Distributed Checkpointing of Virtual Machines in Xen Framework

SYNOPSIS 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

Dr. Arobinda Gupta

Department of Computer Science and Engineering
Indian Institute of Technology
Kharagpur
May 2008

Distributed Checkpointing of VMs in Xen Framework

Sankalp Agarwal, 03CS3023

April 30, 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.

1 Introduction

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[19, 16, 12].

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[19].

Backward error recovery can be achieved by 2 methodologies, operation-based and state-based[12]. 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[19, 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[22, 20, 13, 11, 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 processes 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.

Virtualization is widely used technique in which a software layer multiplexes lower-level resources among higher level software programs and systems. Examples of virtualization systems include a vast body of work in the area of operating systems[15, 18, 17, 14, 8, 4], high-level language virtual machines such as those for Java and .Net, and more recently, virtual machine monitors (VMMs). VMM virtualizes entire software stacks including the operating system and application, via a software layer between hardware and the OS of the machine. VMMs offer a wide range of benefits including application and full system isolation, OS based migration, distributed load balancing, OS-level checkpointing and recovery, non-native application execution and support 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 OS virtualization using VMMs and Virtual Machine will refer to instances of virtualized OS.

Virtual Machines give higher performance by cooperating together to complete tasks. They may be running multiple distributed applications. Failure of one Virtual Machine causes failure of all the processes that were running inside the VM. Recovering from such failures require coordination between checkpointing processes running on various virtual machines.

In our work, we aim to give a distributed checkpointing library in Xen, using which user can implement his/her own protocol for distributed checkpoint of VMs in Xen framework. Library would be packaged with synchronous and asynchronous algorithms for direct usage.

The rest of the report is as follows: Section 2 gives brief introduction to virtual machines, Xen VMM and checkpoint implementation in Xen. Section 3 explains the problem statement, our approach and solution to it. The report is concluded in Section 4.

2 Background

This section 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 and types of VMs. This is followed by description of Xen virtual machine monitor from an architectural point of view. Finally the last section is about checkpointing of VM implemented in Xen.

2.1 Virtual Machines

An abstraction of a common computer defined as the three layers: hardware, operating system and higher level 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. The VMs run on the host

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 The XEN Virtual Machine Monitor

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.

The overall system structure is illustrated in Figure 1. Note that a domain is created at boot time which is permitted to use the control interface. This initial domain, termed Domain0, is responsible for hosting the aplication level management software. The control interface provides the ability to create and terimnate other domains and to control their associated scheduling parameters, physical memory allocations and the accesss they are given to the machine's physical disks and network devices. In addition to processor and memory resources, the control interface supports the creaeting and deletion of virtual network interfaces (VIFs) and block devices (VBDs)[5]. These virtual I/O devices have associated access-control information which determines which domains can access them, and with what restrictions.

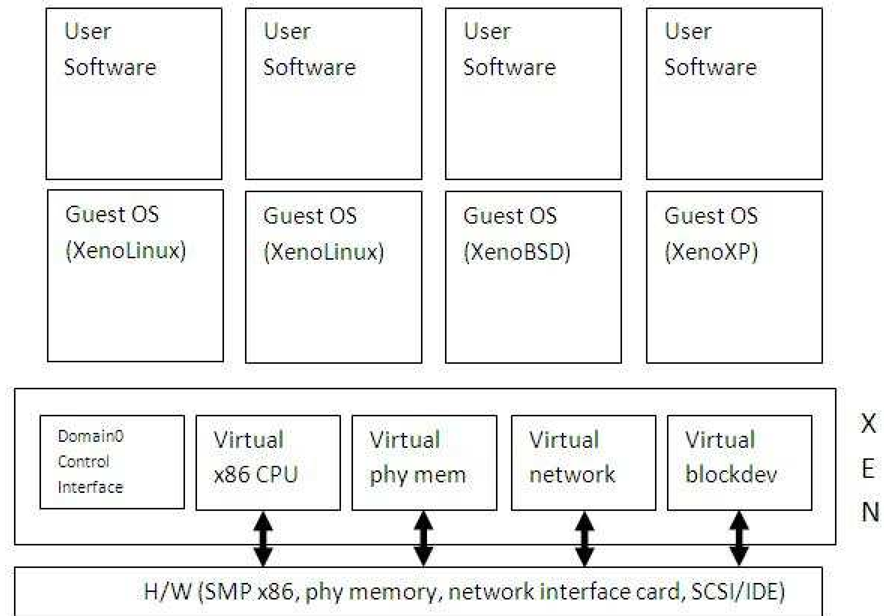

Figure 1: The structure of a machine running Xen hypervisor, hosting a number of different guest operating systems, including domain0 running control interace

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]. 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 domain0 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 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.

Event notifications in Xen travel between domains via Event channels. An event is 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.

2.3 Checkpoint 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

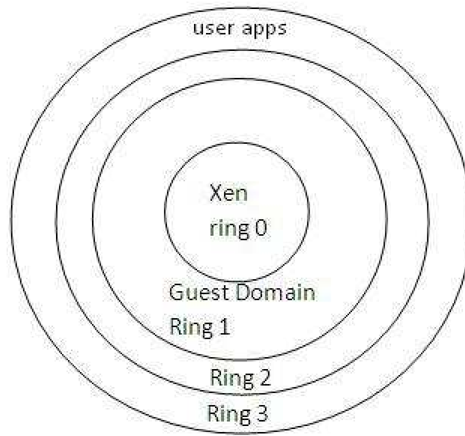

Figure 2: Xen runs in ring 0, Guest Domain runs in ring1, User applications run in ring 3

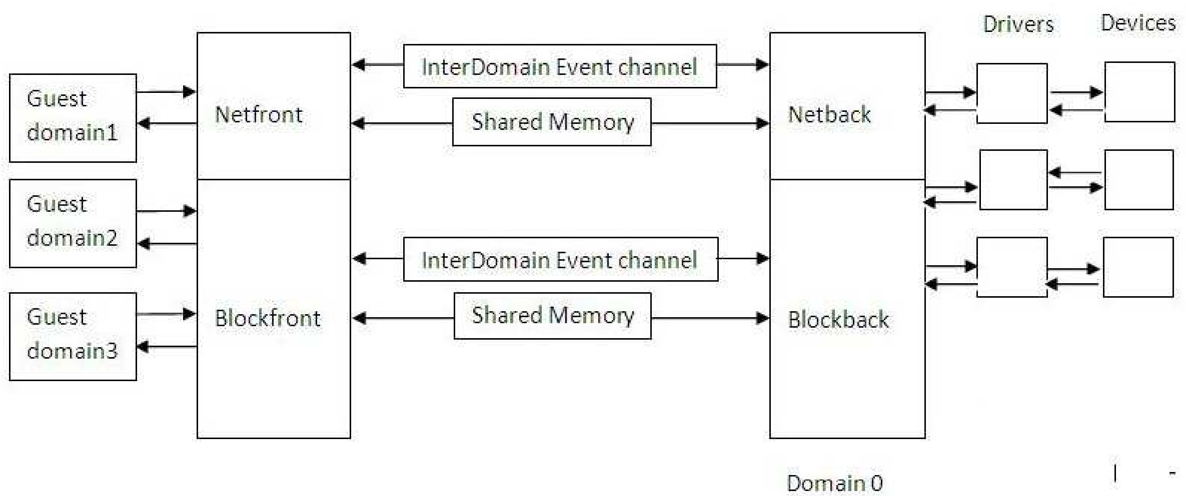

Figure 3: Split device driver structure in Xen

to disk, send/receive data on network. Therefore the checkpoint/restart mechanism for virtual machines is composed of two parts[21]:

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[21]. In section 2.3.3 we discuss briefly about the disk checkpointing.

2.3.1 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 modifications like compression, transfer to remote place, etc.

2.3.2 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

2.3.3 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.

3 Problem definition and approach

Virtual machines provide enhanced performance when they cooperate together to perform tasks. Checkpointing a VM in Xen is on progress. This directly leads us to the path of checkpointing a group of VMs running over various machines. In this work, we aim to provide a distributed checkpointing library that would enable user/application to build up a distributed checkpointing algorithm for a group of Virtual Machines without having to worry about details of Virtual Machine Specifications. We would package 2 checkpointing algorithms (synchronous & asynchronous) along with the library.

The 5-step approach to deliver the library is as follows:

1. Read Xen source code & implementation of VM checkpointing in Xen
2. Implementation of synchronous and asynchronous distributed checkpointing algorithm over a group of VMs
3. Creation of Library
4. Testing of Algorithms and Library
5. Library Documentation

3.1 Synchronous Distributed Checkpoint of VMs in Xen

3.1.1 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. Domains rollback only to their permanent checkpoints.

3.1.2 Checkpoint and Recovery Algorithm

The 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 Toueg[11] according 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 get the information about their domain0. Each Virtual Machine then contacts corresponding domain0 to check whether it is ready to take a checkpoint as hypervisor in domain0 can only take the checkpoint. If domain0s are ready then they send confirmation to the guest domains who want to take checkpoints. The guest domains then send back the domain0 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 domain0 on receiving message from initiator, 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 Asynchronous Distributed Checkpoint of VMs in Xen

Synchronous chekcpoiting 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 [10]. We choose this algoirthm because 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.

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 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.

3.2.2 Checkpoint and Recovery Algorithm

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.

In our implementation, we record a tuple (s_j, r_j) 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.

As in other asynchronous algorithms, the guest domain (with the help of domain0) takes checkpoints at regular intervals without communicating with the other domains involved in distrbuted computation.

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 $CkPt_i$

The recovery algorithm is as follows:

3.3 Plan for Testing

To get an accurate and comprehensive view of working of implemented checkpointing algorithms, rigorous testing 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 as well as asynchronous checkpointing algorithm.

Our plan is as follows:

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
```

1. Correctness of Implementation

   The Correctness of the implemented algorithms is checked by running a distributed application on various VMs, checkpointing it, making it 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 Checkpoint and Recovery Library

The Checkpoint and Recovery Library is developed which provides various functionalities at the kernel level in the form of system calls to the user. Detailed analysis of system calls is covered in the thesis. The system calls can be devided into two broad categories, namely, Communication Control and Logging. Communication Control involves functions like blocking and reviving connection to other VMs at the kernel level. Logging functions aim to help the user collect various statistics about the message exchanges that take place between VMs, for example, getting the count of number of messages sent from native machine to some other VM.

The Library also provides functions to initate the synchronous and asynchronous distributed checkpointing and recovery algorithms.

3.4.1 Testing Plan

Library is tested using Black Box Model of Testing for each system call. System calls run in kernel space, hence, testing them requires a different procedure than normal function calls. Kernel Modules also run in kernel space and are the only way to hack inside the kernel. Hence we develop a kernel module for every test vector and manual testing is done rather than automated testing. Also strace is used to trace the system calls.

3.5 Challenges

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.

4 Conclusion

Virtual machines provide enhanced performance when they cooperate together to perform certain tasks. There are many VMM providers in the market. Out of them, Xen provides an excellent platform for deploying a wide variety of network-centric services. It also provides the facility of checkpointing a virtual operating system (guestOS). Distributed VM checkpointing is not implemented so far by any of the virtualization software including Xen, VMware, etc. We address this problem and give a library along with few ready to use algorithms that user/application can use to either device a distributed checkpointing protocol or directly use the ones provided by us to checkpoint a group of VMs.

References

[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] T.T.Y. Juang and S. Venkatesan. Crash recovery with little overhead. *Distributed Computing Systems, 1991., 11th International Conference on*, pages 454–461, 1991.

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

[12] 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.

[13] 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.

[14] 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.

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

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

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

[18] 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.

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

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

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

[22] 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.