The University of Hong Kong

Faculty of Engineering

Department of Computer Science

COMP7704
Dissertation Title

Building Reliable and Secure Replication Service in Distributed Systems

Submitted in partial fulfillment of the requirements for the admission to the degree of Master of Science in Computer Science

By
Yi Ning
HKU student no. 3035249186

Superviosr: Dr. Heming Cui
Date of submission: 01/12/2016

Abstract

By allowing multiple servers to be consolidated on fewer physical hosts, virtualization technologies are widely used. Nevertheless, consolidation exacerbates the consequence of unexpected host failures. Replication is an attractive way of providing high-availability for virtual machines (VMs). State machine replication (SMR) is an efficient and mainstream approach for replication. It regards the VMs as deterministic state machines that are kept identical by starting them from initial state and ensuring the same requests are applied in the same order.

This dissertation presents an efficient high availability solution named REPKVM for Kernel-based Virtual Machine (KVM). A RDMA-based SMR protocol is used for providing fault-tolerant applications, based on the approach of replicating the execution of the leader application. The critical applications running in the virtual machine can be guaranteed available all the time even in the case of hardware failure. Evaluation on three widely used open-sourced databases (e.g. MySQL and MongoDB) shows that this solution has moderate overhead.

Building Reliable and Secure Replication Service in Distributed Systems

by

YI Ning

A dissertation submitted
In partial fulfilment of the requirement for
the degree of Master of Science
at the University of Hong Kong

December 2016

Declarations

I declare that this thesis represents my own work, expect where due acknowledgment is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institution for a degree, diploma or other qualifications.

Acknowledgements

First and foremost, I would like to take this opportunity to express my deep sense of gratitude to all those people who gave advices and encouragements to me during my study in HKU.

I am very much thankful to my supervisor, **Dr. Heming Cui**, for his continuous support, valuable guidance and keen encouragement throughout my studies. Particularly, I would like to thank him for giving me precious comments and suggestions on my dissertation. He is a responsible supervisor who is always strict with my dissertation. He not only offers advices to me, but also gives me lots of inspiration on my life.

I would like to thank all the members in our System Group, **Cheng Wang, Jianyu Jiang, Zhiyang Li and Jingyu Yang**, for their supports and proposals. **Cheng** is my senior and often gives me a hand. He is the person who implements the FALCON and helps me embed FALCON into KVM. **Jingyu Yang** is a seasoned researcher and I would seek for his help not only for academic studies but also for life

I also appreciate my friends in Main Building Office 110 and 112. With the accompany of them, the study life seems relaxing and interesting. We create a comfortable working environment, and communicating with them in my leisure time delight me.

Finally, I would like to thank my close friend, **Shuangxi Wang**, for his encouragement on my study and life. He is an experienced senior specializing in distributed systems for many years. We shared the happiness and sadness with each other. Once I come across some problems, he would immediately turn up. He is such selfless that never refuses my request. Thank you.

List of Figures

Contents

Chapter 1

Introduction

This dissertation presents REPKVM, an efficient high availability solution for Kernel-based Virtual Machine (KVM) by using the approach of state machine replication (SMR). It allows virtual machines (VMs) to be available even in the case of hardware failures.

1.1 Virtualization

Recently, there has been a wide interest in server virtualization. With more than a decade's development, server virtualization has emerged as a powerful technique in data centers, yet many users still consider it as a new technology. But experts in this area have grown to believe it changes people's life.

Virtualization is used to create one or more Virtual Machines (VMs) that act like real machines in a physical host. Nevertheless, it provides some features which are impossible when constrained within a physical world. It facilitates flexible partitioning and dynamic allocation of computing resources. For each VM running in the host, it is just like a resource container who could schedule and execute the tasks by itself.

Multiple resource containers can be consolidated in a single physical server. Running applications in separate VMs on a single physical server cuts down on server waste by making full use of the resources in host. By allocating appropriate amount of CPU, memory resources the VM needs, it reduces various costs including power, space, etc. Since applications are running in an application virtualized environment instead of a native machine, it also becomes possible to run incompatible applications side-by-side.

1.2 The importance of high availability for VM

Unfortunately, consolidation exacerbates the consequence of unexpected host failures. When VMs are consolidated, failure of a single host may bring down multiple VMs on the host and all applications running inside, resulting in an unacceptable aggregate loss.

Thus, high availability (HA) services are essential for VMs to reduce the downtime of critical applications running inside. It requires that VMs build redundancy into architecture layers to mitigate against disasters. Generally, commercial systems tend to use dedicated hardware or customized software to realize HA. But surviving failure is notoriously hard and complex no matter which method is adopted. As host failures are inevitable, developing a highly available VM protocol has become a crucial task.

1.3 High availability solutions

The strong demand for HA solutions has generated many design strategies to provide fault tolerant servers. These strategies involve systems such as frequent checkpoints and deterministic execution. Many of the solutions are very reliable and popular but at the same time very inefficient in terms of performance overhead and resources cost.

A common approach to provide high availability is the primary-backup approach, where the backup server can take over immediately if the primary server fails. As the name suggests, there are always two servers *primary* and *backup* deploying in two physical hosts. Primary is responsible for receiving all the requests from the external network and executing corresponding commands while the backup is a copy of the primary. The state of the backup server must be guaranteed to be identical with the primary server at all times, so that the external requests could be executed in the backup server quickly once the primary fails. Only in this way the failures are hidden to the outside world and the state remains consistency. One distinguishing feature for primary-backup is to transmit all the changed state of the primary, including CPU, memory and I/O devices, to the backup nearly continuously.

However, the aggressive utilization of network resources to keep the backup consistent with the primary becomes the bottleneck of this approach. Primary-backup system is slow due to the large amount of state needs to be synchronized between the primary and the backup servers. The problem is aggravated by two characteristics. First, this approach fails to separate temporary state from other required state. Second, VM forces it to send the entire pages, rather than than the modified objects to the backup.

Another approach to achieve high availability is state machine replication (SMR). SMR regards the programs as deterministic state machines that are kept identical by starting them from initial state and ensuring the same requests are applied in the same order. It replicates a service on multiple physical servers so that the programs remain available even if one or more nodes fails. As long as the majority of the nodes is running normally, the system can still offer high-performance services.

1.4 Dissertation contributions

Our key insight is that we can embed SMR into VM to achieve HA with low performance overhead and high reliability. On the one hand, in contrast with primary-backup approach, SMR executes requests independently at each replica and does not need to propagate state modifications over the network. As the programs get exact same input in same order, they can directly execute input and reach a state that is consistent with each other. Therefore, less network communication is enough to guarantee consistency. On the other hand, SMR is a strong theory to provide high availability because it is resilient to various failure scenarios like packet loss or network partition.

Leveraging this insight, we present REPKVM, an efficient high availability solution for KVM virtualized environment by embedding SMR approach. With REPKVM, a VM just runs as is, and SMR helps keep the consistent state across replicas. REPKVM intercepts input and invokes the input coordination component of an SMR system called FALCON to efficiently enforce same sequence on inputs across replicas.

To replicate program servers in VM without obtaining the input of each server separately, REPKVM intercepts input and invokes SMR in networking layer. A naive approach for realizing highly available services is to integrate SMR into all applications running inside the VM. In this scenario, critical applications' states are guaranteed to be identical with the counterpart in other replicas where the replication is performed in application layer. However, SMR is notoriously difficult to implement and integrating SMR into every application involves the modifications of source code which leads to huge workloads for each application. Meanwhile, the networking in KVM is implemented in the software-based QEMU application. Intercepting the input is much easier in network.

In REPKVM, input preprocessing is quite necessary before propagating network input to other replicas. First, we should judge the significance of every input, so as to distinguish critical servers' input from other redundant information. Second, we should separate these servers' input and add a unique tag in its header. This header can be useful in the stage of synchronization.

We evaluated REPKVM on three widely used server programs in KVM virtualized environment, including a NoSQL database MongoDB, a key-value store Memcached, and a SQL server MySQL. Our initial results on popular benchmarks show that the proposed replicated VM's performance overhead is moderate (14.7% overhead on throughput and 17.9% on response time on average). Compared with another VM replication system Remus, REPKVM is much more efficient.

The main contributions of dissertation are as follows:

1. The idea of using SMR approach in networking layer for providing high availability to KVM.
2. The method of parsing network packets and getting the payload of critical applications in VMs.

1.5 Organization of the dissertation

The remaining of this dissertation is organized as follows. Chapter 2 discusses related work of virtual machine replication. First, it introduces the popular approach state transfer and some improvements based on this approach. Next, it presents record replay which is totally different from the previous one. Finally, it summarizes the strengths and weaknesses of existing solutions and describes the state synchronous approach which actually fosters strength and circumvents weaknesses. Chapter 3 provides relevant background information and an overview architecture of REPKVM system. It uses KVM and FALCON as its platform and the SMR system relatively. Chapter 4 describes the implementation details of the REPKVM protocol and the method to integrate the protocol into KVM. Chapter 5 reports the experiment results of the dissertation. It includes the analytical model, and the performance comparison between normal case and the replication method. Chapter 6 concludes the dissertation and identities the future work of virtual machine replication.

Chapter 2

Review of literature

Research of Virtual machine replication has been studied for many years both in industrial and academic circle. High availability system that aims to replicate the virtual machine could be implemented by using special hardware or some customized software. Either hardware or software implementation method requires expensive and complex redundant servers. In both case, the system could transparently survive failure and provide fault-tolerant servers.

Currently there are three main approaches to implementing virtual machine replication. The first method for replication is the primary-backup approach (It also can be called state transfer approach as it needs to transfer the state of VM.) where the backup server could take over and recover all the tasks when the primary server fails. The second one is the record replay. It is assumed that two virtual machines are started at the same initial state and they will receive the same requests at the same order. For the deterministic state machines, this approach is viable. The third method is the lightweight state synchronization approach. In fact, it is the combination of state transfer and record replay where two approaches actually complement each other. In this chapter, I will discuss these three approaches in detail.

2.1 State transfer

Remus is a typical state transfer approach which has been developed for many years. In this section, §2.1.1 introduces the basic algorithm of Remus and §2.1.2 discusses the limitations of this approach. §2.1.3 shows some improvements on the basis of state transfer.

2.1.1 Algorithm of Remus

Remus is a software system that provides high availability for applications or operating systems on commodity hardware. It takes advantage of the method in live migration [1, 11]. Live migration is a feature provided by the virtual machine to migrate a running guest operating system from one host to another. Remus extends VM's support for live migration to provide fine-grained checkpoints.

Remus runs VM in an active-passive mode. Paired VMs are deployed in two different physical hosts in local area network (LAN). The active primary VM is responsible for receiving the network requests from the clients while the backup VM remains silent to the external world. By propagating frequent checkpoints of the primary VM to the backup physical host, Remus achieve high availability for virtual machines.

The basic stages of operation in Remus are given in Figure 2.1. This procedure can be divided into six steps: (1) A client established a reliable connection with the primary VM and sends requests to it. (2) Without directly responding to the client, the primary VM buffers these output. (3) After an epoch, pause the running primary and collect all the changed state, including memory, CPU and I/O devices, into a buffer. (4) After that, the primary server resumes itself to forward progress and accept following requests from the clients. Replication engine in host operating system is responsible for sending these changed state to the backup. (5) The backup physical host who receives these buffered state information begins to synchronize immediately. (6) Finally, when all steps are complete, the backup server sends an acknowledgement to the primary which demands it to release the buffered output to the clients.

In Remus, different from the network traffic which is visible by external clients, the disk state is invisible. However, as an essential part of virtual machine replication, the disk state must be propagated to backup server just like the memory. In virtual machine, the dirty page memory is recorded by capturing the page fault. Through analyzing the information in fault, the dirty page can be clearly recorded. To maintain disk replication, all writes to the primary disks would be forwarded to backup server, where these writes are stored in memory first. After the changed state has been synchronized to the backup server and the backup responds with an acknowledgement. The primary server releases the output and send a signal to backup. It means the backup server could apply the writes to the disk right away.

The servers can hide the internal issues to the outside world when applying this technique in critical VMs. If the primary fails in this process, to the sight of the external, the state is consistent as no output is made externally visible. The senders just think that previous operations are not well performed in virtual machine and resend them to new primary again.

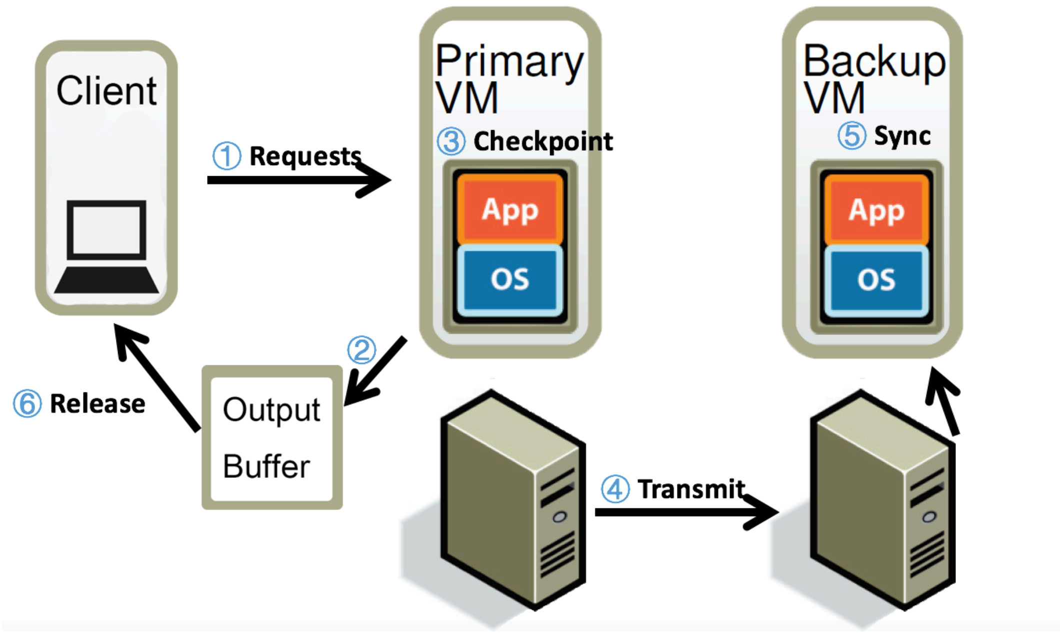


Figure 2.1: Virtual machine replication in Remus

2.1.2 Limitations of state transfer

The state transfer approach in Remus entails a potentially steep performance cost in spite of reliable replication. Most importantly, it needs to consume large amount of resources to propagate associated changed state to the backup nearly continuously, especially the dirty page memory. The state synchronization is working with the page-by-page basis and a small change within a page leads to the whole page transfer. No doubt, large changed memory would burden the host machine with state transmission. The underlying flow control is based on a stop-and-go model which is sensitive to the round-trip delay. If the network bandwidth between two host machines is not large enough, it directly affects the frequency of checkpoint. Furthermore, suspending the virtual machine too frequently reduces the service time of the primary, and in the suspension period, the resources cannot be allocated to other applications, which contributes to the low CPU usage. The suspension leads to the degradation of performance.

2.1.3 Improvements based on state transfer

The main bottleneck for this approach is the state synchronization of memory. Many optimization techniques have been proposed to reduce the performance overhead associated with transferring modified memory. Maohua Lu and Tzi-cker Chiueh [24] introduced three optimization methods, namely *fine-grained dirty region tracking (FDRT), speculative transfer* and *active slave*.

FDRT technique is trying to decrease the size of dirty unit. Since the primary needs to transfer the whole memory page in Remus, an obvious method is to accurately identity the regions of modified memory and propagate these regions to backup. The minimum unit in FDRT is dramatically smaller than the memory page size. And the smaller of the dirty block region is, the smaller amount of memory requiring forwarding. The whole procedure involves the calculation of block's hash value, hash value comparison and block memory transfer.

The idea of *speculative transfer* is that a dirty page can be sent to backup as soon as it is modified. There is no need waiting for the suspension of primary and the colleting dirty memory operation can be done at any time. However, it is possible that this method generates many redundant pages when many modifications are acting on a single memory page. A tradeoff between the risk of generating too many unnecessary memory and the benefit of speculative transfer should be balanced.

Active slave proposes the idea to run backup concurrently with the primary. For typical state transfer approach, the backup is passive and never consumes any CPU resources. However, for the active slave approach, both of them receive the same requests and execute the the operations. After a period of time, suspending these two virtual machines and computing the hash value of memory page. In this way, the difference between memory states can be tiny.

For the other improvements, Jun Zhu et al. [34] improved the performance by adding read fault reduction, write fault prediction and software-superpage to the Xen hypervisor. However, the problem of bandwidth occupation is still outstanding. Kemari [30] reduced the bandwidth usage by only triggering synchronization by I/O requests. However, the use case only favors storage services and it uses shared storage, which needs additional effort to avoid one point of failure.

2.2 Record replay

2.2.1 Replica-coordination protocol

Replica-coordination protocol was introduced in 1995, which provided a ground for later work of record replay. The difficulties of implementing replica coordination in the hardware, the operating system and the application programs caused the researchers to explore alternatives to replace them. Use of a hypervisor to implement replica coordination becomes attractive since it addresses all the problems caused by the hardware, operating system and programs. At the same time, hypervisor is a software layer implemented to serve virtual machines. In this way, replica coordination protocol based on hypervisor becomes a replication protocol which is used for achieving high availability virtual machines.

In this protocol, a set of given instructions must have the same effect whether it is executed by the primary server or the backup server. This protocol laid a solid foundation for record replay virtual machine replication.

2.2.2 VMware vSphere replication

VMware corporation embed the replication into virtual machine with the deterministic record replay approach [17]. Like Remus, for the virtual machine which we desire to replicate, we place another backup server on a different physical host. Nevertheless, VMware runs VM in an active-active configuration. It means that both primary and backup receive the input and produce the output in the process of execution. Therefore, the input-output model is extremely different from that of state transfer approach. The basic configuration is displayed in Figure 2.2.

For the input of the replicating VM, the backup server executes the same input as the primary server with a small time lag. Of course, only the primary server sends and receives network packets. The backup server remains silent all the time and only communicates with the primary server. The communication here means the logging channel in vSphere. To guarantee the backup server executes identically to the primary, we should ensure all the operations must be deterministic. For those non-deterministic operations, they must be executed in the same way.

Apart from the deterministic events in the operating system, there are still many non-deterministic operations requiring synchronization. How to correctly capture all the non-determinism to ensure the execution deterministic is the biggest challenge in this technique. The vSphere provides a log file to record all the non-determinism. When the input is captured by the vSphere, it makes a judgment immediately whether this operation is deterministic. If not, it means the primary needs to provide sufficient information to allow the backup server could reproduce the same state. During replay, the event is delivered at the same point in the instruction stream. VMware deterministic log-replay implements an efficient event recording and event delivery mechanism that employs various techniques, including the use of hardware performance counters developed in conjunction with AMD and Intel.

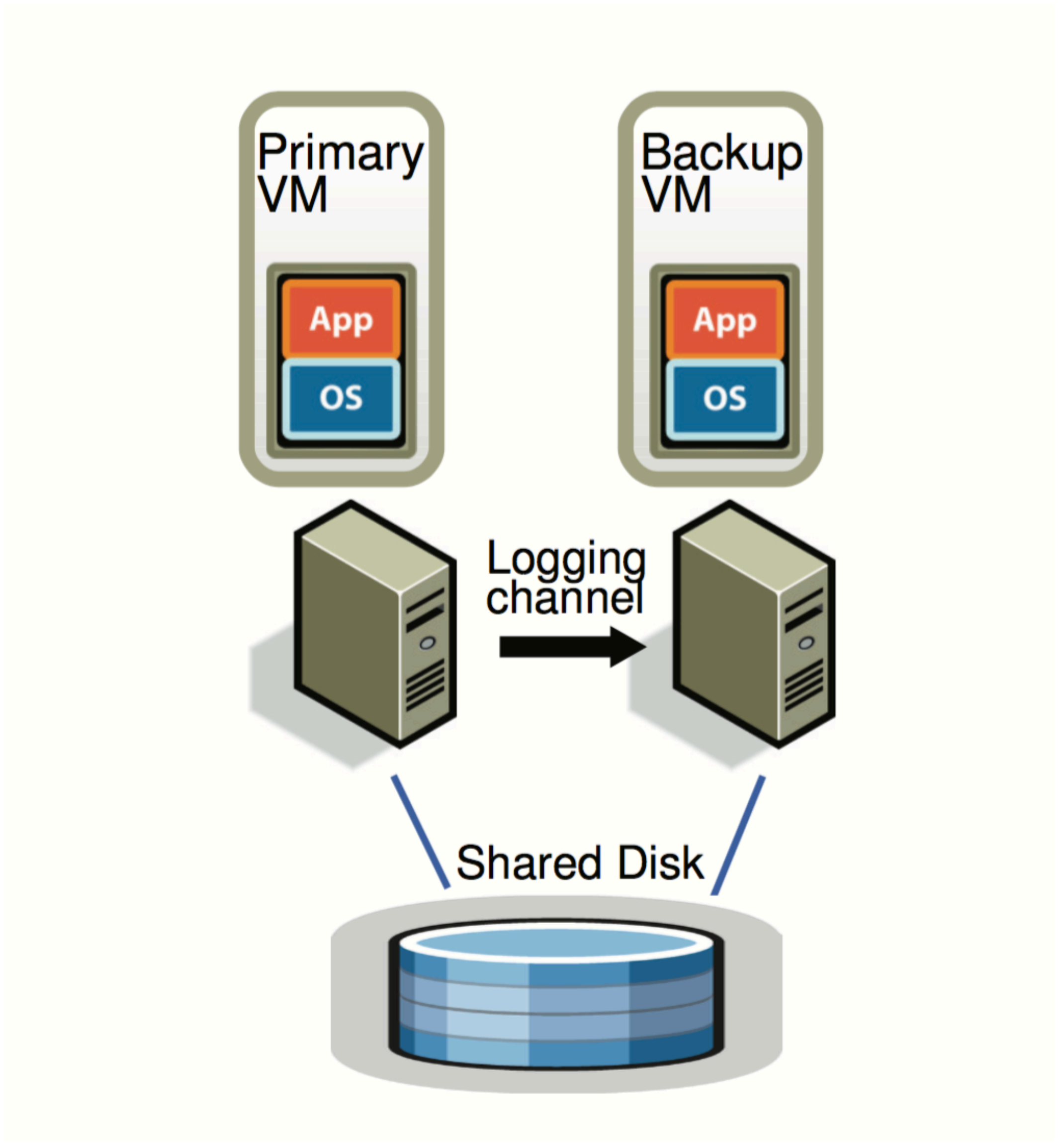


Figure 2.2: Basic fault tolerant servers in vSphere

The output includes write operations to the memory and disk. In order to simplify the output part, VMware utilizes the shared storage to store the disk. Both the primary and backup could read the information in shared disk but only the primary has the privilege to write to it. The primary's output would first be buffered until the backup virtual machine has received and acknowledged the log entry associated with the operation producing the output. The backup's output which should be sent to the clients will be dropped as the primary has already sent to the client side. The architecture of the fault-tolerant virtual machine and the protocol are displayed in Figure 2.3.

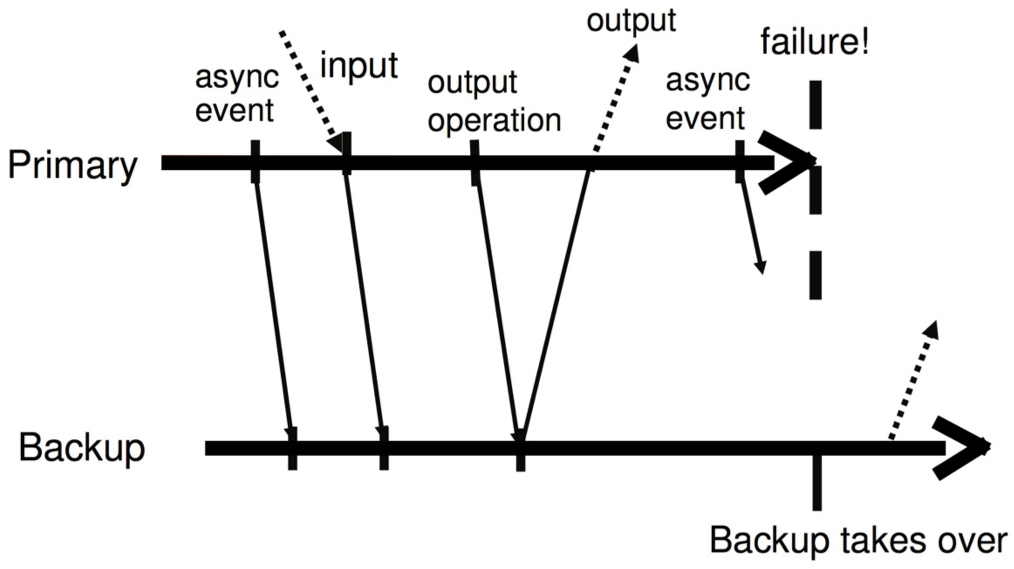


Figure 2.3: VMware replication protocol

2.2.3 Limitations of record replay

It is obvious that compared with the state transfer, the record replay approach is more convenient as it only needs to send small amount of log information. With this log file, the state between two virtual machines are identical with each other. There is no need to ship all the changed memory which consuming large amount of network bandwidth resources.

However, there are still some limitations for this approach, first of all, as I mentioned above, the record replay approach cannot efficiently resolve the problem of non-deterministic operations. For the non-deterministic operations like clock functions, the primary server must collect enough data to reproduce the same state in backup server. In this way, the developer must know all the non-deterministic operations in advance. Otherwise, the primary server may lose sight of some non-deterministic operations which contribute to the inconsistent between two machines.

Secondly, it cannot support multi-thread in guest operating system. The multi-thread application would produce some concurrency bugs if some codes are not locked when executing. In the virtual machine, multi-thread would account for the inconsistency between virtual machines. The developer cannot ensure the operations in multi-thread applications are executed in the same order. So the results of execution may vary.

Thirdly, this approach has the split-brain problem. It cannot effectively determine which virtual machine is more up-to-date when the old primary fails and comes back while the backup server becomes the new primary. Although it proposes a good method to determine who is the primary by using the shared disk. The shared disk is just like the third party to judge who is the primary like the SMR protocol. However, if the record replay approach is running in two different VMs who are not willing to share the disk, the third party decider becomes difficult to find.

2.3 Lightweight state synchronization

Lightweight state synchronization is actually the combination of SMR and state transfer approach. SMR can help state transfer approach overcome the limitation of synchronizing large amount of state while state transfer approach can benefit SMR by repairing the divergence of execution states. Leveraging this insight, it is possible to implement high availability KVM by combing SMR and a lightweight state-transfer approach. It could effectively resolve the practical problem of non-determinism.

To detect and repair divergence, it leverages an efficient network output checking mechanism provided by FALCON. It could be implemented by calculating an accumulated hash value by interposing on the server programs' network outputs and the periodically invoking the output checking mechanism of FALCON. If divergence of execution states between replicas is detected, the replicated virtual machine triggers a lightweight state transfer process. This process works by first computing hashes of the changed states, then making an efficient comparison to identity the divergent states and finally transferring them.

To do this, each replica maintains a Merkle tree kept updating from the dirty pages. By "dirty", we mean the memory the memory page has been modified since last synchronization. One problem here is that each VM may have dirtied different pages. This leads to different structures of Merkle trees and they are not comparable. This approach addresses this problem using a fast method as follows (It uses A to denote the diverged replica; it will fetch state from replica B).

1. A sends its dirty page bitmap $M_B$to B.
2. B does a bitwise or calculation $M_{union} = M_A|M_B$, and sends M_{union} back to A.
3. A and B both update their own Merkle tree according to M_{union}. A sends its updated Merkle tree to B.

4. B dose a comparison on both Merkle trees to find the differences, as illustared in Figure 2.5.

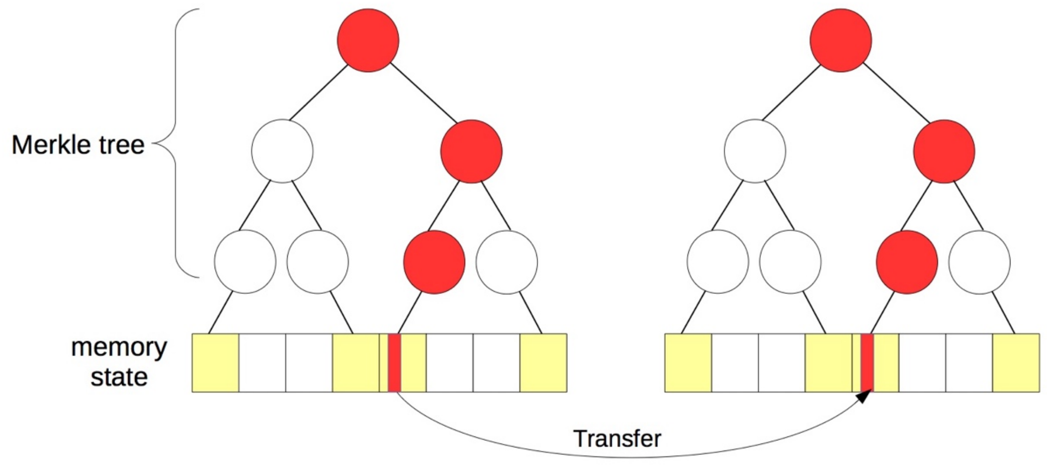


Figure 2.5: Identifying divergent states using Merkle tree. Blcoks in yellow are dirtied memory. Diverged memory state are shared in red. Divergent states can be located by traversing the red nodes of the Merkle tree.

This method is efficient for two reasons. First, it only transfers the different pages between replicas. Because it has made consensus on inputs and it can turn off *Address made consensus Randomization* to trade for correctness, the difference rate will be really small. Second, the use of Merkle tree optimizes the time complexity of comparison between memory states to *O(logn)*.

2.4 Summary

In this chapter, we studied the related work on the virtual machine replication. Overall, the research on virtual machine replication is mature. The two types of replication, record replay and state transfer, both has its own pros and cons.

State transfer checkpoints the state of the primary periodically and propagates the state updates to the backup. However, the frequent checkpoints may bring in large performance overhead. The propagation of states may take up precious network bandwidth and increase latency to incoming requests.

VMware integrated the record replay approach into its product vSphere together with several performance optimizations. However, recording the interleaving of multi-core processors and replaying them deterministically still shows large overhead.

Although the lightweight state synchronization approach performed better than the other two, the detailed algorithm still needs improving.

Chapter 3

Architecture

This section introduces the architectures of two key techniques used in this dissertation, Kernel-based Virtual Machine, with reference to its network I/O architecture in QEMU-KVM [9] and the FALCON SMR system.

3.1 FALCON

FALCON is a typical kind of Paxos protocol implemented with RDMA library. We run replicas in our cluster where all servers are connected to a switch. Every replica registers a memory for itself and connects with each other using RDMA QPs. And every replica has the privilege to access the memory in other nodes.

We chose FALCON because it is fast and robust. In a FALCON-replicated system, a set of 2f+1 machines (nodes) are connected through an interconnect with support for RDMA. Once the FALCON system starts, each replica would try to establish reliable connections with others. After all the connections are created among replicas, it would elect a leader which proposes the order of requests to execute. An arbitrary number of clients in LAN or WAN send network requests to the leader and then get responses from it.

FALCON is a strong consensus algorithm consisting of four main steps. On receiving network request from a client, the leader node would first intercept the requests and get the real data from relevant packets. The second step is local preparation, add the viewstamp information for this consensus log and write it to a local parallel logging storage on SSD. Third, it would invoke a RDMA-based distributed consensus protocol on this request to enforce that all the replicas see the same input. The last step is the leader collecting the ACKs from the replicas, once it reaches Quorum, it means the consensus is completed. Benefiting from RDMA and FALCON's fast protocol, each consensus process takes less than 10us. Compared with TCP/IP-based Paxos protocol, whose latency was always over 600us, FALCON has the advantage of lower performance overhead.

3.2 KVM

We use Kernel-based Virtual Machine as the platform of our study mainly because of the full capacity of user-space tools in its I/O device emulator process. KVM [20] is a full virtualization or hardware-assisted virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V). It consists of two modules, namely, kvm.ko and an architecture dependent kvm-amd.ko or kvm-intel.ko module. Under KVM, every virtual machine is a regular Linux process handled by the standard Linux scheduler by adding a guest mode execution which has its own kernel and user modes.

KVM becomes the new industry standard in virtualization with multiple operating system support. Its source code has been merged into Linux kernel after version 2.6.20 and a wide variety of guest operating systems could be run in the KVM such as Linux, OpenBSD, Windows and FreeBSD benefit from the full virtualization technique. By comparing KVM with other products such as VMware and Xen, it is easy for us to find KVM support in more situations and has better performance.

3.3 QEMU-KVM

QEMU-KVM is an assistive tool for KVM. In our system, KVM is responsible for providing the hardware acceleration for VM and boosting its performance. However, KVM by itself cannot provide the complete virtualization solution. As a matter of fact, it also requires the QEMU-KVM emulator which is located in the user space to provide emulated I/O devices.

A virtual Ethernet bridge is installed in physical machine to allow all VMs to appear on the network as individual hosts. Figure 3.1 shows the architecture of the virtual Ethernet bridge. In this configuration, a software bridge is created in the physical host (br0). The backend virtual network device (vneth0) are created and added to this bridge connected with the physical Ethernet device (eth0). In VM, a virtual network device consists of a pair of devices (e.g. vneth1 and eth0). The frontend resides in the VM while the backend resides in the physical host. This pair of devices shares the same MAC address and configuration which contributes to smooth network packet transition between guest and host.

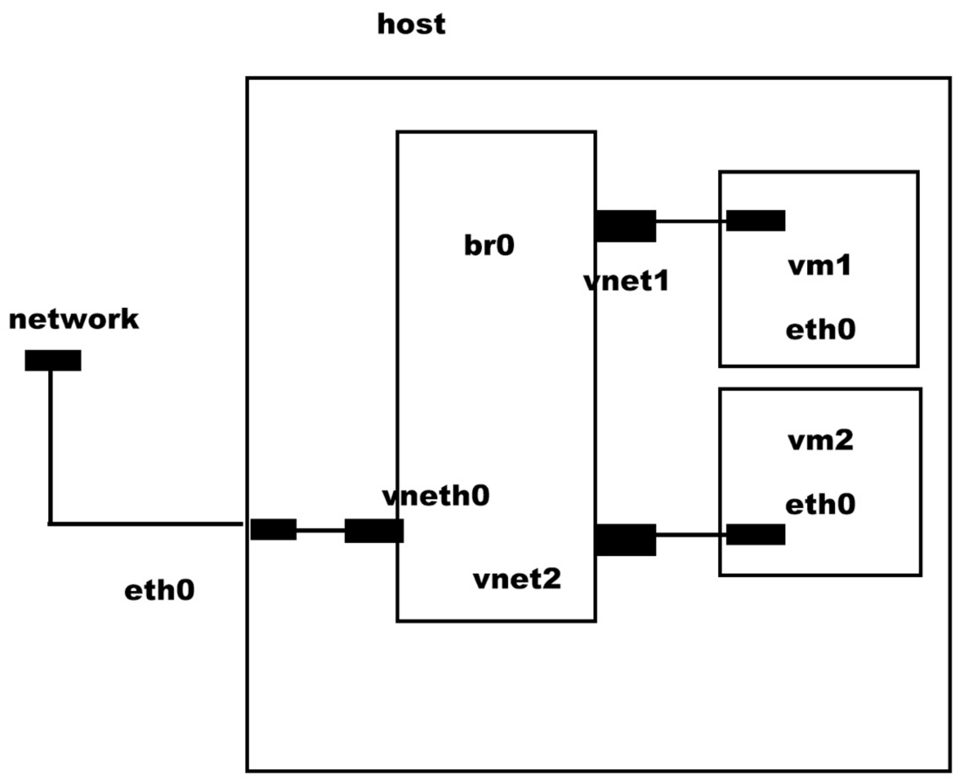


Figure 3.1: The architecture of virtual bridge

The networking in KVM is implemented in the user space QEMU-KVM application. A typical packet flows through the KVM virtualization host in the following way. When a packet arrives at the physical NIC in the host, interrupts generated by NIC are handled by the physical driver. Then the physical NIC device driver handles the interrupts by transferring the packet to the virtual Ethernet bridge, which forwards the packet to the appropriate virtual machine's front end interface. The front end interface we used in QEMU-KVM is the TAP device which is entirely supported in software. It simulates an Ethernet link layer device which operates the Ethernet frame. The TAP device would create a file descriptor in host machine when we start it. And this file descriptor is responsible for receiving all the frames from the local area network. When the network packet arrives at the TAP device, the TAP device sends a signal to the KVM driver which sends a virtual interrupt to the QEMU in the guest notifying it of the new packet. Once receiving the virtual interrupt, QEMU-KVM delivers the packet to the guest operating system's network stack. The architecture of the whole network flow can be seen in Figure 3.2.

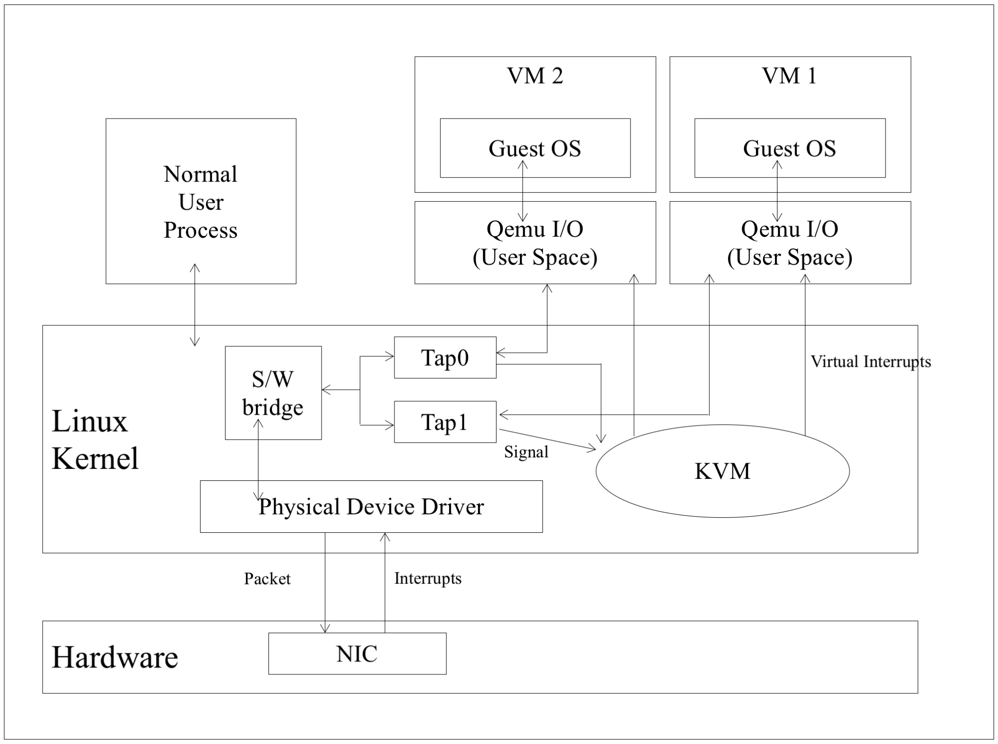


Figure 3.2: The architecture of KVM network

3.4 Overall architecture

The overall architecture of my dissertation is shown in Figure 3.3. The QEMU-KVM is responsible for the emulation of network devices. Through installing a virtual bridge in host and creating a virtual software-based network device TAP for every virtual machine, the network packet can be easily obtained from QEMU-KVM. FALCON, the SMR system running in the user space of physical host, plays a major role in VM replication. It intercepts the network packet and forwards them to its counterpart in other replicas. Finally, FALCON also applies all the changes to the the replicas.

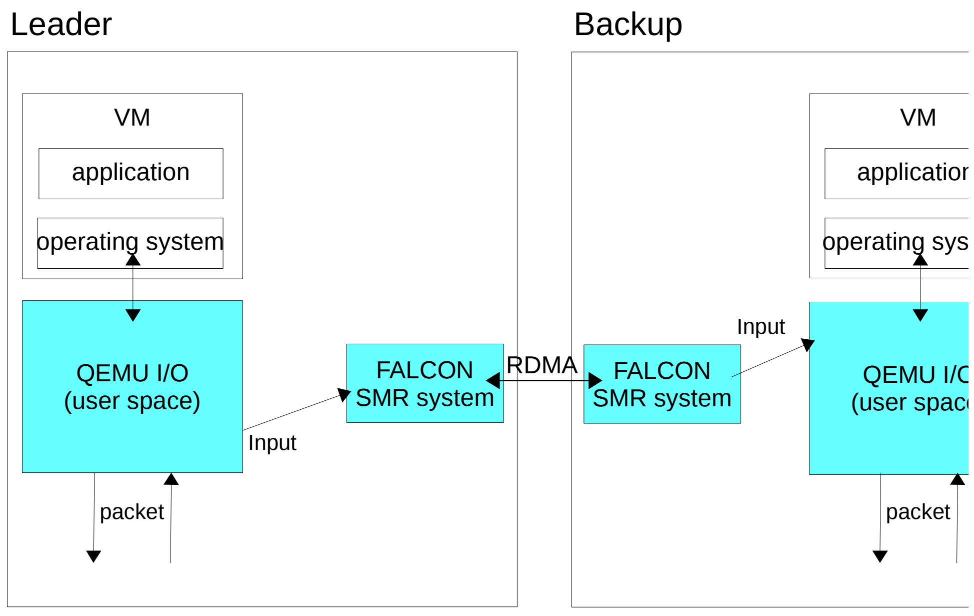


Figure 3.3: Overall architecture of replicated VM

Chapter 4

Implementation

In this chapter, we describe the implementation details of REPKVM. §4.1 discusses the process of intercepting the network packet from the QEMU-KVM. Then, §4.2 focuses on the preprocessing of packet and §4.3 describes the method of imbedding FALCON into KVM.

4.1 Network interception

The TAP virtual network device is used to support network transmission for VM. Packets sent by a host OS via a TAP device are delivered to VM which attaches itself to the device. VM can also pass packets into a TAP device. In this case the TAP device delivers these packets to the host OS network stack thus emulating their reception from an external source.

In order to intercept the network packets of server programs running inside the VM, we interpose on the tap_send() function in QEMU-KVM. The tap_send() function is called to check whether there is any frame received by the TAP device. And the tap_receive() function is used for receiving frame from the VM. These two functions are jointly concerned with a file descriptor. The details of interception process could be described as follows. The file descriptor is created before the activation of VM. QEMU-KVM keeps reading this file descriptor using the system call read and a non-zero integer is returned when there is a packet received by TAP device. Then, qemu_send_packet is called to send packet to the upper layer asynchronously.

4.2 Preprocessing of the packet

The packets we intercepted contain the information of Ethernet header, IP header and TCP header [36]. The data is followed after these headers. To get the data of the packets, we should understand the structure of Ethernet frame first. Furthermore, not all the packets we intercepted are critical applications' packets. It contains other network packets including ARP, RARP, ICMP and IGMP. It is necessary to distinguish the critical applications' packet from others.

Through printing out the information in the packet, we find the initial 10 bytes of the packet is used for judging the type of the packet for the QEMU-KVM. We call the initial 10 bytes qemu header. So, deleting the qemu header, the rest is a whole Ethernet frame.

An Ethernet frame is preceded by the qemu header and starts with an Ethernet header. The Ethernet header contains destination and source MAC address as its first two fields (each six octets in length). The destination and source presents the MAC address of VM and client-side separately. The middle section of the frame is EtherType field. It can be used for two different purposes. Values of 1500 and below mean that it is used to indicate the size of the payload in octets, while values of 1536 and above indicate that it is used as an EtherType, to indicate which protocol is encapsulated in the payload of the frame. As the packet captured uses this field as an EtherType and critical server program's network packet must be a IP packet, EtherType field shoud be separated first to judge whether it contains a IP header. The EtherType value of 0x0800 signals that the frame contains an Internet Protocol version 4 datagram. We could drop all the other frames except IPv4 datagram. The structure of the Ethernet frame is displayed below (Figure 4.1).

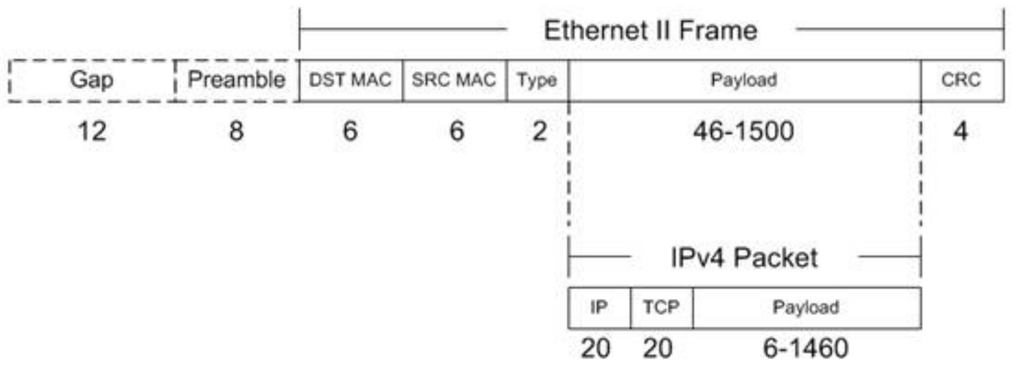


Figure 4.1: Structure of Ethernet header

Like Ethernet protocol message, IP uses a specific format for its datagrams. The IPv4 datagram is conceptually divided into two pieces: the header and the payload. IP header is much more complex than Ethernet header (Figure 4.2). It contains addressing and control fields. Although two different versions of IP headers have been defined (IPv4 and IPv6), the VM is only assigned with a IPv4 address. Thus there is no need to consider IPv6 header. In IPv4 header, the control information we need to acquire include three fields: source IP address, destination IP address and the length of IP header. For the other information like service type and header checksum, we did not take them into consideration. The preprocessing of IP datagram can be divided into three steps. First, with the

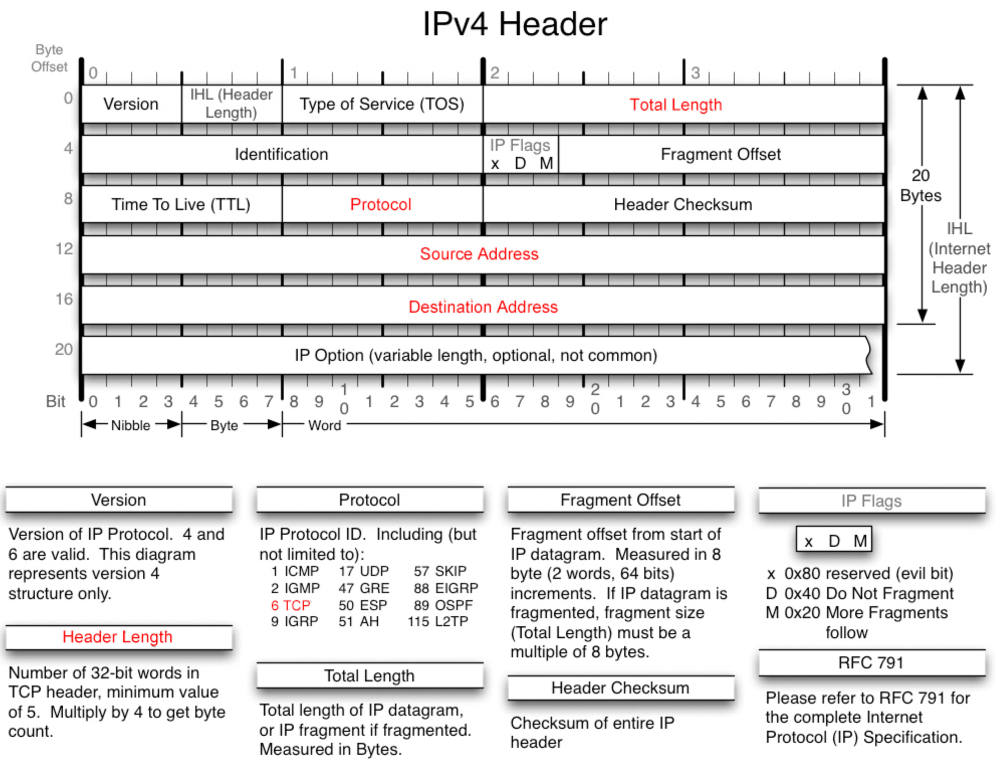

IPv4 Header
Byte Offset
Version
IHL (Header Length)
Type of Service (TOS)
Total Length
Identification
IP Flags x D M
Fragment Offset
Time To Live (TTL)
Protocol
Header Checksum
Source Address
Destination Address
IP Option (variable length, optional, not common)
20 Bytes
IHL (Internet Header Length)
Bit
Nibble
Byte
Word
Version
Version of IP Protocol. 4 and 6 are valid. This diagram represents version 4 structure only.
Header Length
Number of 32-bit words in TCP header, minimum value of 5. Multiply by 4 to get byte count.
Protocol
IP Protocol ID. Including (but not limited to):
1 ICMP 17 UDP 57 SKIP
2 IGMP 47 GRE 88 EIGRP
6 TCP 50 ESP 89 OSPF
9 IGRP 51 AH 115 L2TP
Total Length
Total length of IP datagram, or IP fragment if fragmented. Measured in Bytes.
Fragment Offset
Fragment offset from start of IP datagram. Measured in 8 byte (2 words, 64 bits) increments. If IP datagram is fragmented, fragment size (Total Length) must be a multiple of 8 bytes.
Header Checksum
Checksum of entire IP header
IP Flags
x D M
x 0x80 reserved (evil bit)
D 0x40 Do Not Fragment
M 0x20 More Fragments follow
RFC 791
Please refer to RFC 791 for the complete Internet Protocol (IP) Specification.

Figure 4.2: Structure of IP header

destination IP address, we could judge if the virtual machine is the single destination of this datagram. For the broadcast or multicast messages, the destination IP address would not equal to the IP address of the VM. These packets obviously are not critical applications' packets. Second, the protocol information in the 13 bytes offset of the IP header represents the IP protocol. Here, only TCP messages with the value 0x06 is accepted. Third, the IP header may include some options, the length of the header is not a constant value. It is necessary to get the length of IP header and calculate the first byte address of TCP segment. The structure of the IP header is displayed in Figure 4.2.

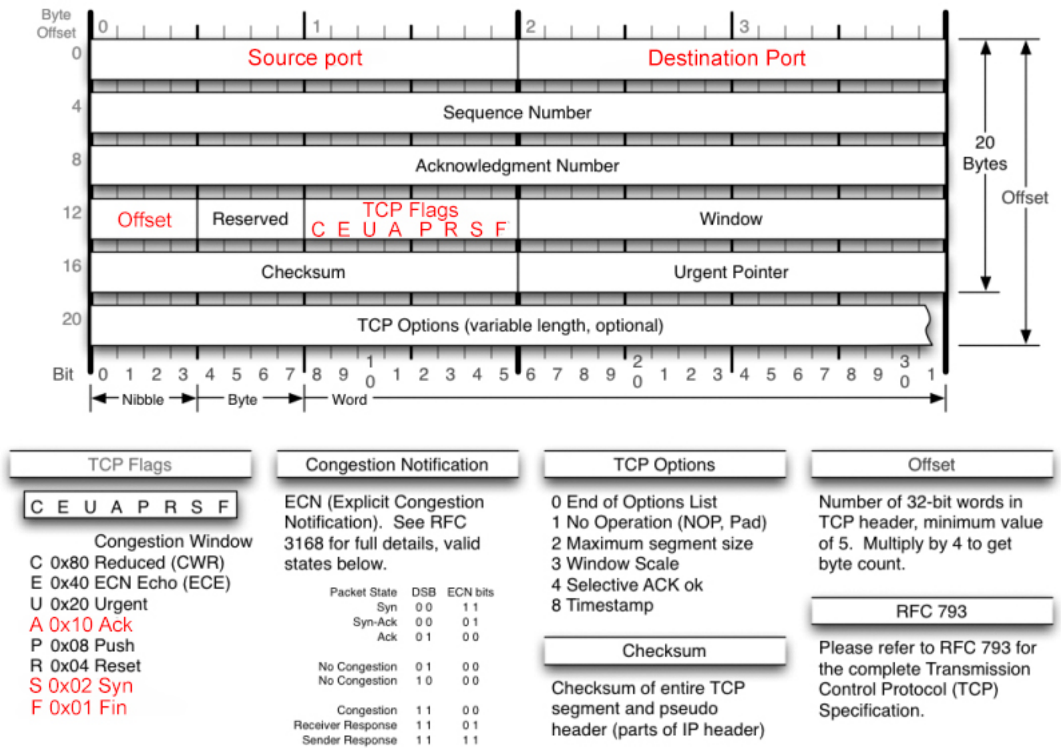


Figure 4.3: Structure of TCP header

TCP is considered as a reliable protocol (Figure 4.3). It checks to see if everything that was transmitted was delivered at the receiving end. And it also allows for the retransmission of lost packets, thereby making sure all data transmitted is eventually received. Due to its reliability, most critical applications prefer to use the

TCP protocol to create reliable connections with the clients to exchange the information. Here, we would discuss the structure of the TCP header in detail.

From the figure, it is shown clearly that the TCP header has 10 mandatory parts, plus a TCP option field. The real data follows this header. The important fields contain source port, destination port, offset and TCP flags. We should first record the critical applications' ports, only the packets with these ports could be replicated in other VMs. As for the source port, it is a good field to distinguish the clients so as to support concurrent connections. The size of the TCP flags is 8 bits. It represents 8 different flags in TCP header. From left to right, they are CRW, ECE, URG, ACK, PSH, RST, SYN and FIN. To make preprocessing simple, we only consider the flag ACK, SYN and FIN. SYN flag means the client wants to synchronize sequence number with the server. This TCP segment is the beginning of connection establishment. ACK flag indicates the acknowledgement field is effective, and the acknowledgement number is set in this TCP segment. This flag has to be set in every packet after the connections between the client and server has been established. Otherwise, this packet would be dropped by the operating system of the virtual machine as it is not a significant packet. The FIN flag is the symbol of disconnection. It means there is no data from the client anymore. These three flags are corresponding to three relative operations in FALCON. The detail will be discussed in the next section. Finally, the offset here is similar with the length of the TCP header. With the offset, we could easily get the first byte address of the real data part. Of course, not all the network packets include data part, before transporting the data part to the FALCON, we also have to calculate the length of this part. As long as the length is larger than zero, we would forward the address of the data to FALCON.

4.3 Embed FALCON into KVM

To make the whole process simple, Embedding FALCON into KVM could be completed in two steps. First, add the preprocessing codes into FALCON, compile the whole codes and produce a shared library. Then, add this shared library into QEMU-KVM, rewrite the function in it.

As the functions in preprocessing codes would be called frequently, we directly add the codes into FALCON. The three TCP flags are corresponding to three different operations in FALCON. When the SYN symbol is marked in network packet, it indicates a reliable connection is established for the leader. FALCON would forward this message to the replicas and demand the host machine create an identical connection for replicas. When the ACK symbol is marked in network packet, the real data, port of the destination information would be grouped together and sent to the FALCON. The FALCON in the replica side would parse the log and send the real data to the virtual machine in replica through the connection established previously. Finally, the FIN flag would invoke the function deleting the related file descriptor in the replica's host machine, in other words, it is a disconnection operation.

After adding the codes into FALCON, it is of great significance producing a shared memory. With this shared memory, it becomes much easier for embedding the whole FALCON into QEMU-KVM.

As we have produced the shared library libfalcon.so, the following task is to link the library with QEMU's executable file. In the main function of QEMU, we directly invoke the RDMA initialization function to establish connections between replicas. Only after initialization is completed would VM begins to run.

4.4 Summary

The steps to run a replicated virtual machine is not complicated. First, intercept the network packet in the source code of virtual network device. Then, analyze the network packet and preprocess the packet. Finally, embed FALCON into KVM and run a virtual machine with replication.

Chapter 5

Evaluation

Our evaluation was done on a set of three replica machines, with each having Linux 3.16.0, QEMU-KVM 1.2.50, 2.6 GHz Intel Xeon with 24 hyper-threading cores, 64GB memory, and 1TB SSD. Each machine is equipped with a Mellanox ConnectX-3 Pro Dual Port 40 Gbps NIC. These NICs are connected using the Infiniband architecture. For the VM, 2GB memory and 4 CPU cores are allocated to every replica. Except the difference of IP address and Mac address of these VMs, they are identical in the other aspects.

To mitigate network latency, benchmark clients were running within the replicas' LAN. The average ping latency between the client machine and a virtual machine on the server machine is 210us. Larger latency will mask FALCON's overhead.

We evaluated the performance of REPKVM on three widely used server programs, including MySQL, an SQL database; a key-value store Memcached [25]; and a NoSQL database MongoDB [26]. REPKVM's high availability and fault-tolerance are attractive to these server programs, because these programs provide on-line service and contain important in-memory execution states and storage. We compared REPKVM with Remus, a well-known software-based system that provides high availability using the state transfer approach.

The rest of this chapter focuses on these questions:

§5.1: What is the performance overhead of running REPKVM with general server programs?

§5.2: How much data does REPKVM need to transfer compared to Remus?

§5.3: How to quantity the availability of REPKVM? How fast is REPKVM on electing a new leader?

5.1 Performance Overhead

We evaluated the performance of REPKVM on MySQL, Memcached and MongoDB. MySQL is an open-sourced relational database management system where the data is permanent stored in disk. Memecached is a high-performance distributed memory object caching system. It provides a large hash table and uses CRC-32 to calculate the key. Users could use Memcached as the cache of some slow backing databases. Here we evaluate Memcached as a key-value database. MongoDB is a free and open-source NoSQL database widely used in distributed system. It can run over multiple servers, balancing the load or duplicating data to keep the system up and running in case of hardware failure.

REPKVM ran with these server programs without modifying them. No special configuration is applied in programs except the listen IP address. For Mysql, we used benchmark Sysbench [4] to generate random select queries. For Memcached, we used twemperf benchmark [3] from Twitter to issue mixed set and get operations. The size of every fixed item is 10 bytes. Every connection is created after the previous connection is closed. We sent total 10000 requests in every test. For Mongodb, we used Yahoo Cloud Serving Benchmark (YCSB) [5]. Every time, we sent 10000 set requests to server in VM to get the average throughput and latency.

The Figure5.1, Figure 5.2 and Figure 5.3 shows the throughput and latency of MySQL, Memcached and MongoDB separately. We varied the number of concurrent client connections for each server application from 1 to 8 threads. As the number of threads increase, all programs' unreplicated executions got a performance improvement except Memcahed. The reason is that every connection is created after the previous connection is closed in our benchmark.

Overall, compared to these server programs' unreplicated executions, REPKVM's average overhead on throughput and response time was 14.7% and 17.9% respectively. REPKVM scaled almost as well as the unreplicated executions.

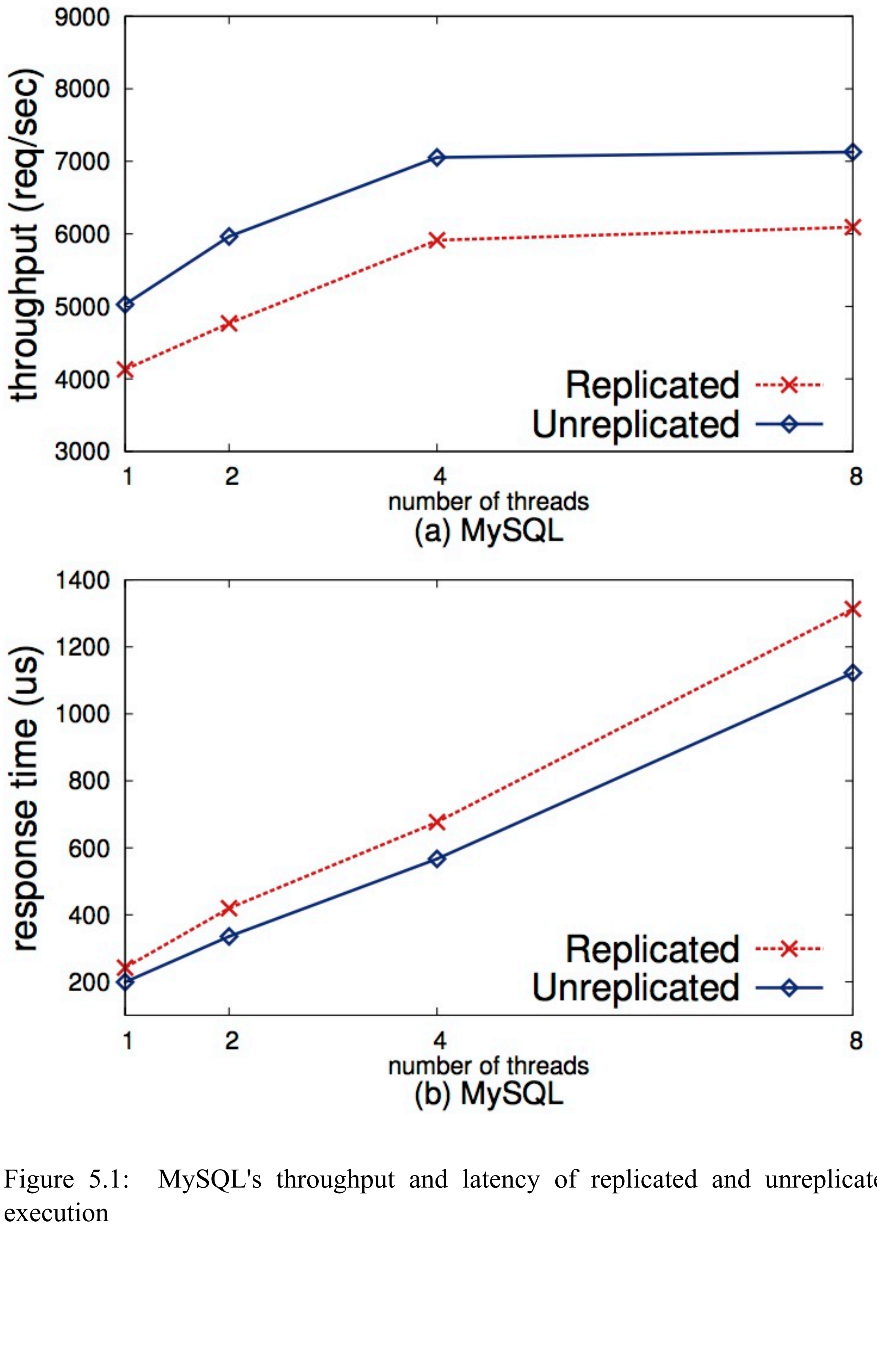
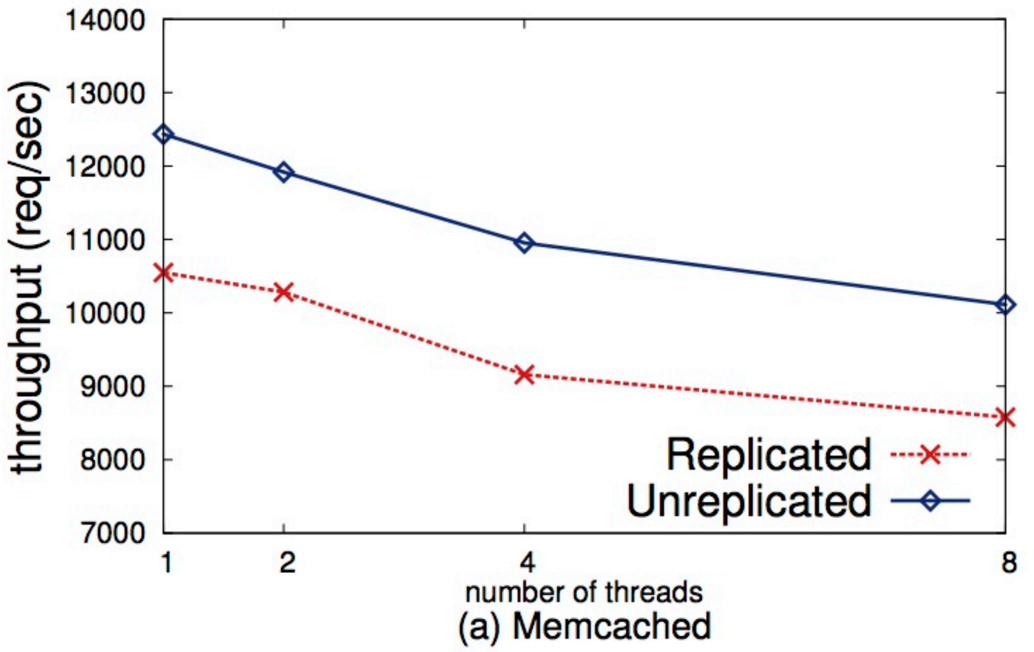


Figure 5.1: MySQL's throughput and latency of replicated and unreplicated execution



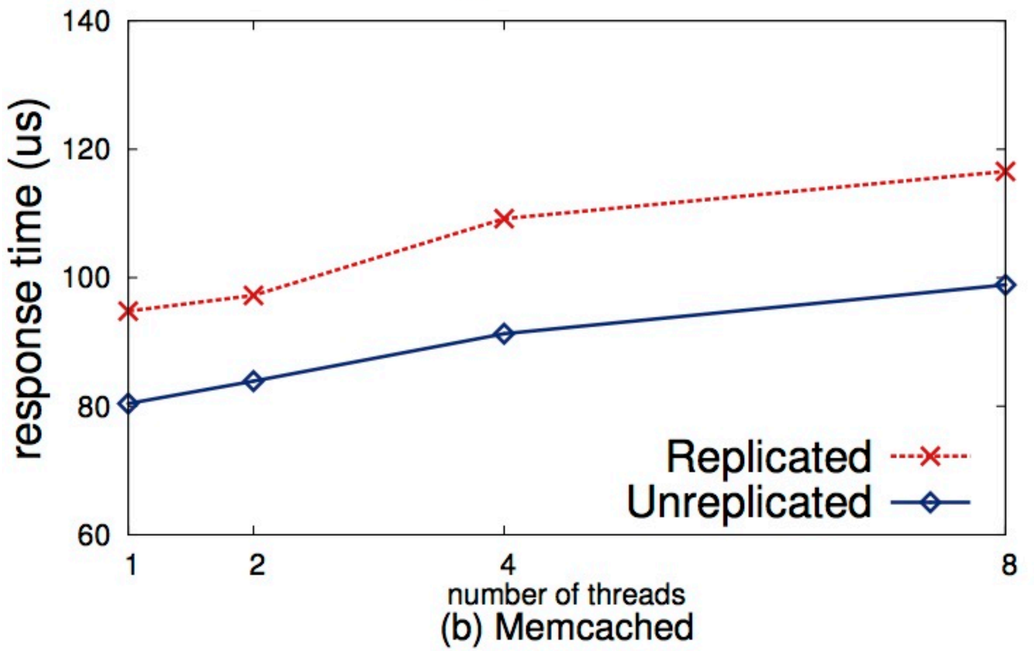
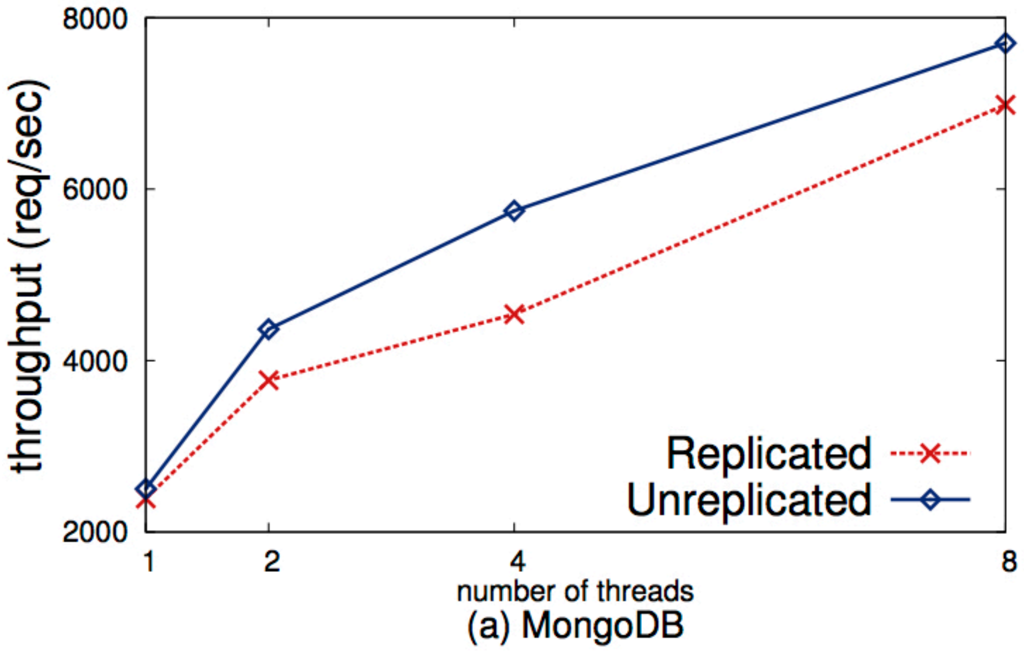


Figure 5.2: Memcached's throughput and latency of replicated and unreplicated execution



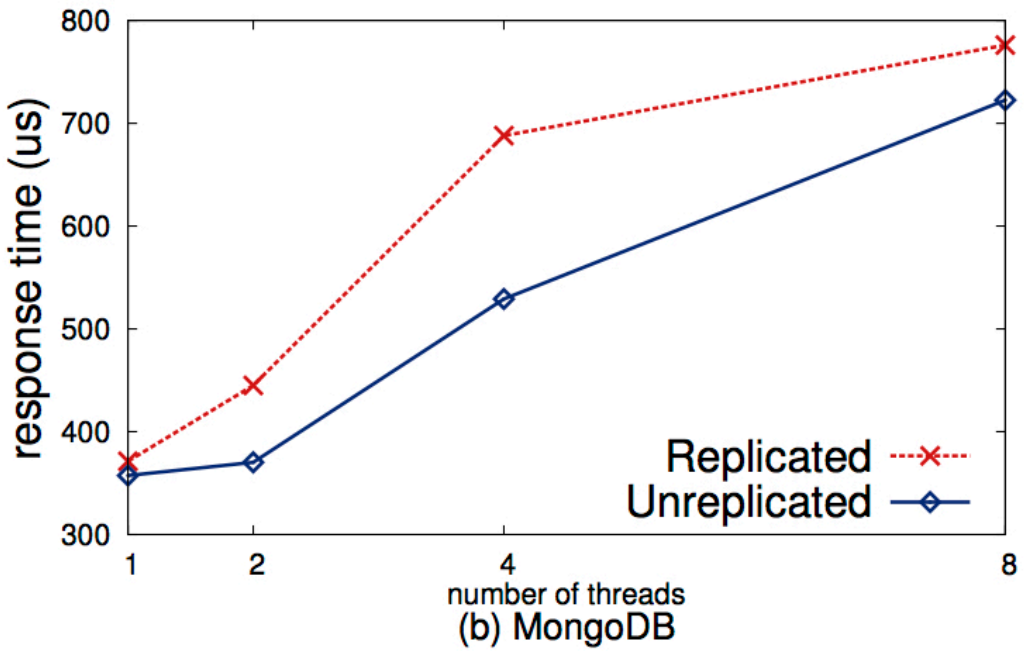


Figure 5.3: MongoDB's throughput and latency of replicated and unreplicated execution

5.2 Comparing with Remus

Remus provides high availability by continuously sending modified state from primary to the backup server. A drawback of Remus we mentioned is that it aggressively utilizes network resources to keep the backup consistent with the primary. It aggregates this issue by sending the entire dirty memory pages. On the other hand, SMR approach only needs to synchronize the network inputs.

Since we have not installed Xen on our machine, we modified the source code of KVM to estimate the amount of data that Remus needs to transfer. We used a thread to record to record the number of data that Remus needs to transfer. We used a thread to record the number of dirty pages every 25ms, the same frequency as in Remus paper. Our experiments show that, for a million requests, Remus needs to transfer 13 GB of data for MySQL, 2.4 GB for Memcached and 3.9 GB for MongoDB. On the other hand, the only data which SMR approach needs to transfer is the requests themselves, which is only tens of MB. The amount of data to be transferred of our replicated VM has been reduced by about 90%.

5.3 Availability and recovery

We evaluate high availability of replicated VM by measuring the recovery latency of services. For SMR approach, two roles are enrolled in the algorithm, leader and acceptor. Only the leader has the privilege to propose a value and send this value to all the acceptors. The deficiency of the leader means the downtime of the whole system. In my dissertation, I tried to evaluate HA by measuring the time of electing a new leader in our system.

To measure the latency of leader election of our replicated VM system, we deploy three VMs in different physical hosts. We then manually killed the leader and got the time of completing a new round leader election. We got the latency of electing a new leader is around 10us. Because SMR leader election is rarely invoked in practice, we can conclude our system is high available.

Chapter 6

Conclusions

In my dissertation, I aim at exploiting the replication feature of KVM for providing fault tolerant servers in virtual machines. I research on the state machine replication for KVM where incoming network packets are interposed and FALCON is invoked to agree on the inputs across replicas.

Compared with the traditional state-transfer based high availability system Remus. We can find that the amount of data transferred for REPKVM is much less. Different from the Remus requires to transport changed memory, CPU and I/O device information frequently, our replicated virtual machine method provides high availability feature by replicating a small amount of network packets.

Evaluation on three widely used programs shows that our proposed replication method achieves moderate overhead. Even in the LAN, the throughput is not dropping fiercely, and there is just a little increase in response time.

Reference

[1] Live migration on KVM. http://www. linux-kvm.org/page/Migration.

[2] Paxos made more scalable with RDMA. Technical report, The University of Hong Kong.

[3] A tool for measuring memcached server performance. https://github.com/t witter/ twemperf, 2004.

[4] SysBench: a system performance benchmark. http://sysbench.sourceforge. net, 2004.

[5] Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/Y CSB, 2004.

[6] MySQL Database. http://www.mysql.com/, 2014.

[7] 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. In *SOSP ’03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, 2003.

[8] L. A. Barroso, J. Clidaras, and U. Holzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[9] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[10] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Dec. 1995.

[11] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[12] H. Cui, R. Gu, C. Liu, and J. Yang. Paxos made transparent. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Oct. 2015.

[13] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174. San Francisco, 2008.

[14] G. Dunlap, S. T. King, S. Cinar, M. Basrat, and P. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 211– 224, Dec. 2002.

[15] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM*

SIGPLAN/SIGOPS international conference on Virtual execution environments, pages 121–130. ACM, 2008.

[16] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the speed of multicore. In *Proceedings of the 2014 ACM European Conference on Computer Systems (EUROSYS '14)*, page 11. ACM, 2014.

[17] V. Infrastructure. Automating high availability services with vmware ha. *VMware Technical Note.*

[18] V. Infrastructure. Resource management with vmware drs. *VMware Whitepaper*, 2006.

[19] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, et al. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, volume 12, pages 237–250, 2012.

[20] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[21] L. Lamport. Paxos made simple. http: //research.microsoft.com/en-us/um /people/lamport/pubs/paxos-simple. pdf.

[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, 1978.

[23] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[24] M. Lu and T.-c. Chiueh. Fast memory state synchronization for virtualization-based fault tolerance. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 534–543. IEEE, 2009.

[25] https://memcached.org/.

[26] Mongodb.   http://www.mongodb.org, 2012.

[27] M.Nelson, B.-H.Lim, G.Hutchins, etal. Fast transparent migration for virtual machines. In *USENIX Annual technical conference, general track*, pages 391–394, 2005.

[28] D. J. Scales, M. Nelson, and G. Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *ACM SIGOPS Operating Systems Review*, 44(4):30–39, 2010.

[29] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[30] Y. Tamura, K. Sato, S. Kihara, and S. Moriai. Kemari: Virtual machine synchronization for fault tolerance. In *Proc. USENIX Annu. Tech. Conf.(Poster Session)*. Citeseer, 2008.

[31] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. ACM *Computing Surveys (CSUR)*, 47(3):42:1–42:36, 2015.

[32] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 193–204. ACM, 2010.

[33] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, 2002.

[34] J. Zhu, W. Dong, Z. Jiang, X. Shi, Z. Xiao, and X. Li. Improving the performance of hypervisor-based fault tolerance. In *IPDPS*, pages 1–10, 2010.

[35] Redis Database http://redis.io/

[36] D. E. Comer. Internetworking with TCP/IP: *Principles, Protocols, and Architecture*. Volume I. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1991.

[37] An introduction to the InfiniBand Architecture. http://buyya.com/superstorage/chap42.pdf.

[38] Mellanox Products: RDMA over Converged Ethernet (RoCE). http://www.mellanox.com/page/products_dyn?product_family=79.

[39] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. Aug. 2014.

[40] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the USENIX Annual Technical Conference (USNEX '14)*, June 2013.

[41] M. Poke and T. Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC ’15, 2015.

[42] Zookeeper. https://zookeeper.apache.org/.

[43] M. Burrows. The chubby lock service for loosely-coupled distributed system. In *Proceedings of the Seventh Symposium on Operating System Design and Implementation (OSDI ’06)*, pages 335-350, 2006.