

3. **Privileges**

   Checkpoint application need *root privileges* at every host. These privileges were checked by using geteguid system call on Linux. The effective user id of root is 0.

4. **Other Tests**

   Size of each checkpoint file is large which is approximately equal to size of the physical memory we allocate to each virtual machine. Check to ensure that enough disk space is available to store the checkpoint.

3.4 Implementation of Checkpoint and Recovery Library in Xen

The implementation of checkpointing and recovery on the system level requires support from the operating system in controlling the network drivers. In case of Xen since each domain has modified OS to support paravirtualization. Hence, we will have to change the modified OS. In our case we do it for modified Linux Kernel. The control over these changes are made available to the users/applications via system call.

3.4.1 Functions

The functions that have been implemented as system calls are as follows:

1. initIpArray

This function is used to tell the kernel the ip address of the domains participating in the distributed task. It is assumed that the ip remains constant throughout.

2. deinitIpArray

   Once, we are done with everything, we need to free the kernel space occupied by the ip address of various participating domains. DeinitIpArray serves the same purpose

3. blockips

   Once, we take a checkpoint for eg. in synchronous checkpointing, we need to ensure that no message exchange takes place between participating domains. Since applications are not aware of the checkpointing process, we implement blockips function at the system level assuming that the message loss will be taken care by lower network layers like TCP.

4. unblockips

   The blocked communication with the domains is started with this function

5. initMsgStatsSpace

   This function allocates space in the kernel for storing the logs and stats of the message exchange that takes place between the domains. These logs are required in asynchronous checkpointing and recovery process.

6. takeMsgStats

   TakeMsgStats function tells kernel to start the logging the details of the messages sent and received from the local domain to other participating domains

7. storeMsgStats

   This function allows user to stop the logging and get the current logs in user data structures, so that user can store them in stable storage.

8. resetMsgStats

   ResetMsgStats function is used to reset the values stored in the logs to some desired value. This function is needed when we recover from the failure and the domains restart from the last recovery point decided during the recovery algorithm.

9. removeMsgStats

   Once, we are done with the messages, we need to remove the space allocated in the kernel for taking the stats. We use removeMsgStats to do it for us.

Apart from these functions we have fucntions to initiate the synchronous and asynchronous checkpointing and recovery procedures described in previous sections. But these are not implemented as the library functions, instead they are included in the domain control interface in domain0.

3.4.2 Plan for Testing

Testing of system calls is not similar to the testing any function call. It is different because, these system calls are integrated with the kernel and mistakes often give rise to the OS and filesystem crash. The testing of these system calls is not possible via normal programs either. Only way to hack kernel is via Module Programming. Hence, kernel modules are written for each test case for system call. Testing is done manually instead of automatic, as the size of library is not big.

3.5 Challenges Faced

While implementing distributed checkpointing in Xen, we faced following challenges:

1. *Communication between guest and parent*: Communication between guest and parent can be done via TCP/IP sockets or by inbuilt Event channel Mechanism. If we do it via event channel mechanism and grant table mechanism (using shared memory between dom0 and VM) then we need to develop a fake split driver that would enable both of the features for us. Presently, POSIX sockets are used to communicate between guest and parent.

2. The applications presently running inside the VM are unaware of the checkpoint process of VM. It is intuitive that an application can benefit itself from the distributed checkpoint of Virtual machines.

3. Testing of Kernel Functions is not easy. Mistakes lead to OS crashes and are difficult to debug. Crashes often corrupt the file system as well by corrupting Inode tables. This often requires building up the system from scratch. Proper backups are taken to handle such situations.

4. Since Xen is an opensource software, various optimizations and changes take place. The implementation was started on Xen 3.0.x and now Xen 3.2.x is released. Implementation of our library was updated to current version to utilize the benefits of optimizations and bug-fixes.

Chapter 4

Conclusion and Future Work

In this final chapter, we recapitulate the road we travelled to get here. We hope that it helps reader to understand the rationale behind our work in thesis.

4.1 Roads Travelled

Starting with the problem of inability of VMs to recover a distributed computation inspite of their increasing usage in Distributed Applications, we analyzed the reason for the same. Primary reason behind this was there was no way to implement distributed checkpointing of Operating System till now. With the introduction of VMs, taking checkpoint was offloaded to the a special domain called Domain0, while the original domain can continue the distributed application.

Checkpoint and Recovery for processes is a well-researched topic and there are many approaches available to checkpoint a group of processes participating in a distributed computation and a corresponding approach to recover using those checkpoints. For, our work, we picked up synchronous and asynchronous distributed checkpointing and recovery approach to checkpoint a groop of VMs. Xen is one of

the popular VMM and is opensource. Now, our task was first to understand the working of Xen, how OS are modified to VMs, mechanism to bring up VMs, networking using Xen, etc. Then we looked on how checkpointing is implemented in Xen.

Then we implemented the synchronous and asynchronous checkpointing and recovery algorithms for VMs. CnR library was also formed. Now, we need to validate and verify our algorithms and functions, hence, we made proper test plans and conducted tests accordingly. Finally, we give the documentation of the library.

4.2 Contributions

The contributions of this work are as follows:

1. Checkpoint and Recovery Library
2. A synchronous CnR algorithm using library
3. An asynchronous CnR algorithm using library
4. Proper Documentation

4.3 Future Directions

As future work, we can do the following:

1. Garbage Collection

   Garbage collection is the procedure to remove the local checkpoints which don't constitute the most recent consistent global checkpoint. Garbage Collection (GC) is necessary in our systems if there are constraints on storage

space. GC algorithms run as secondary tasks and quite a few algorithms exist. It would be good to have such an algorithm to save storage space.

2. Quasi Synchronous Checkpointing

   One of important variants of Synchronous Checkpointing and Recovery is Quasi Synchronous Checkpointing and Recovery. Adding its implementation would provide more choices to the user.

Bibliography

[1] Xen Interface manual 2.

[2] Xen Interface manual 3.

[3] Xen wiki http://wiki.xensource.com/xenwiki.

[4] JD Bagley, ER Floto, SC Hsieh, and V. Watson. Sharing data and services in a virtual machine system. *Proceedings of the fifth ACM symposium on Operating systems principles*, pages 82–88, 1975.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, 2003.

[6] K.M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[7] B. Cully and A. Warfield. Virtual Machine Checkpointing. *Xen Summit*, 2007.

[8] SW Galley. PDP-10 virtual machines. *Proceedings of the workshop on virtual computer systems table of contents*, pages 30–34, 1973.

[9] D.B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. *J. Algorithms*, 11(3):462–491, 1990.

[10] R. Koo and SAM Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13:23–31, 1987.

[11] PA Lee, T. Anderson, JC Laprie, A. Avizienis, and H. Kopetz. *Fault Tolerance: Principles and Practice.* Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1990.

[12] P.J. Leu and B. Bhargava. Concurrent robust checkpointing and recovery in distributed systems. *Data Engineering, 1988. Proceedings. Fourth International Conference on*, pages 154–163, 1988.

[13] S.E. Madnick and J.J. Donovan. Application and analysis of the virtual machine approach to information system security and isolation. *Proceedings of the workshop on virtual computer systems table of contents*, pages 210–224, 1973.

[14] RA Meyer and LH Seawright. A virtual machine time-sharing system. *IBM Journal of Research and Development*, 9(3):199, 1970.

[15] V.P. Nelson. Fault-tolerant computing: fundamental concepts. *Computer*, 23(7):19–25, 1990.

[16] G.J. Popek and R.P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[17] G.J. Popek and C.S. Kline. The PDP-11 virtual machine architecture: A case study. *Proceedings of the fifth ACM symposium on Operating systems principles*, pages 97–105, 1975.

[18] B. Randell. *Reliable Computing Systems.* Springer-Verlag London, UK, 1978.

[19] Y. Tamir and C.H. Sequin. Error recovery in multicomputers using global checkpoints. *13th International Conference on Parallel Processing*, pages 32–41, 1984.

[20] G. Vallee, T. Naughton, H. Ong, and S.L. Scott. Checkpoint/Restart of Virtual Machines Based on Xen. 2006.

[21] K. Venkatesh, T. Radhakrishnan, and HF Li. Optimal checkpointing and local recording for domino-free rollback recovery. *Information Processing Letters*, 25(5):295–304, 1987.