

USING VM REPLICATION TO BUILD A
CONSISTENT, FAST HA SYSTEM

A Thesis for Master Research

Muyang He

Submitted in Partial Fulfilment of the
Requirements for the
Degree of Master of Computing

UNITEC INSTITUTE OF TECHNOLOGY
Department of Computer Science
New Zealand
February 2016

Abstract

Virtualization supplies a straightforward approach to high availability through iterative replications of virtual machine (VM) checkpoints that encapsulate the protected services. Unfortunately, traditional VM replication solutions suffer from deficiencies in either response latency or state recovery consistency, which constrains the adoption of VM replication in production. In this work, we redesign the typical consistency model and network architecture for virtual machine replication. In doing this, we extend the function of the secondary host to be the primary recipient of network requests so that the state of the primary VM (PVM) is retained by the secondary host in the form of network packets. Specifically, the secondary host is set for network redirection and packets recording. Should the primary host fail, the recorded packets are used to recreate the state on the secondary host. We name the system reverse replication of virtual machines (rRVM). Experiments demonstrate the simultaneous strong recovery consistency and low response latency of our real-time disaster recovery system. To be specific, we achieved 100% consistency without additional latency imposed.

Declaration

During the master study, I have two research works rRVM [1] and Cor-Honeypot [2] published from UCC and IEEE Cloud respectively. In which, rRVM [1] contribute to some chapters in this thesis.

Acknowledgement

The author would like to thank Prof. Shaoning Pang and Prof. Hossein Sarrafzadeh for the mentoring as well as the meticulous text review, and Denis for the in-depth knowledge transfer in networking. In particular, I want to thank Mr. Ding Lu and Mr. Yuan Zhang for their participation in peripheral utility development and bug shooting. They step forward in prototype demonstration independently with their knowledge in rRVM operation. Also, I want to present my gratitude to Mr. Lei Zhu and Mr. Xueyi Shi for their aid in data collection.

Contents

1	Introduction	6
1.1	Research Questions	7
1.2	Major Contributions	8
1.3	Thesis Organization	8
2	Research Project Development	9
2.1	Research Background and Context	9
2.2	Definitions	9
2.2.1	VM, VMM	9
2.2.2	Hardware virtualization, hardware-assisted virtualization, and paravirtualization	10
2.2.3	Xen	10
2.2.4	Domain 0, Domain U [3]	10
2.2.5	PVM and SVM	11
2.2.6	Unikernel	11
2.2.7	Shadow page table	11
2.2.8	Replication and migration	12
2.2.9	Stop and copy, pre-copy and post-copy	12
2.3	Related Research	13
2.3.1	Process migration	13
2.3.2	Virtual machine migration	14
2.3.3	VM replication, original work	17
2.3.4	Packets record & replay assisted VM replication	18
2.3.5	VM Replication on Unikernel	20
2.3.6	Heuristics	20
3	Research Objectives and Challenges	23
3.1	Research Challenges	23
3.1.1	Challenge of consistency	23
3.1.2	Trade-offs between consistency and latency	24

4	Proposed Approach and Implementation	26
4.1	Design overview	27
4.2	Research Methodologies	29
4.2.1	System bootstrap	29
4.2.2	Normal execution	30
4.2.3	Failover	34
4.3	Outcomes	35
4.3.1	Test environment	35
4.3.2	Latency overhead on interactive data flow	36
4.3.3	Throughput overhead on bulk data flow	37
4.3.4	Recovery consistency	38
4.3.5	Recovery time	39
4.4	Alternative research methods	39
4.4.1	No programming approaches for network redirection	40
4.4.2	Layer 3 network redirection	41
4.4.3	Network behavior differences in PV (paravirtualization) and HVM (Hardware VM)	41
4.4.4	The impact of non-determinisms	42
4.4.5	Potential optimizations	42
4.5	Application and Impact of Research Results	43
5	Conclusion and Future Works	44
6	Appendix	53

List of Figures

3.1	The failover semantics of different virtual machine replication mechanisms	24
4.1	The architecture of rRVM	27
4.2	System bootstrap	32
4.3	Different configurations for downstream and upstream intensive services	33
4.4	Queue expiration operation of SSA	34
4.5	Throughput of bulk traffic	37
4.6	Recovery consistency evaluation	38

List of Tables

4.1	Decouple of network and service activity, where 'P' and 'A' represent passive and active status, respectively.	28
4.2	Summary of the activities of rRVM components in different stages.	32
4.3	Average latency and timeout number of rRVM, Remus and primitive replication under different replication frequencies and network setups.	36
4.4	Recovery time	39

Chapter 1

Introduction

Modern organizations usually implement distributed servers [4] for better computing power, higher throughput, and modular separation. However, the evolved complexity is accompanied by the inherent fragility as a failure of a single node can cause the whole system to break down. The situation with distributed systems is such that the increased quantity of elements, the imperfect quality of each component, and the prolonged period of run time render a failure inevitable, which leads to the notion that services have to treat failure as routine rather than exceptional [5].

High availability (HA) services are those that are able to recover from failures automatically. Traditionally, HA services are achieved with a HA cluster [6], which directs requests exclusively to healthy nodes. Despite its merits (i.e., scalability [6] and module isolation), HA clusters are highly specialized to a particular hardware or software architecture because it has to be implemented in an ad-hoc manner. For instance, a load balancer [7] designed for the API layer can not be employed in the logic layer, master-slave database replication is useless for a shared nothing cache system [8], and a HA framework (clustering) developed on one operating system is hardly adaptable to another OS.

Service availability becomes a critical issue when a data center expands to a certain point where hardware failure becomes routine rather than an exception [5]. To minimize the economic loss, systems running on top of untrustable hardware should be able to automatically recover when encountering an hardware failure [9].

In view of the researchers that utilize virtual machine replication for high availability services, the merits of virtualization technology render its recent highlight by both academy and industry. It shapes a new breed of server infrastructure, Cloud [10], [11], with its easiness and flexibility for large-scale system management. Unlike physical machines, virtual machines are able

to perform provisioning, scaling up/down and migration purely [12] on a software level, which eliminates most of the rigid and complex hardware tasks for those everyday system operations. Moreover, computing power can be made more efficient with a finer granularity resource allocation, since the capacity of real machines can be sub-divided and reallocated catering to different workloads. The spectrum of virtualization technology expands to several areas including cyber security. For example, virtual machine can be used for honey pot with its capacity of fast provisioning of instances and privilege isolation [13]. In this work, we are focusing on the utilizations of virtualization in the area of high availability services.

1.1 Research Questions

Theoretically, a high availability service should be immune to hardware failures. One of the well adopted approaches to high availability is high availability cluster. In this solution, automatic disaster tolerance can be achieved by redundantly deployed elements of the system. Examples are, reverse proxy for Web server [14], [15], log replay mechanism and master-slave architecture of database [16], distributed file system maintaining multiple replicas [5], [17], [18] and software raid-1 like DRBD [19]. But systems of this kind have to be implemented in a ad-hoc manner.

Virtualization [3] provides a generalized solution to HA services [20]. Through the replication of VM states from a primary host, a secondary VM (SVM) is brought up to continue services whenever the primary host fails. Here, the VM states include contents stored in CPU registers, memory, disk, and other peripheral devices. Technically, a VM can encapsulate any kind of software and hardware, thus a VM replication-based HA system can be implemented once, then applied to a wide variety of architectures. Compared to those ad hoc HA solutions whose components can be used exclusively in the system where they are designed for, virtualization enables the replication of the whole state of a VM [20], which is a more generalized HA solution. However, existing VM replication mechanisms are generally not performant enough to be adopted in production.

This presented research focuses on virtual machine replication in back-end, where virtual machine live migration is a key topic, since existing virtual machine replication systems are mostly built on live migration predecessor. Note that neither HA for front-end application (e.g., [21]), nor HA in the form of task validity [22] is within the scope of this study.

This research proposes reverse replication of virtual machines (rRVM), a novel virtual machine replication system. As the name implies, several

properties of traditional systems are *reversed* in the proposed system: 1. In rRVM, network positions of the primary and secondary hosts are reversed compared to traditional solutions as clients requests are directed to the secondary host instead of the primary host; and 2. the secondary host can recreate the primary VM's state independently.

1.2 Major Contributions

The contributions of this research are as follows:

1. We solved the latency-consistency dilemma by decoupling network and service activities. Moreover, we developed rRVM that removes the superfluous blocking network buffer mentioned above and enables the protected system to execute with both strong recovery consistency and low response latency.
2. We proposed the state lost as a new measurement for evaluating inconsistency caused by failover. In performance evaluation, we summarized the state lost of rRVM and existing solutions at different parameters to test recovery consistency.

1.3 Thesis Organization

The rest of this thesis is organized as follow:

Chapter 2 gives the background of VM replication in which related works are discussed.

Chapter 3 presents the problem statement and the motivation of the proposed research.

Chapter 4 describes the overall design and detailed implementation of the rRVM.

Chapter 5 concludes the thesis with an outlook on future work.

Chapter 2

Research Project Development

2.1 Research Background and Context

State consistency is crucial to VM replication. States are client's data stored on servers, which includes log-in authentication, committed financial transactions, VoIP sessions, etc. In practical terminology, systems with *strong consistency* always have all replicas as one state in the client's view; whereas for systems with *weak consistency*, writes which have been acknowledged to clients might be lost after recovery. In practice, when a failure occurs, *strong consistency* is more desirable because the protected system can be seamlessly recovered to the state that existed immediately before the crash.

2.2 Definitions

2.2.1 VM, VMM

In cloud computing, VMM (Virtual Machine Monitor) or hypervisor is a software that provides virtualized hardware environment, as well as management interfaces for virtual machine (VM) instances [11]. In another word, the hypervisor is the operating system (OS) of VMs. Hypervisors are divided into two categories: Type 1 that runs directly on hardware, and type 2 that runs on a host OS. Regardless of the type of hypervisor, a virtual machine is presented to a guest OS, where the VM is almost identical to a real machine.

2.2.2 Hardware virtualization, hardware-assisted virtualization, and paravirtualization

Hardware virtualization refers to the technology that a hypervisor simulates all hardware mechanisms in software. This type of hypervisor can run an operating system unmodified. Qemu [23] [24] and BOCHS [25] are two mostly adopted hypervisors in this category. The technology is initially used for cross-compilation and cross-debugging, so the relatively weak performance [25] imposed by software simulation is acceptable. However, the performance becomes critical if virtualization is adopted on an industry scale system such as cloud [10].

Hardware-assisted virtualization is introduced to make virtualization more efficient. This requires support from processors such as Intel VT-x and AMD-V.

Note that in practice, QEMU is usually teamed with KVM [26], a kernel module that enables hardware-assisted virtualization, to compose the hypervisor in a cloud system. Hardware virtualization, being hardware-assisted or not, is referred to as full virtualization (FV).

Paravirtualization (PV) is introduced in Xen [3]. In contrast to FV, PV offloads some tasks (e.g., IO and network stack) in a guest to host OS, where those operations are light weighted compared to HV. Thus PV achieves nearly native performance, and it is especially useful in a system without processor virtualization extensions (i.e., Intel VT-x or AMD-V). However, it requires a modified guest operating system to address the alien virtualization environment underneath [3].

2.2.3 Xen

Besides PV, Xen supports FV as well. Like KVM, Xen includes QEMU to simulate the peripheral hardware and utilizes processor virtualization extensions to enable hardware-assisted virtualization. It is worth noting that the majority of VM replication researches are conducted based on Xen [20] [27] [28] [29].

2.2.4 Domain 0, Domain U [3]

In Xen's terminology, the operating systems managed by the hypervisor are called domains. Domain0 is a special domain that has direct access to the physical hardware. It is sometimes called as host operating system. One of the purposes of Domain0 is to provide drivers for all the other domains

running on virtual machines, called domainU. Thus, Domain0 is more privileged than domainUs in terms of protection rings. From administration's point of view, most of the tasks, e.g., creating and starting a virtual machine, are conducted through Xen utilities installed on domain0.

2.2.5 PVM and SVM

Throughout this thesis, PVM and SVM are referred to as primary virtual machine and secondary virtual machine respectively. Basically, the PVM is the virtual machine that is being migrated or replicated. In some works, it is also referred to as primary host, primary server or just primary. Inversely SVM is the replicated version or mirror of PVM running on a remote machine. In the HA context, the standby SVM is used to replace the PVM after a disaster event during which PVM is unavailable.

2.2.6 Unikernel

Unikernel [30] [31] is the future of Cloud computing. Adopting the LibOS approach, operating system kernel can be highly tailed by linking the binary of services to necessary operating system components (e.g., network stacks, drivers). As a result, the service binary can run directly on hypervisor. By reducing the additional layer of guest operating system, the system performance can be enhanced tremendously and its resource consumption can be made really low. VM replication based on unikernel is also ideal for long distance replication because of its small memory footprint which requires only minimal bandwidth for replication traffic.

2.2.7 Shadow page table

Basically, shadow page table is the Xen mechanism mapping the virtual physical address that is presented to a guest Oses and the real physical address. Shadow page table can be also used to track dirty pages in a hypervisor, the information can be used for VM replication [12]. It functions by introducing another page fault exception, and after the writing activity is tracked, the exception will return the control to the original memory operation routine. Another similar technique is called ghost buffer [32] being used for cache hit rate estimation [33], for the ghost buffer records all the cache hit actions as a middle layer.

2.2.8 Replication and migration

In this thesis, both virtual machine migration and replication are referred to the activities of transferring of the virtual machine state to a different physical machine or location. However, migration and replication have their intrinsic differences. Firstly, migration is a one-off activity while replication is usually long lasting process during which the migration action is being performed recurrently. Secondly, migration is an operative activity on the other hand replication is a fail-over resort preserving back-up(s) for the potential disaster. Moreover, replication is usually a feature embedded within a larger system which is executed automatically, while migration is manually carried out by the operator or system administrators. In certain circumstances migration could also be an automatic process. One an example is the use case of load balancing, in which VMM is responsible for moving VMs from one physical machine to the another with lower workload after a resource shortage is detected by VMM.

As in many other texts, synchronization and replication are used interchangeably in this article.

2.2.9 Stop and copy, pre-copy and post-copy

There are broadly three ways to migrate a virtual machine: stop and copy, pre-copy and post-copy. The mechanism of stop and copy is well self-explained, the PVM is stopped first, then the state of it is checkpointed and transferred to the SVM and the SVM is raised. The obvious problem of this kind of migration is that it doesn't support continuous service so except for the very early research on VM migration [34] this approach is not adopted. In post-copy mechanism [35], only the essential portion of memory in the VM is transferred first and then SVM is started. Other resources are copied from PVM as required. The requirement of the resources from the original machine is called residential dependency, which is the major reason why post copy mechanism can't be used for HA because if PVM fails during the process, all the resources are lost although it supports continuous service before and after the copy. On the other hand, The mechanism of pre-copy supports both continuous service with resilience to disaster. In this approach, all the memory state is transferred to SVM before it is started and PVM is not needed by the SVM after the migration completes. Since most of virtual machine synchronization systems are based on migration, the 3 types can also be applied to synchronization technologies. For example, the goal of Remus [20] built upon pre-copy mechanism is for HA while SnowFlock [35] is implemented as a middleware for the purpose of cluster computing. The

design and implementation detail of the two systems are discussed in the following sections.

2.3 Related Research

2.3.1 Process migration

Process migration [36] [37] is the transfer of a sufficient amount of a process' state from one node to another during when the process is executing. The ability to migrate process offers many benefits, especially in a distributed system. It does not only give more power to the system to control the workloads among the hosts but also optimizes the speed by reducing network traffic. Besides, when the host on which a process runs failed, both the persistent data and the complete process state should have been transferred to the backup node so that the process can be started on another node, and the clients will not notice any downtime, to keep the services available at all time. [38]

There are four basic process migration algorithms exist in process migration. They are: Eager copy, Pre-copy, Lazy copy, and Flushing. Eager copy algorithm requires the entire state of the process has to be transferred before the process execution resumed on the destination node, e.g., MOSIX [39]. On the contract, lazy copy algorithm would only transfer as minimal amount of data as possible to restart the process on the destination node rather than all of them. The pre-copy algorithm is almost similar to the eager copy algorithm. The difference is that pre-copy algorithm operates migration while the source process is still executing. When reaching a certain point, the process on the source machine would be suspended and all other states would be sent to the destination machine [40] [41]. Finally, the flushing algorithm trades off the increased overhead of the state copy soperation for the elimination of residual dependencies that exists in the lazy copy algorithm. This algorithm is initially introduced in Sprite OS [42].

However, there are many issues that would happen during migration. For instance, residual dependencies are some part of the process state that left on the source host or sometimes on an intermediate host while migrating. They typically occur when implementing two techniques: either using Copy-On-Reference (e.g., Mach [43]), or as a mean for achieving transparency in communication. Note that Copy-On-Reference is a technique that copies the state whenever the state is read. Residual dependencies will affect reliability because a migrated remote process will always depend on its home node [44]. Shared memory is another problematic part [45].

Most systems disallowed migration of processes using shared memory (e.g., Sprite [42]), while for others that supported both shared memory and migration chose not to provide transparent access to shared memory after migration (e.g. Locus [46]).

2.3.2 Virtual machine migration

Virtual machine migration usually is performed for the purpose of system maintenance, e.g. hardware replacing, software upgrading or load balancing. Generally speaking, the migration of a virtual machine involves two operations: 1. check pointing; 2. copying the checkpointed image to a remote location. In this section, fundamental researches on live migration based on three major kinds of hypervisors (Xen, ESX Serve and KVM) are discussed.

Live migration on Xen

In [12], a live migration mechanism implemented based on Xen is presented. It is the technical basis for some of the virtual machine synchronizing systems discussed in this text. A hybrid approach involving pre-copy and stop-and-copy is adopted in this research. The migration of running, in service systems, implies the needs for both minimized downtime and the consistent server state including memory, network connection, and disk.

The inevitable suspension of the system is caused by the existence of the so-called "writable working set (WWS)". Both of writable and working set refers to the recently updated contents [47] [48] [49]. In this case, the memory region of WWS is characterized by its very high modifying frequency, which utilizes previous pre-copy iterations so often that substantial overhead is imposed. And the WWS is identified by utilizing the shadow page table. Moreover, if the updating speed is even faster than a threshold the possibly fastest transferring speed, the migration will not be able to complete because of the newly dirtied pages. So stop-and-copy is triggered in this scenario until WWS is fully transferred to SVM. The inconsistency between PVM and SVM is eliminated in this way.

The migration mechanism is divided broadly into two phases accordingly: Firstly, all pages are copied in a pre-copy manner, and this phase is called "pre-migration" [12]. In pre-migration, pages are copied by several rounds. In each round, incremental checkpointing [50] is performed thus only dirty pages newly updated after previous rounds are copied. Again, dirty pages are identified by using Xen's shadow page table (§2.2.7). Secondly, the active memory area or WWS is copied in "stop-and-copy" man-

ner [12]. By copying WWS only in this phase, the system downtime is substantially shortened compared to that with pure stop-and-copy. During "stop-and-copy phase", the system is completely unavailable so how to shorten the duration of this phase is one of the primary goals for later researchers (§2.3.6).

To achieve the continuous service, the network connection should not be reset during or after migration, and in order to be adaptive to various of network environments, 3 different optimizations are applied. In an environment that allows broadcast ARP (Address Resolution Protocol), an unsolicited ARP is generated to inform peers in a local network the new location of the system. In broadcast forbidden network, the system can either send ARP requests to entries in current system's ARP cache or passively waits for switch finding the new location automatically. The unavoidable packets lost is spontaneously dealt by upper protocols. These techniques for maintaining the network connection are kept unchanged in the following virtual machine synchronization systems like Remus [20] and Colo [27].

Unlike its VM synchronization derivatives, the burden of disk replication is on NAS (Network-attached storage) or alike specialized storage systems. This design decision imposes the most noticeable drawback in this research since the mechanism is not compatible to quite a few data centers without NAS facilities. In later research like Remus [20], this problem is well solved by taking disk replication into account.

One of the biggest challenges in this research is to determine the value of bandwidth limit for migration traffic: the high value of the limitation has the impact on the running service while a low limit imposes a long time for the WWS copy [20], which as afore-discussed is virtually the downtime of the system during the migration. Although they failed to point out that in real network topology, there is usually a dedicated LAN for migration traffic and optimized bandwidth limitation is still required when the migration is applied to WAN or a dedicated LAN is not available. In this research, a preliminary optimization is proposed, in which the limit value is determined by dirty page rate in previous rounds.

There are two implementation approaches for VM migration, i.e. self-migration and managed migration. The difference between them is whether the mechanism is implemented in domain U or domain 0. In real world Only managed migration implemented in domain 0 is used because its simplicity and less intrusion it imposes to the guest system.

To evaluate the performance of the live migration [12], the system is tested with 3 kinds of servers with different requirements, i.e. a web server with a heavy load; an SPECweb99 server requiring QoS and a Quake3 server

which demands for real-time responses. In fact, the reasonable system performance is one of the reasons that this research becomes the fundamental of numerous coming ones.

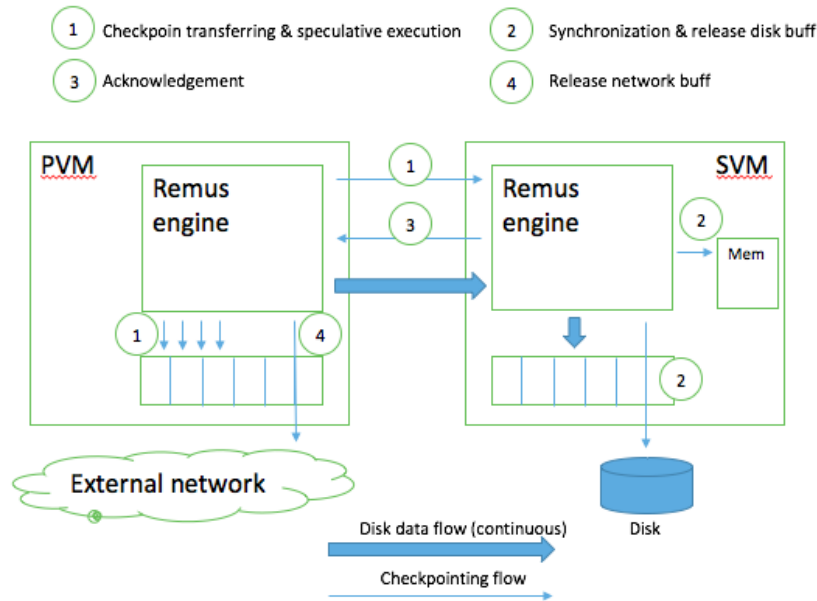
Live migration on ESX Server

In [51], a similar live migration system called VMotion based on VMWare ESX Server [52] is proposed. There are two differences between VMotion and [12]: first, VMotion does not use shared storage, so it copies memory state as well as storage content during the migration; secondly, it takes a post-copy approach [51] so a residual dependency will exist temporary after the secondary VM is brought up. The migration takes three steps also: 1. Iteratively copy: all the memory is copied to the destination in rounds while the primary VM is still running. This step is almost the same as the first step of [12]. One thing to be noticed in this step is the dirty pages are tracked also in the same way with the shadow page table used in [12], i.e. by marking the pages read-only. 2. Converging: Suspend the primary virtual machine and copy the "non-memory state" when the dirty memory is modifying too fast that no progression is forwarded. This step is equivalent to "stop and copy" 3. Resume the secondary VM and copy the remaining resources [53].

Live migration on KVM

The Live migration on KVM is similar to that on Xen and VMWare except for that in KVM live migration, a consistent images directory is necessary [26]. To carry out the live migration, firstly a new virtual machine with -incoming TCP parameter on destination host server is created. This new virtual machine attaches the source virtual machine image which locates in a shared storage with dirty page logging enabled; Secondly, the PVM memory page is locked and whole memory content is transferred to SVM. Moreover, a temporary memory zone is enabled to keep PVM running. Thirdly new temporary memory zone records incremental memory data and iteratively transfer them to SVM, then these pages were written by the guest. Finally, PVM is suspended and transfer the remaining data and resume the SVM on the destination host. Throughout the process, virsh is used to carry out the operations [26].

2.3.3 VM replication, original work



In [20], a virtual machine replication system called Remus is proposed. Compared to its migration predecessor [12], the new system [20] has 3 advantages.

Firstly, the performance is optimized so that the sustainable iterative migrations is feasible. Secondly, in addition to WWS outward network buffering is added for the maintenance of consistency. Last but most importantly, the state of secondary storage is taken into account so expensive and specialized hardware like NAS is no longer compulsory. This improvement is of great value since it largely enhances the generality of the system. The details of these advantages are discussed below.

As aforementioned, the performance demand of Remus is higher than that of live migration systems. This additional demand is imposed by the reiterative and real-time manner in which the VM state is replicated. In [20], the researchers find out that the latency is mostly incurred by the communication overhead required by domain U suspend/resume between migration process, xend and xenstore, the two core components of Xen. So by creating a direct channel through which migration process and domain U can communicate directly, the interprocess communication overhead is diminished and the latency is reduced by two orders of magnitude.

The most outstanding optimization which by and large characterizes this research is the asynchronizing of the VM state copying process. Basically speaking, rather than stop-and-copy the WWS in the last round, the system

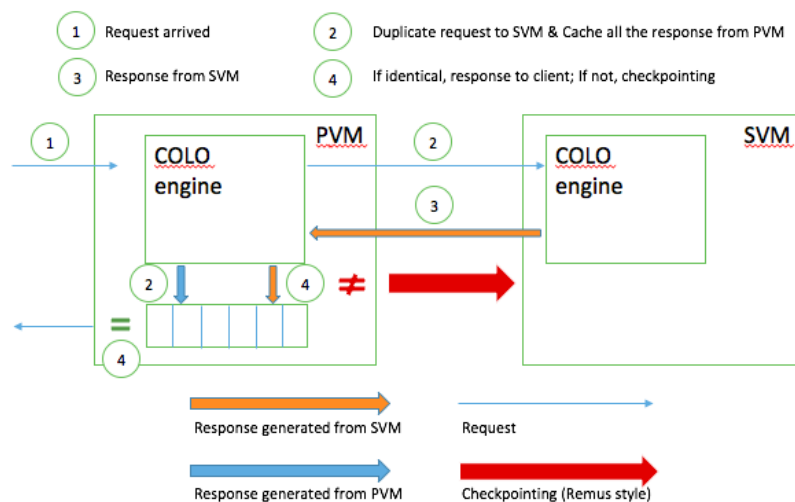
keeps on running speculatively or asynchronously during the process. One problem in asynchronized scenario is that the consistency is periodically undermined by the speculative execution of PVM. To solve this problem, a buffer layer is added at PVM. The buffer is used to hold all the outbound packets generated by the speculative execution which in turn are released to clients after a checkpoint acknowledgment is received from SVM. As a result, despite the inconsistency of internal states of PVM and SVM, they are consistent from the client point of view.

When it comes to secondary storage state the inconsistency between first and secondary storage rises. This inconsistency is incurred by the fact that disk state is copied as a stream in a continuous manner while checkpoints of memory are transferred in epochs. To maintain the consistency, another layer of the buffer is added at SVM, which holds all the disk modifications received from PVM. Similar to network buffer, the disk buffer is released when an epoch arrives and a real disk writing is performed afterward.

Although the system is well tuned for virtual machine synchronization, there are two apparent drawbacks performance-wise. First, the network cache holding all outbound traffic during speculative execution incurs noticeable latency; Secondly, the considerable bandwidth occupied by the continuous copying of VM state is a noticeable overhead. And the high epoch frequency in Remus exacerbates both of these two problems.

2.3.4 Packets record & replay assisted VM replication

COLO



Based on Remus, COLO [27], or so-called coarse-grained lock-stepping is proposed to reduce the epoch frequency. Apart from the lock stepping approach discussed earlier, COLO [27] is a lock-stepping VM replication mechanism on the network level. This work makes an assumption that consistent virtual machines always generate identical network output for the same network input. Based on the theory, COLO's mechanism is as following. Firstly, whenever an external network input arrives, PVM a) processes the packet and buffer the output and b) relays it to SVM; Secondly on SVM, same executable processes the packet and send its output to PVM instead of external network; Thirdly, PVM compares the SVM's response to that it buffered in the previous step; Lastly, COLO will notice an inconsistency if there is discrepancy in the two versions of output (from SVM and PVM) and invoke a Remus [20] replication.

COLO is scalable in most real world environments, notably when the protected VM interacts with a backends database that may or may not be protected by the VM replication system. It avoids duplicated output from SVM to the backend since SVM only talk to PVM and treats the database as the external system as well.

However, COLO has numeric flaws. 1) Although TCP protocol is modified to generate deterministic packets, operations at the non-network level are often non-deterministic. One example is the OS scheduling of multi-thread application; 2) Despite one of the goals of COLO is to reduce latency, it introduces new source of latency by waiting for response from SVM synchronizedly. And the latency of the whole system will be increased linearly increase linearly with the node number, and 3) consistent virtual machine states are neither sufficient nor necessary condition to identical network outputs since network responses are largely determined by the implementation of certain application. So the states of SVM could be long corrupted before COLO can detect inconsistency. Thus, protected by COLO, it is very hard for systems to survive in an instant failure.

LLM

LLM [54] is another work towards reducing the frequencies of resource consuming state replication. LLM compensates the deducted checkpoint replication with the replication of the network. In a failover event, LLM can rebuild the missing VM state due to the low state replication frequency by replaying those packets to the secondary VM. In the primary host, LLM records network packets in kernel space and then copy them to a user space program. After LLM completes the lost information during the copy process, it buffers the packets. Then LLM replicates the packet's buffer to sec-

ondary host in rounds. The packets replication is conducted during the interval of state replication to avoid network congestion.

However, the authors of this work failed to recognize that the Remus network output buffer is the crucial component to maintaining strong consistency by blocking the last gasp packets as well as the uncommitted VM state. Reasoning the buffer imposes significant response latency in the low-frequency replication setting of LLV, they simply discard the buffer. Consequentially, without any supplement means, LLM is weakened inevitably inconsistency level.

2.3.5 VM Replication on Unikernel

Tardigrade [55] is a virtual machine replication system using a Unikernel guest, Drawbridge. The small footprint of VM guest gives Tardigrade an obvious performance advantage over Remus even though Tardigrade borrows Remus replication policies (e.g., it too adopts a network buffer to block the last gasp packets). Besides the underlaying software stack, there are several differences between Tardigrade and its Xen based VM live migration & replicatoin counterpart. a) Tardigrade enables a multiple backups configuration, another benefit offered by the small footprint of Unikernel; b) Tardigrade uses record and replay of ABI (narrow binary interface) calls for disk replication while Remus replicate the actual content and c) Tardigrade enforces independent networks for business and replication traffic while Remus supports both independent and shared network layout. Despite the advantage of Unikernel, Tardigrade has some flaws compared to Remus. Firstly, unlike Remus, Tardigrade can not maintain a persistent TCP connection in failover events because of a problematic design in Bascule, that is, applying the socket on host rather than guest so it is not visible by the checkpointer; Secondly, some of the nondeterministic ABIs, that are used in disk replication, imposes more complexities in the system implementation.

2.3.6 Heuristics

There are numerous drawbacks in the original work of VM live migration and synchronization, and some of them are well fixed by the following researches. Although the majority of the optimization is based on live migration, since live migration is the basic element of virtual machine synchronization, the later can directly benefit from optimization based on it. For example, the current version of real world Remus [20] adopts a similar way described in [47] for disk synchronization and abandon its original design

discussed in the paper [20]. In this section, the major optimizations are discussed and categorized according to the policy despite the different mechanisms they seemly adopt.

Performance

Some researches aim to shorten the suspending time or total migration time by performance optimization. In [56], the time and space efficiency of checkpointing is further enhanced by excluding the page cache in a snapshot. The portion of memory could be safely omitted because the content already resides in the secondary storage and a page-to-disk block map is maintained to store the relation information of page cache and the associated disk location (ADL) [33] [56].

Since the research of [56] is purely on checkpointing, it is not applicable to live migration until [57], in which the technology introduced by [56] is integrated to a shared storage, and because only the page-to-disk block map is needed to be transferred rather than the real content of page cache, it is reasonable to assume that the speed of migration could be dramatically enhanced in comparison to full migration techniques even though the value is not explicitly given by the paper.

[58]

Long distance (WAN) replication

When the synchronization is applied in a wide-area network (WAN) environment, new challenges arise: 1. limited or heterogeneous bandwidth and latency; 2. huge data volume involved in storage relocation because of a lack of shared storage in the environment as ARP is a LAN based protocol; 3. service continuity on WAN is hard to maintain, since the IP of the VM is supposed to change as the physical location and the unsolicited ARP adopted by live migration is not applicable in WAN. As a result, most researches in this area mainly solve these three problems.

XvMotion [59] is based on the core part of live migration of VMWare ESX Serve, VMotionq [51] (§2.3.2). It uses I/O mirroring, that replicates all I/O activities from primary to secondary, developed in [59] to solve the storage relocation problem and introduces a few optimizations adjusting to the unstable WAN.

One of the optimizations is a new transport protocol called stream transport framework [59]. The protocol is different from TCP by opening multiple connections for avoiding HOL (head of line). While stream transport framework can saturate the bandwidth better than TCP, the authors fail to

evaluate the impact of the migration activity impose to other live services. Other optimizations include how to scheduling among first-time disk transferring, memory state synchronization, and I/O mirroring. Moreover, XvMotion throttle the VM after detection of a lacking of bandwidth in order to reduce the number of dirty pages and eventually, the volume of data being transferred. Congestion control mechanism is also introduced in the research for the same purpose because standard TCP congestion control can not be applied to the system.

However, in terms of service continuity on WAN, the work [59] just mentions some potential solutions without pointing out the solution it adopts. One of those solutions is OpenFlow [60], which is the most intuitive solution if the purpose is to redirect network packets on WLAN. OpenFlow [60] is adopted by [61] as well. Other solutions include IP mobility [62]; Overlay network [63] and dynamic DNS [64], all of which enable WAN packets redirection.

in [65], a three-phase, fractional, hybrid migration solution for memory and storage are proposed to achieve adaptive and responsive WAN-wide VM migration. The complete migration consists three phases: pre-copy, stop-and-copy, and post-copy, where M percent of memory and S percent of storage are transferred within the first two phases. In the pre-copy phase, the system started with storage transfer, when S storage is transferred, the available bandwidth is then allocated to move M memory. When further pre-copy iterations are not expected to improve the memory transfer (i.e., page dirtying rate is higher than transmission rate), the system enter the stop-and-copy phase until M memory an S storage are fully migrated. At the pos-copy phase, the remaining memory is migrated and then the storage.

Chapter 3

Research Objectives and Challenges

3.1 Research Challenges

Remus [20] is the groundbreaking system of this kind. To perform the expensive virtual machine replication consecutively, Remus adopts a few heuristics such as an improved memory copying mechanism, a direct communication channel between the migration process and domU (i.e., guest operating system), and speculative execution of the primary virtual machine. However, Remus has not solved the fundamental difficulties of VM replication: high response latency and weak recovery consistency.

In general, solutions to these problems offer tradeoffs between the two objectives. Remus enhances consistency by entailing a blocking network buffer, but this incurs a high response latency as a side effect. Another mainstream VM replication-based HA system, COLO [27] is tuned for a lower latency, but consistency is compromised. Hence, the difficulties of high/low latency and weak/strong consistency has never been satisfactorily solved.

3.1.1 Challenge of consistency

Typical VM replication systems only support weak consistency, in which the pre-copy stage [12] [51] of the VM migration is performed in epochs. An epoch comprises three consecutive operations: (a) executing the virtual machine, (b) checkpointing the state, and (c) replicating the state to a remote host. The duration of an epoch is configurable and normally has a predefined value of hundreds of milliseconds. One problem with these systems is that when a failure occurs in an epoch, the secondary VM can only

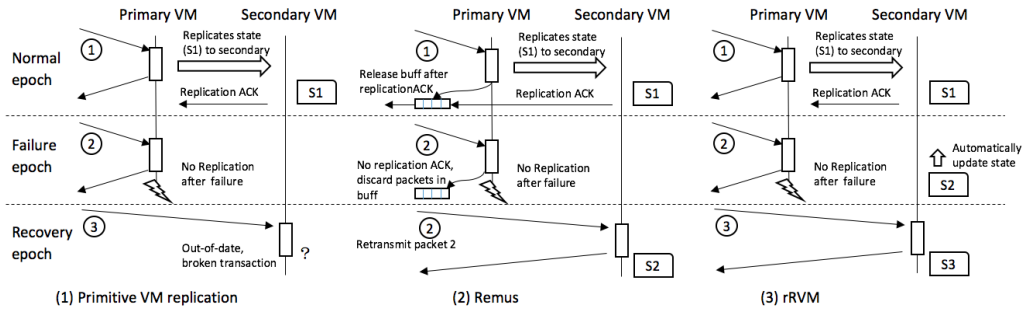


Figure 3.1: The failover semantics of different virtual machine replication mechanisms

be brought up in an out-of-date state, which reflects the completion of the previous epoch.

Figure 3.1-1 illustrates this states inconsistency described above. In a normal epoch, a client sends requests to the primary VM and the packets updates the VM states accordingly. Then the HA system replicates the primary VM states to a secondary host. In a failure epoch, the primary VM state (S2) before the crash is lost because the failure of the primary host interrupts the scheduled replication in the HA system. In a recovery epoch, the HA system brings up a secondary VM to a state (S1) obtained through the replication in a normal epoch, yet leaves the client view as the state (S2) acknowledged by the packet (2) in the failure epoch. Consequently, whenever the client sends the request packet (3) after the failover, a transaction error occurs. This scenarios was described as the *output commit problem* in [66]. We henceforth call the problematic acknowledgment of packet (2) in the failure epoch as *last gasp packets*.

3.1.2 Trade-offs between consistency and latency

Remus adopts atomization to achieve strong consistency [67]. Figure 3.1-2 depicts a UML sequence diagram of Remus, in which the network response and VM state replication are manipulated as one atomic operation by adding in a network buffer. The buffer blocks the output from the protected system until a replication is acknowledged by the secondary host. The *last gasp packets* thus are discarded whenever VM state replication is interrupted. Theoretically, this design should maintain a consistent client’s view of the system state. This approach, however, has three significant drawbacks. Firstly, the server state is rolled back after the failure, which incurs packets retransmissions and additional recovery time. Secondly, it does not achieve strong consistency for low-frequency replication (i.e., less

than five times per second or 200 milliseconds epoch duration). In such cases, there will be state lost after recovery because that the network buffer is only active in part of an epoch, that is, from checkpointing start to migration acknowledgment. As a result, it does not block *last gasp packets* for the rest of the epoch and does not guarantee state recovery consistency (§4.3.4). Thirdly, the blocking network buffer imposes significant latency overhead in protecting the failure-free system. This is the most important drawback addressed in rRVM.

COLO [27] reduces response latency by the lock-step approach in which two VMs are running in parallel. To determine if the states of the two VMs are consistent, COLO compares their network output by leveraging the TCP ordered data transfer. When an inconsistency is detected, COLO triggers a Remus VM replication that overwrites the secondary VM state. This design minimizes the number of replications and latency overhead is reduced. However, the identical output from the two VMs does not guarantee the consistency of their states, which follows that inconsistency cannot always be detected. In the event of a failure, the primary state might have been lost on the secondary host because of the absence of the necessary replication. Therefore, the low latency of COLO is gained at the price of recovery consistency.

Chapter 4

Proposed Approach and Implementation

In this work, we propose reverse replication of virtual machines (rRVM) as an advanced solution for real-time disaster recovery. In contrast to the atomization and lock-step approaches, rRVM has all completed network interactions committed during normal execution and incomplete ones replayed during recovery. Figure 3.1-3 shows the working flow of rRVM. The distinguishing feature of our system is that, during failover, the secondary VM is capable of updating its state independently, which enables the VM to recover to a state 100% consistent to that existed immediately before the crash.

We also introduce the concept of decoupling network and service activity, which leads to three benefits. First, it supports strong consistency. As the primary recipient of network traffic (a.k.a., network active), the secondary host can intercept and record network packets. During failover, the logged packets are replayed to roll forward the state of the secondary VM until consistent to that of the primary VM. Second, it imposes nearly no latency overhead. In theory, rRVM is slower than primitive VM replication since the redirection and interception of packets required by the decoupling have a slight latency cost. However, the difference is barely measurable as both operations are asynchronous (non-blocking). Third, the network architecture supports flexible deployment of VMs. In rRVM, it is easy to run multiple primary and secondary VM instances on the same host and allocate hardware resources for VMs as required. The design of decoupling network and service activity is discussed in detailed in the next section.

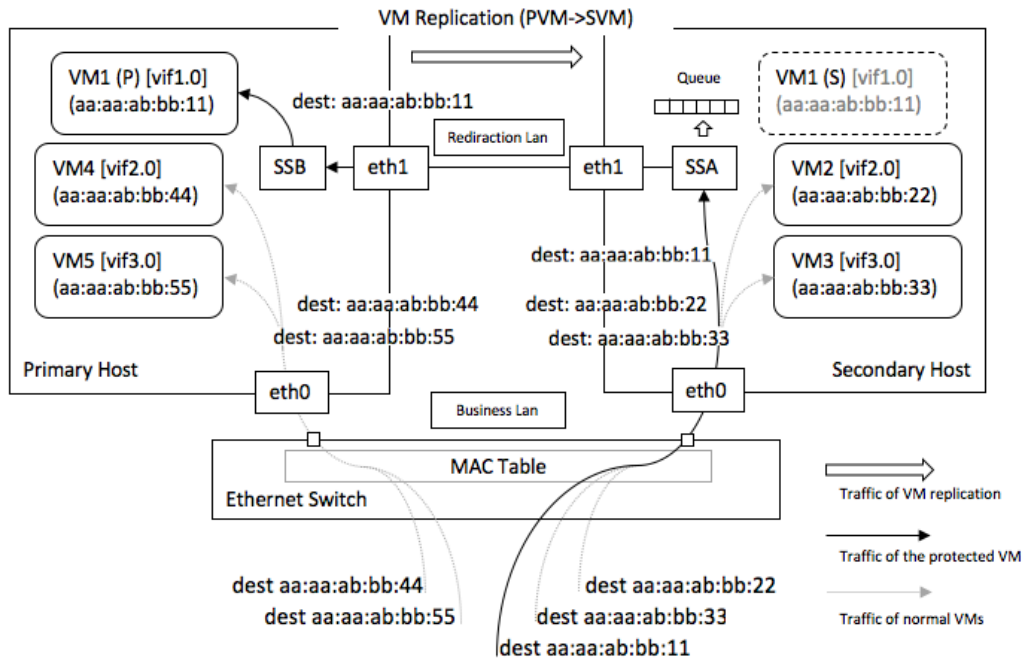


Figure 4.1: The architecture of rRVM

4.1 Design overview

Figure 4.1 shows the architecture of rRVM. In our setting, we enlist two separated LANs. The primary LAN, composed of all hosts for physical network interfaces eth0, deals with all external network traffics destined to the protected VMs. The redirection LAN, composed of eth1 interfaces of the hosts, is dedicated to network redirection. For any given VMs protected by rRVM, two hosts are involved in normal execution: the primary host runs the VM executing the service binary and the secondary host receives and redirects network packets.

rRVM uses a customized active-and-passive configuration as a failover model. In this setting, the protected services run on the primary VM, while the secondary VM is a dormant recipient, receiving system states from the primary. When the system fails over, the secondary VM is brought up with the state received. An active-and-passive configuration enlists only one running instance that consumes resources, thus this configuration is usually more efficient than its active-and-active counterparts that have two instances running in parallel. As mentioned before, COLO adopts the active-and-active configuration. In our case, we renovate the typical active-and-passive configuration by decoupling network and service activities. We define a system as service active if it executes the application binary and

network active if it is the principal recipient of business packets. Table 4.1 gives the results of our design, where the service and network are simultaneously active or passive for those hosts with a standard active-and-passive configuration. In contrast, during the normal execution of rRVM, a host can be either network or service active. As the result of the decoupling, during normal execution, the primary host is service active (network inactive), and the secondary host is network active (service inactive). It is worth noting that a VM shares the same property regarding service activity as its host.

System type:	Traditional systems		rRVM	
	Network	Service	Network	Service
Primary	A	A	P	A
Secondary	P	P	A	P
Pre-initialized	N/A	N/A	P	P
Failed	P	P	P	P
Promoted	N/A	N/A	A	A

Table 4.1: Decouple of network and service activity, where ‘P’ and ‘A’ represent passive and active status, respectively.

In general, normal execution of rRVM consists of two major network operations: packets redirection and interception. Considering that the primary VM is network inactive, and not accessible from the business LAN, we redirect network traffic to the primary VM through the secondary host and redirection LAN, as depicted in Figure 4.1). rRVM achieves this redirection by (a) setting up a layer two network proxy on the secondary host, which relays the packets from the host’s eth0 to eth1, and (b) creating another proxy on the primary host that interconnects the host’s eth1 to the virtual network interface of the protected VM. In this article, we also refer to these two proxies as software switches SSA and SSB respectively. During packets interception, the secondary host records the packets in a first-in-first-out (FIFO) queue group to retain the newest system state in the form of network packets.

In rRVM, network redirection is the primary cause of the extra latency added to the overall round-trip time (RTT) between clients and the protected VM since the other network operation, that is, non-blocking packets interception, incurs nearly zero latency [68]. The amount of latency added here is equals to the LAN RTT between the primary and secondary hosts, (i.e., less than one millisecond in standard data centers) [69] [70]. Despite the minimal latency penalty, rRVM maintains close to 0% state lost during failover according to our testing result.

As depicted in Figure 4.1, rRVM checkpoints the VM states and copies them from the primary to the secondary host periodically. The state divergence occurs for every epoch when the primary VM resumes from the checkpointing suspension. For state recovery consistency, rRVM reconciles this divergence by replaying the recorded packets in the queue group to the secondary VM during the failover.

Unlike the standard active-and-passive configuration, the failover mechanism of rRVM covers both primary and secondary hosts. The reason behind this is that our network architecture incorporates the secondary host into the critical path, and its failure causes the breakdown of the whole system. Generally speaking, in the case of the failure of either host, we promote the healthy one to be both service and network active (defined in Table 4.1). The failover model of rRVM is discussed in detail in Section 4.2.3. In rRVM speak, a failure of the primary host is called a service activity failure, and a failure of the secondary host is referred to as a network activity failure responsively.

We program all of the above components and operations in C as two user space ELF executables, which gives the system a better compatibility and flexibility. In the following sections, we discuss rRVM implementation in three stages: system initialization, normal execution, and failover.

4.2 Research Methodologies

4.2.1 System bootstrap

Figure 4.2 introduces rRVM initialization. The first step is a live migration that sets up the desired architecture given in Figure 4.1. In doing that, we add a physical machine for the target VM being protected and live migrate the VM towards it. The host stays as pre-initialized (defined in Table 4.1) throughout the migration process. Next, we set the added host as the primary host and the pre-existing one as the secondary host. In this way, the primary host is responsible for service and the secondary host is ready for network redirection. It is worth noting that owing to the live migration, updating the MAC table in a physical switch is not required. The reason is that the secondary host remains network active even with the VM removed by the migration. Because the physical switch in the LAN does not need to update the physical location of the protected VM; hence, no additional latency is caused here.

Other operations for system initialization include (a) stops the network traffic destined to the primary VM from the business LAN, so that the traffic

has to travel through the secondary host, (b) start software switches on the primary and secondary hosts, and set all network interfaces involved to promiscuous mode, and (c) initiate the replication and enter into a normal execution stage (§4.2.2).

4.2.2 Normal execution

In this section we discuss the three major components of rRVM. All components are initialized with the process discussed in the previous section, and they remain functioning in the initial configuration until a failure of either the primary or secondary host occurs.

Network redirection

When the system initializes, the primary VM is set as network passive. As such, network packets are relayed to the primary VM through the secondary host and the two software switches (as defined in §4.1). Three steps are involved in this process.

- Step 1: SSA opens two packet sockets [71] listening to eth0 and eth1 on the secondary host. This redirects the network traffic destined to the protected VM from eth0 of the secondary host to the redirection LAN, as depicted in Figure 4.1.
- Step 2: SSB on the primary host interconnects the backend virtual network interface vif1.0 of the primary VM and the eth1 of the primary host. This redirects packets received from Step 1 to the protected VM. Note that when failover occurs, SSB activates the eth0 to promote the primary host as network active, as described in Section §4.2.3.
- Step 3: SSB blocks the direct route to the primary VM by dropping address resolution protocol (ARP) packets going to the primary VM via the eth0. This ensures that future packets destined to the primary VM go through the secondary host.

Note that SSA and SSB are full-duplex and all corresponding outbound traffic is redirected in the same route back to clients. Further, network traffic for collocated unprotected VMs is not affected by our software switches since they operate only on the protected VM. This network redirection is transparent to the protected VM since neither software switches tampers the packets, except for the necessary low-level operations like TCP checksum and segmentation as described above.

Packets recording and replay

As mentioned in Section 4.1, SSA records all the network packets destined to the protected VM in a group of FIFO queues on the secondary host. When the primary host fails, SSA replays the recorded network packets to the protected VM and updates correspondingly the server state. At the normal execution stage, the recording of packets works in a non-blocking mode, thus no latency is involved. When failover occurs, the SSA switches to blocking mode, which stalls all newly incoming traffic. Only after the replay is finished does SSA release all blocked packets and becomes inactive. As a result, network packets are ordered in a consistent sequence during the failover. The detail of mode switch of SSA is discussed in Section 4.2.3.

The FIFO queues include those dynamically created for established TCP connections and a static one shared by all UDP and other packets. The two types of queues can be differentiated based on a dequeue operation, which works as the first step of packets replay. We implement the first category dequeue operation to simulate standard TCP interaction to maintain established TCP connections, where a packet is dequeued only after the response from the protected VM is received as the right SEQ and ACK numbers [72]. The second dequeue operation simply flushes UDP and other packets in one batch and does not throttle packets replay because the packets here belong to stateless connections.

In rRVM, the lost states of a failed VM acknowledged by *last gasp packets* (§3.1.1) can be reproduced with the packets recorded in the queues on the secondary host so that they cause no problem to service continuity. However, *last gasp packets* (§3.1.1) may cause a breakdown in the TCP connection when they are ACKs in a TCP 3-way handshake. Specifically, partially established TCP connections can not be recreated even with the packets replay, because the secondary host is not aware of the sequence number agreed previously by the primary VM and client. In facing this problem, we simply discard queues with only SYN packet, recognizing that partially established connections are not essential server states. This design might cause connection failure after recovery for servers based on short-session connections. The impact in practice, however, is negligible in that long session connections are more widely adopted (e.g., keep-alive headers of HTTP [73]) normally for reducing the RTTs caused by hand shakes.

State replication and packets expiration

For state replication, same as with Remus, memory is checkpointed and copied in an iterative manner through VM live migration [12] and disk state

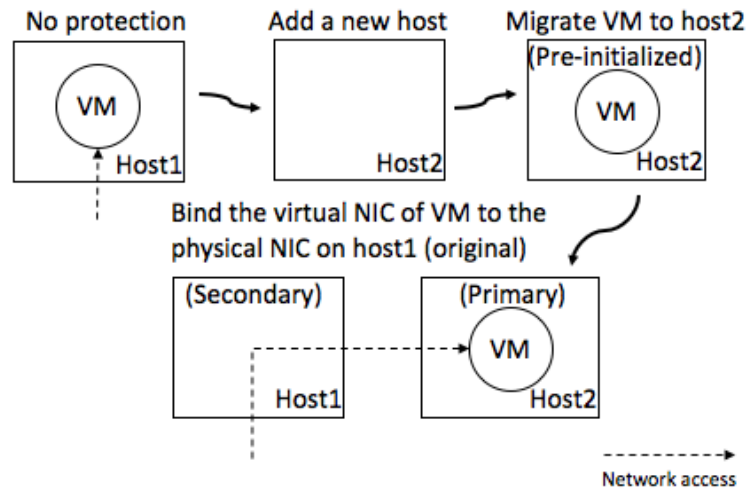


Figure 4.2: System bootstrap

Stages:	Initialization	Normal execution	Network failover	Service failover
Bootstrap	live migrates VM to the new primary host blocks instream to VM sets NIC to promiscuous starts SSA, SSB starts VM replication	N/A	N/A	N/A
SSA (on secondary)	N/A	bridges eth0, eth1 records instream queue expiration	N/A	disconnects eth0, eth1 blocks new instream bring up VM replays queued packets unblocks instream
SSB (on primary)	N/A	bridges eth1, vif1.0 drops ARP to VM records the latest ACK	bridge eth0 to vif1.0 sends an artificial MAC masquerading	N/A

Table 4.2: Summary of the activities of rRVM components in different stages.

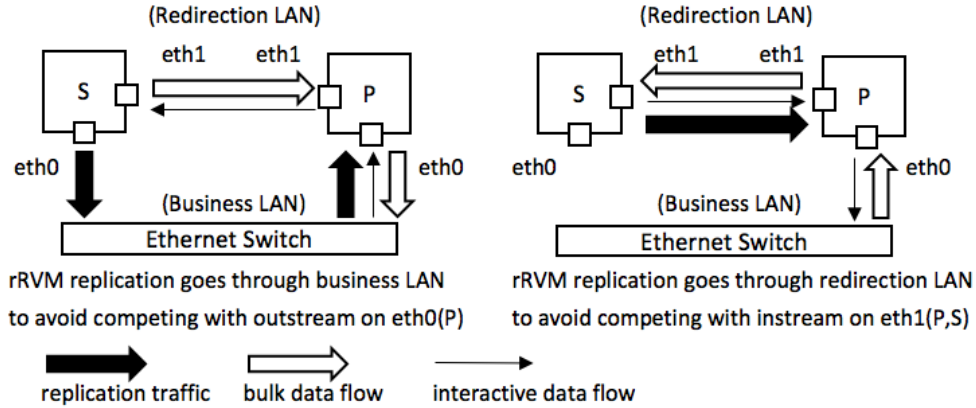


Figure 4.3: Different configurations for downstream and upstream intensive services

is replicated using I/O mirroring [19]. The replication traffic can be configured to go selectively through the redirection LAN or the business LAN. This ability makes rRVM adaptive to services with different traffic patterns. As demonstrated in Figure 4.3, to avoid competition for bandwidth on physical network interfaces for both primary and secondary hosts, replication traffic always goes in a direction opposite to bulk data flows with two different configurations. We set the default path of rRVM replication to the business LAN through eth0s of both hosts because most cloud applications are upstream-intensive. In the very few cases that services are both downstream and upstream intensive, competition for bandwidth is inevitable. The performance impact on protected VM imposed by this competition is examined in Section 4.3.3.

For packets expiration, packets seen by the primary VM and also reflected in the latest checkpoint are removed from queues to keep the memory consumption small. This operation is depicted in Figure 4.4. Firstly, SSB records, in the primary host, the most recent acknowledge numbers extracted from response packets in every network connection. Secondly, the recorded ACKs are transmitted to SSA in the secondary host along with the replication traffic. Thirdly, when monitoring this traffic, SSA triggers expiration of the queue entries with a less sequence number than the ACKs. Note that all the packets in the queue group are in the order of the sequence number. Thus, the packets expiration can be performed efficiently from the tails. Also, the fine-grained expiration avoids packets duplication or lost since all packets recorded in the queue are not reflected by the initial state of the secondary VM when failover occurs.

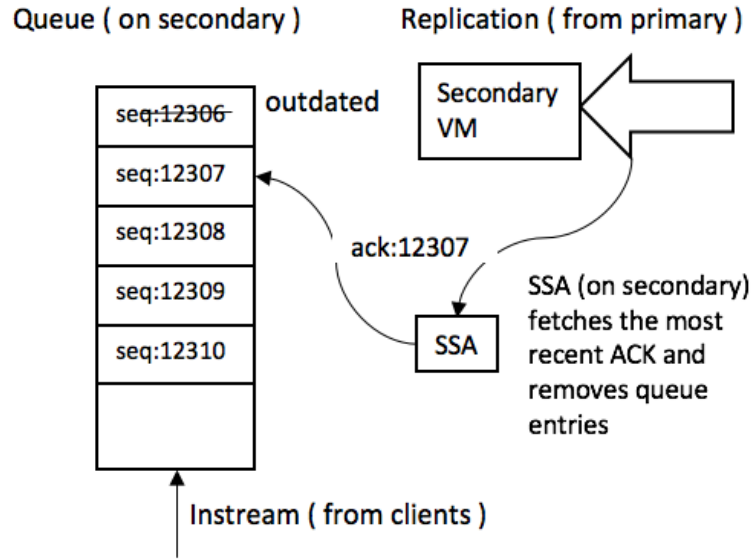


Figure 4.4: Queue expiration operation of SSA

4.2.3 Failover

In rRVM, a monitoring component is set to run on both primary and secondary hosts and to send heartbeats to each other. This component triggers a failover if the heartbeat is absent from the other side, and promotes the healthy host to be both service and network active. As discussed in Section 4.1, the failover proceeds differently for the failure on the primary and secondary hosts.

Primary failure

In the scenario of primary host failure, the secondary host is promoted to be service active. This enlists changing the functions of SSB. At failover, SSB stops redirecting network packets to the eth1 linked to the failed primary host. Instead, it transmits all packets to the secondary VM. Other changes that occur on failover include: First, incoming packets from the clients are blocked. Second, packets in the queue group are replayed. As mentioned in the previous section, the packets replay simulates real TCP interactions, thus the established TCP connection is maintained. Finally, after packets replay, SSB unblocks packets from the clients.

In the case of packets that can not be ordered (e.g., UDP packets), rRVM works as a best-effort service and all packets in the queue are replayed in

one batch. The protected system should cope with issues like lost, disorder, and duplication of packets caused by the replay in failover. Given that the UDP packets are generally less essential than TCP packets, we believe these issues, that exists in a very short period are acceptable by most of the services being protected.

Secondary failure

The failover of the secondary host is to promotes the primary VM to be network active. Several operations in SSA achieve this. First, it opens the eth0 port on the primary host and bridges the virtual network interface vif1.0 of the VM to eth0. Second, to facilitate the MAC table update in the hardware switch, SSA also sends an MAC masquerading packet with the address of the protected VM. Since the VM is already service active and the system state is up-to-date, no state update (e.g., network replay) is required here. Packets that are not be transmitted by the SSB are simply treated as packets lost by the protected VM.

We summarize the activities of SSA and SSB in the three stages in Table 4.2.

4.3 Outcomes

In this section, we evaluate the performance of rRVM by addressing various benchmark criteria, which correspond to our two prime objectives, low latency and strong consistency. In measuring rRVM performance impact, we firstly evaluate its latency overhead on the interactive network traffic by measuring the RTT between clients and an HTTP server. Then we measure the performance of a streaming server to test its impact on bulk data transfer.

To measure rRVM effectiveness, we adopt two metrics, recovery consistency and recovery time. Recovery consistency indicates how well an HA system preserves server states after a disaster and recovery time counts the time cost of a HA system for performing failover. To eliminate variance incurred by a WAN, we conduct all experiments with servers and clients in the same LAN.

4.3.1 Test environment

Considering that computers used as a primary host usually have better performance than those used as a secondary host, we do not use identical ma-

chines for the two hosts. In our setting, a Dell R210 II server with a Xeon 3.30GHz CPU and 2048MB RAM is used as the primary host and an IBM X3550 server with a Xeon 2.66GHz CPU and 1024MB RAM is used as the secondary host. Both the servers have 250GB SATA disks and two 1GB network interfaces connected to a business LAN and redirection LAN separately. We use Cisco SG300-28 as the Ethernet switch. We use Ubuntu 12.04 with a kernel version of 3.2.0-23 as the operating system and Xen 4.1.4 as the hypervisor for both hosts. We use Alpine Linux as the guest operating system, which is set to take 512MB memory.

4.3.2 Latency overhead on interactive data flow

Replication interval(ms):	Average latency(ms)			Timeout (>3000ms) number		
	50	100	300	50	100	300
Remus	59.20	68.52	77.48	28	30	28
Primitive	36.69	22.92	19.78	0	0	0
rRVM	35.55	25.01	20.37	0	0	0
Baseline		12.3			0	

Table 4.3: Average latency and timeout number of rRVM, Remus and primitive replication under different replication frequencies and network setups.

To test the latency impact of rRVM on interactive data flow, we use a standard LAMP (Linux, Apache, MySQL, PHP) hosted on the protected system and press the server by running JMeter set on 30 concurrent threads with each sending 2,000 requests per minute. We ignore the latency from the system warming up, as in the long term this gives no performance overhead on the system. For this measurement, we present the mean value calculated from the above experiment, and we record timeout number. Timeout number is an important indicator for a HA system as it helps disclose the occurrence and frequency of service unavailability.

We perform the latency test in four cases: (a) no replication, (b) primitive VM replication, (c) Remus VM replication, and (d) rRVM. First we conduct the test on the system without protection and mark the results as the baseline. In each case involving VM replication, we perform the replication in three different intervals: 50 ms, 100 ms, 300 ms. As shown in Table 4.3, primitive replication and rRVM perform similarly, latency is less than 40 milliseconds, and no timeout is found for both cases. Remus gives an average latency of > 50 milliseconds and it has a number of timeouts, which indicates a persisting unavailability of the protected system during normal

execution. Given n threads each sending r requests, we calculate the availability of the protected system as:

$$\mathcal{A} = 1 - \bar{t}/(r * n) \quad (1)$$

where \bar{t} = Average timeout number,
 r = Requests per thread,
 n = Thread number.

In our experiment, Remus provides the availability of $1 - 28/(2000 * 30) \approx 99.95\%$ for protection of a failure-free system. One thing to note is that the 0.05 % unavailability caused by excessive latency overhead contradicts to the purpose of HA systems.

We observe that low-frequency replication (e.g., 300 ms interval) alleviates the latency cost of Remus. As explained in Section 3.1.2, the latency is reduced at the expense of weakening recovery consistency, because the life span of the blocking network buffer does not cover the entire epoch. We will examine the state lost later in Section 4.3.4.

4.3.3 Throughput overhead on bulk data flow

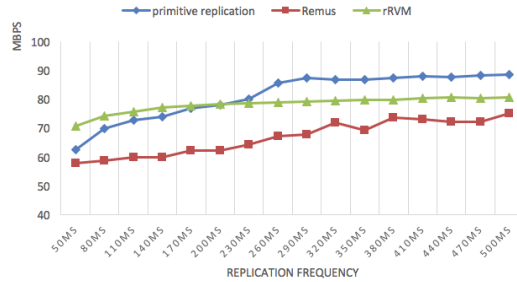


Figure 4.5: Throughput of bulk traffic

In testing the rRVM impact on bulk data flow, we measure the throughput of an upstream-intensive FTP server in protection where Siege [74] is used, on a client machine, to send requests to the server that responds with 3MB of data for each transaction. Then we record the downloading time to calculate the aggregate throughput. We conduct the experiment on 17 different replication frequencies in a range from 50 to 500 milliseconds and calculate the mean value of throughput from 15 identical tests repeated for each frequency. We omit downstream-intensive service in that it gives a similar system performance, and services of this kind are relatively rare.

Figure 4.5 compares the results from the three HA systems. As we expected, rRVM offers 10% less throughput than the primitive VM replication for >200 milliseconds intervals due to the additional costs of network redirection. Surprisingly, the throughput of primitive replication is worse than rRVM for <200 milliseconds intervals. This can be explained by the fact that in primitive replication, the replication traffic and the bulk data flow compete for bandwidth on the eth0 of the primary host and the overhead of such competition surpasses that of network redirection in rRVM (§4.2.2). As compared to Remus, the superiority of rRVM is obvious and consistent across all replication intervals.

4.3.4 Recovery consistency

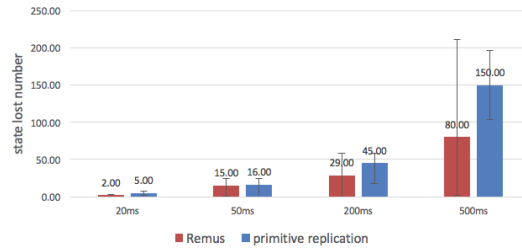


Figure 4.6: Recovery consistency evaluation

When testing recovery consistency, we develop an Ajax front-end on top of the LAMP model and set a client to send 5000 requests with a frequency of 50 requests per second. The server stores the requests in both memory and database to simulate a standard four-layer architecture (i.e., API, logic, cache, permanent storage) widely used in modern Cloud applications. Then, the state lost can be discovered by comparing the data recorded by the server and the original sequence with which the client sent out in the packets.

As shown in Figure 4.6, the recovery consistency of primitive VM replication degrades linearly with the replication frequency. For Remus, the blocking network buffer protects the service from the *output commit problem* and alleviates such degradation. However, the state lost can be seen in the case of low-frequency replication. We omit the data for rRVM in Figure 4.6 as all 5000 requests are fully retained after packets replay.

As explained in Section 3.1.2, the state lost in Remus is because the network buffer protection can cover only a fixed part of an epoch. Thus, as the replication frequency becomes low, the state lost is more observable. In contrast, rRVM recovery consistency shows little correlation to the replica-

tion frequency, thus it does not rely on high-frequency replication to enable strong consistency. Note that low-frequency replication is more efficient in both CPU and memory usage than high-frequency replication. Thus, the peripheral performance impact of rRVM should be much less than that of existing solutions in achieving the same level of recovery consistency.

4.3.5 Recovery time

Recovery type:	Min	Normal	Max
Primitive VM Replication	<1s	2-5s	>30s
Remus	<1s	2-5s	>30s
rRVM network activity	<1s	2-5s	<5s
rRVM service activity	0s	1-2s	<3s

Table 4.4: Recovery time

We use ICMP packets collected on a client for measuring the failover time. With the same setup as the above experiments, we measure the time cost from when a failure occurs and when the first response is received at the client after the recovery of the protected VM. In the testing, occasionally we observe more than 30 seconds recovery time for both primitive replication and Remus occasionally. This excessive recovery time is caused when the physical switch can not detect the new location of the protected VM and update its MAC table accordingly (§4.2.3) because there is no network output from the newly recovered VM. Only after the MAC table aging time is past, will the switch broadcast the new incoming packets to identify the new physical port to which the VM connects. In the hardware switch we use, the MAC table aging time is 30 seconds, which matches to our testing result. It is worth noting here that this value varies in practice depending on the configuration that is applied.

For network activity failover, SSA sends out an MAC masquerading with the address of the protected VM to actively update the MAC table in the hardware switch (§4.2.3). The worst case failover time here is 5 seconds. For rRVM service activity failover, which does not require MAC table updating, the worst case failover time is 3 seconds.

4.4 Alternative research methods

In this section, we discuss alternate design options and potential optimizations of rRVM.

4.4.1 No programming approaches for network redirection

An option for network redirection is link aggregation, namely, BONDING (5) and BRIDGE (8). In this method, virtual network interface vif1.0 and physical interfaces eth0, eth1 are bridged together on the secondary host, which practically makes the secondary host work as a layer 2 switch. During normal execution, packets are redirected through eth0 and eth1; for service activity failover, the redirection is switched to be in-between eth0 and vif1.0. On the primary host, physical interfaces eth0 and eth1 are configured in mode one network bonding (active-backup), and virtual network interface vif1.0 is bridged to the logical bonded interface of the two slaves. In this way, eth1 is set as the active interface for receiving packets during normal execution and network activity failover is conducted by the mode one bonding. This solution is simpler and more compatible than the in-use approach as it requires only scripts for network configuration.

The drawback of this solution is the asymmetric configuration in which a physical machine can be either a primary or secondary host for all VMs on it, because network configuration is performed in a granularity of host rather than VM. This incurs difficulty in system management. For example, this solution is not suitable for certain tasks like chained failover that enlists multiple backup points to avoid failures of multiple host. The larger deployment granularity also causes resource waste as it contradicts the flexibility of modern cloud applications. In contrast, the symmetric configuration provided by rRVM supports flexible deployment so that a host can be a primary or secondary host to different protected VMs at the same time.

Another approach to network redirection is port mirroring on the hardware switch. In this method, the hardware switch broadcasts instream to both primary and secondary hosts; the primary VM consumes the input and executes the application binary of protected service; and the secondary host just stores all the packets received. To avoid an *output commit problem* [66] the primary host has to buffer the network output until the corresponding input recording is acknowledged by the secondary host. So the latency imposed here is the same as the network redirection solution in use. The most obvious drawback of this approach is that it results in an even more rigid deployment policy than the asymmetric configuration, as the roles of the physical machines (i.e., the primary and secondary hosts) are fixed by the settings in the hardware switch. Moreover, hardware switch is not accessible in all environments.

4.4.2 Layer 3 network redirection

In our initial design, network redirection is designed on layer 3. The intuition behind this is Network Address Translation (NAT). First, when the network proxy receives the broadcast ARP request destined to the protected VM, it replies with the MAC address of its host (i.e., the secondary host). Second, for inbound packets, the target IP addresses are translated to the protected VM's IP address of the redirection LAN. Third, for outbound packets, the source IP addresses are altered back to the original IP of the business LAN. We note that the layer 3 network redirection does not trigger broadcast radiation, which could be caused by misconfigured layer 2 software switches.

The difficulty of layer 3 network redirection is ARP cache updating when network activity failover occurs. During normal execution, an ARP response from the network proxy updates the ARP cache in all peers belonging to the business LAN, where the MAC associated with the protected VM IP is set to be the MAC of secondary host. When network activity failover, the MAC address needs to be converted back to the protected VM by an unsolicited ARP. However, the router normally drops the ARP, seeing it as spoofed. Thus, the router keeps an obsoleted ARP cache, and the protected VM stays invisible in the business LAN after the failover. In contrary to layer 3 network redirection, the layer 2 alternative keeps the destination MAC address the same all the time, so no ARP cache update is required for the proposed rRVM.

4.4.3 Network behavior differences in PV (paravirtualization) and HVM (Hardware VM)

In a paravirtualization environment, the guest operating system offloads TCP operations to the host operating system, as it does not include a full network stack. However, the TCP stack of the host operating system is bypassed by rRVM since it uses packet sockets. To make sure software switches are working properly, we implement basic TCP operations, namely, TCP checksum, segmentation and congestion control, in the user space.

Apart from that, further offloading the TCP operations to the network interface card (NIC) [75] presents another approach to conducting the TCP operations. For a HVM environment, the user space implementation of the TCP stack is not required because packets are already processed by the TCP stack of the guest operating system before being delivered to software switches through the virtual network interface.

4.4.4 The impact of non-determinisms

In practice, non-determinisms might corrupt the state consistency in two ways: by non-deterministic system calls and by non-deterministic scheduling. In this section, we discuss their implications in rRVM.

Some HA solutions enlist system calls in event replay [76]. However, in the case of rRVM, the impact of non-deterministic system calls on the protected systems is negligible. This can be explained as 1) The time between the last replication and crash is less than 1 second. The divergence that appears in such a short time window is small. 2) Most random states are not critical for recovery. Examples here include timestamp for logging, random numbers for load balancing, nonce for playback defense, and padding for block ciphers. For those random states that are essential for recovery, such as the initial key used in credentials exchange, their impact also can be also ignored since the handshakes seldom happen during the 1 second failure epoch. To demonstrate this, consider the following extreme case. If all random states are essential, which follows that every transaction relies on previously generated random data, then transactions within the 1 second failure epoch will all fail. In this case the success rate is calculated as $100(1/(10 * 60)) = 99.98\%$, where availability is calculated commonly for every 10 minutes (n.b., during normal execution, the availability is intact). Considering all the above, we can safely omit non-deterministic system calls in event replay and focus on network packets.

Non-deterministic scheduling, might also corrupt the state consistency. In bug-free services, scheduling application-level threads are not allowed to generate random essential service states or it is considered as dangerous practice that should be avoided. Scheduling tasks (processes) amongst cores causes CPU state inconsistency in the SMP architect, which should be examined carefully in an instruction level lockstep approach. Fortunately, this is not a concern for high-level solutions like rRVM because a deviation of the CPU state does not cause system panic in the VM replication approach.

4.4.5 Potential optimizations

One potential optimization of rRVM is tunneling. As discussed in Section 4.2.2, rRVM replication traffic competes with bulk data flow for bandwidth in services that are both downstream and upstream-intensive. By tunneling the network, business data flow can be redirected back through the business LAN and the replication and business traffic can be completely isolated.

Another potential optimization of rRVM is a kernel mode software switch. Kernel space applications are more efficient than user space applications as

they do not include operations such as context switch and additional memory copying. Moreover, TCP checksum and segmentation (§4.2.2) can be executed faster in kernel space because hardware offloading can be enabled by simply setting `ip_summed` and `nohdr` in `SKB`. The drawback of kernel space application is generality since the interfaces for kernel module programming varies, even in the same operating system kernel of a different release version. Moreover, it usually requires a rewrite of the whole system to port a kernel module to another operating system.

4.5 Application and Impact of Research Results

rRVM demonstrates an enhanced HA system that offers an optimal combination of strong recovery consistency and low response latency. As compared to the previous solutions, we decouple network and service activity and propose a novel reverse replication approach. rRVM does not trade off on performance: despite the apparent performance advantages, the following virtues of virtual machine replication are fully preserved.

Transparency: rRVM is transparent to the protected system. Aiming at a less intrusive solution, we design the network redirection in layer 2, we adopt a queue group to simulate TCP interaction for network replay, and we implement all functionalities in domain 0 (i.e., the host operating system).

Generality: rRVM is agnostic to the virtual machine monitor (VMM) and the host operating system. In our implementation, 95% of rRVM does not contain any platform-specific constructs. Owing to this, it can be easily ported to any reasonably mature VMMs (e.g., KVM [26]).

Real-time failover: rRVM has a shorter and more stable failover time than previous solutions. After failover occurs, rRVM sends out an MAC masquerading packet containing the address of the protected system, which practically reduces the time for MAC table updating in an physical switch.

Chapter 5

Conclusion and Future Works

In summary, we believe the improvements in rRVM design can lift the limitations of VM replication and HA systems in general. With the new architecture invented, rRVM exceeds the capabilities of all known VM replication systems; With the existing virtualization infrastructure untouched, rRVM can work in existing cloud environments out of the box. The technology is particularly useful for task critical backend systems that demand high frequent requests and high availability guarantee, for example, the backend of securities trading, bank or medical systems. In those systems, 0.001% enhancement of the availability can create enormous value for a corporation.

Moreover, the design of decoupling network and service has more practical implication beyond the virtual machine replication, although the proposed reverse replication approach is implemented only on virtual machines for HA service. For instances, the HA components of databases, distributed file systems, or cache systems etc., in which master/slave configuration is widely used, can be upgraded with this design to gain a better latency and consistency metrics. For instance, in a database, database query requests can be sent to, and be queued in the secondary host, and in turn be relayed to the primary host. The replay queue in secondary stores transactions of SQL in this case instead of raw ethernet packets (as in rRVM). And the failover and replay mechanisms can be made the same.

There is great potential to improve rRVM itself as well. Firstly, besides the innovation on the architecture design, we can put more efforts on system engineering. One of the potential approaches is to implement the rRVM in kernel mode to further reduce the overhead of memory copying, context switching etc. Secondly, we can use Unikernel instead of ordinary virtual machine guests to further reduce the overhead imposed by the high availability system. Incorporating with LibOS (a.k.a., Unikernel) [30], the perfor-

mance of rRVM, that is, latency and bandwidth consumption can be largely enhanced. This performance improvement can further lead to wider use cases of this technology. For example, rRVM can be extended to an inter-datacenter disaster recovery solution. Here, the LibOS's small footprint can solve the bandwidth bottleneck in a WAN environment. Also, rRVM naturally supports low-frequency VM replication which recovers a VM from a checkpoint replicated seconds ago, thus the WAN latency is not an obstacle for inter-datacenter recovery. This will be left as a significant future work.

Bibliography

- [1] Muiyang He, Shaoning Pang, Denis Lavrov, Ding Lu, Yuan Zhang, and Abdolhossein Sarrafzadeh. Reverse replication of virtual machines (rrvm) for low latency and high availability services. In *Proceedings of the 9th International Conference on Utility and Cloud Computing, UCC '16*, pages 118–127, New York, NY, USA, 2016. ACM.
- [2] D. Lavrov, V. Blanchet, S. Pang, M. He, and A. Sarrafzadeh. Cor-honeypot: Copy-on-risk, virtual machine as honeypot in the cloud. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 908–912, June 2016.
- [3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.*, 2003.
- [4] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *High Performance Distributed Computing, 1997. Proceedings. The Sixth IEEE International Symposium on*, 1997.
- [5] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003.
- [6] Azzedine Boukerche, Raed A. Al-Shaikh, and Mirela Sechi Moretti Annoni Notare. Towards Highly Available and Scalable High Performance Clusters. *J. Comput. Syst. Sci.*, 2007.
- [7] Will Reese. Nginx: The High-performance Web Server and Reverse Proxy. *Linux J.*, 2008.
- [8] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux J.*, 2004.

- [9] V. Chaurasiya, P. Dhyani, and S. Munot. Linux highly available (ha) fault-tolerant servers. In *Information Technology, (ICIT 2007). 10th International Conference on*, 2007.
- [10] Amazon.com. Amazon ec2. <http://aws.amazon.com/ec2/>.
- [11] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Commun. ACM*, 2010.
- [12] Fraser Clark, Hansen Hand, Limpach Jul, and Warfield Pratt. Live Migration of Virtual Machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design and Implementation - Volume 2*, 2005.
- [13] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. *SIGOPS Oper. Syst. Rev.*, 2005.
- [14] apache.org. Apache module mod_proxy. <http://httpd.apache.org/docs/2.2/mod/mod\underline{\hspace{.06in}}proxy.html/>.
- [15] nginx.com. High availability clustering with nginx plus. <http://nginx.com/blog/high-availability-clustering-with-nginx-plus/>.
- [16] mysql.com. Mysql ha/scalability guide. <http://dev.mysql.com/doc/mysql-ha-scalability/en/index.html>.
- [17] Sage A. Weil. *Ceph: Reliable, Scalable, and High-performance Distributed Storage*. PhD thesis, University of California at Santa Cruz, 2007.
- [18] Alex Davies and Alessandro Orsaria. Scale out with glusterfs. *Linux J.*, 2013.
- [19] drbd.linbit.com. DRBD, Software developement for high availability clusters.
- [20] Lefebvre Cully, Feeley Meyer, and Warfield Hutchinson. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.

- [21] Masoomeh Rudafshani, Paul A. S. Ward, and Bernard Wong. Memred: Towards reliable web applications. In *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*, 2012.
- [22] Benjamin Wester, James Cowling, Edmund B. Nightingale, Peter M. Chen, Jason Flinn, and Barbara Liskov. Tolerating latency in replicated state machines through client speculation. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009.
- [23] qemu project.org. Qemu. <https://github.com/qemu/qemu>.
- [24] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.
- [25] bochs.sourceforge.net. Bochs. <https://github.com/larsr/bochs>.
- [26] Y. Kamay A. Kivity, U. Lublin D. Laor, and A. Liguori. KVM: the Linux Virtual Machine Monitor.
- [27] YaoZu Dong, Wei Ye, YunHong Jiang, Ian Pratt, ShiQing Ma, Jian Li, and HaiBing Guan. COLO: COarse-grained LOck-stepping Virtual Machines for Non-stop Service. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, 2013.
- [28] Maohua Lu and Tzi cker Chiueh. Fast memory state synchronization for virtualization-based fault tolerance. In *DSN*, 2009.
- [29] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: Separating data and metadata for efficient and available storage replication. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, 2012.
- [30] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.
- [31] Anil Madhavapeddy and David J. Scott. Unikernels: The rise of the virtual library operating system. *Commun. ACM*, 57(1):61–69, January 2014.

- [32] Shen Lu. Virtual machine memory access tracing with hypervisor exclusive cache. In *USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, 2007.
- [33] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. *SIGARCH Comput. Archit. News*, 2006.
- [34] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Mobile Computing Systems and Applications, 2002. Proceedings Fourth IEEE Workshop on*, 2002.
- [35] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proceedings of the 4th ACM European Conference on Computer Systems*, 2009.
- [36] Hargrove P. Duell, J. and E Roman. Requirements for linux checkpoint/restart. berkeley lab technical report. Technical report, Lawrence Berkeley National Laboratory, 2002.
- [37] E.A Roman. Survey of checkpoint/restart implementations. Technical report, Lawrence Berkeley National Laboratory, 2002.
- [38] Dejan S. Milošević, Fred Douglass, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process Migration. *ACM Comput. Surv.*, 2000.
- [39] Amnon Barak and Ami Litman. Mos: A multicomputer distributed operating system. *Softw. Pract. Exper.*, 1985.
- [40] David Cheriton. The v distributed system. *Commun. ACM*, 1988.
- [41] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. *SIGOPS Oper. Syst. Rev.*, 1981.
- [42] John K. Ousterhout, Andrew R. Cherenon, Frederick Douglass, Michael N. Nelson, and Brent B. Welch. The sprite network operating system. *Computer*, 1988.
- [43] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. 1986.

- [44] Frederick Douglass. Transparent process migration in the sprite operating system. Technical report, University of California at Berkeley, 1990.
- [45] David L. Black, Dejan S. Milojević, Randall W. Dean, Michelle Dominianni, Alan Langerman, and Steven J. Sears. Extended memory management (xmm): Lessons learned. *Softw. Pract. Exper.*, 1998.
- [46] B J Walker, Mathews, and R M. Process migration in aix's transparent computing facility (tcf. *IEEE Technical Committee on Operating Systems Newsletter*, 1989.
- [47] K. Haselhorst, M. Schmidt, R. Schwarzkopf, N. Fallenbeck, and B. Freisleben. Efficient storage synchronization for live migration in cloud infrastructures. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, 2011.
- [48] Peter J. Denning. The working set model for program behavior. *Commun. ACM*, 1968.
- [49] P. J. Denning. Working sets past and present. *IEEE Trans. Softw. Eng.*, 1980.
- [50] James S. Plank, Yuqun Chen, Kai Li, Micah Beck, and Gerry Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Softw. Pract. Exper.*, 1999.
- [51] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005.
- [52] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 2002.
- [53] Ali Mashtizadeh, Emr  Celebi, Tal Garfinkel, and Min Cai. The design and evolution of live storage migration in vmware esx. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, 2011.
- [54] Bo Jiang, Binoy Ravindran, and Changsoo Kim. Lightweight live migration for high availability cluster service. In *Proceedings of the 12th International Conference on Stabilization, Safety, and Security of Distributed Systems, SSS'10*, pages 420–434, Berlin, Heidelberg, 2010. Springer-Verlag.

- [55] Jacob R. Lorch, Andrew Baumann, Lisa Glendenning, Dutch T. Meyer, and Andrew Warfield. Tardigrade: Leveraging lightweight virtual machines to easily and efficiently construct fault-tolerant services. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 575–588, Berkeley, CA, USA, 2015. USENIX Association.
- [56] Eunbyung Park, Bernhard Egger, and Jaejin Lee. Fast and space-efficient virtual machine checkpointing. *SIGPLAN Not.*, 2011.
- [57] Changyeon Jo, Erik Gustafsson, Jeongseok Son, and Bernhard Egger. Efficient live migration of virtual machines using shared storage. *SIGPLAN Not.*, 2013.
- [58] Anthony Nocentino and Paul M. Ruth. Toward dependency-aware live virtual machine migration. In *Proceedings of the 3rd International Workshop on Virtualization Technologies in Distributed Computing*, 2009.
- [59] Ali José Mashtizadeh, Min Cai, Gabriel Tarasuk-Levin, Ricardo Koller, Tal Garfinkel, and Sreekanth Setty. Xvmotion: Unified virtual machine migration over long distance. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, 2014.
- [60] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 2008.
- [61] Fang Hao, T. V. Lakshman, Sarit Mukherjee, and Haoyu Song. Enhancing dynamic cloud-based services using network virtualization. *SIGCOMM Comput. Commun. Rev.*, 2010.
- [62] E.C. Perkins. Rfc3344: Ip mobility support for ipv4. <https://tools.ietf.org/html/rfc3344>, 2002.
- [63] Young-Chul Shim. Distributed cloud computing environment enhanced with capabilities for wide-area migration and replication of virtual machines. *CoRR*, 2014.
- [64] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, 2007.

- [65] Weida Zhang, King Tin Lam, and Cho-Li Wang. Adaptive live vm migration over a wan: Modeling and implementation. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, 2014.
- [66] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.*, 2002.
- [67] Seth Gilbert and Nancy Lynch. Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 2002.
- [68] Dan Kegel. The C10K problem.
- [69] Jeff Dean. *Designs, Lessons and Advice from Building Large Distributed Systems*, 2009.
- [70] Ilya Grigorik. *High Performance Browser Networking*, chapter 1. Primer on Latency and Bandwidth. O’REILLY, 2013.
- [71] opensbsd.org. PF: The OpenBSD Packet Filter.
- [72] W. Richard Stevens. *TCP/IP Illustrated (Vol. 1): The Protocols*. Addison-Wesley Longman Publishing Co., Inc., 1993.
- [73] Ilya Grigorik. *High Performance Browser Networking*, chapter 2. Building Blocks of TCP. O’REILLY, 2013.
- [74] joedog. *Siege*, 2012.
- [75] Juan M. Solá-Sloan and Isidoro Couvertier-Reyes. A TCP Offload Engine Emulator for Estimating the Impact of Removing Protocol Processing from a Host Running Apache HTTP Server. In *Proceedings of the 2009 Spring Simulation Multiconference*, 2009.
- [76] Dmitrii Zagorodnov, Keith Marzullo, Lorenzo Alvisi, and Thomas C. Bressoud. Practical and Low-overhead Masking of Failures of TCP-based Servers. *ACM Trans. Comput. Syst.*, 2009.

Chapter 6

Appendix

This appendix contains some of the scripts used to establish the running environment of rRVM as well as the day-to-day operation.

```
#!/bin/bash

# install dependency
echo -e "Begin installing Xen dependency...\n"
apt-get update
apt-get install lrzsz expect git-core bridge-utils
libjpeg-turbo8-dev libvncserver-dev iproute libcurl3
libcurl4-openssl-dev bzip2 module-init-tools transfig
tgif texinfo pciutils-dev build-essential make gcc
libc6-dev zlib1g-dev python python-dev python-twisted
libncurses5-dev patch libvncserver-dev iasl libbz2-dev
e2fslibs-dev uuid-dev libtext-template-perl autoconf
debhelper debconf-utils docbook-xml docbook-xsl dpatch
xsltproc rconf bison flex gcc-multilib ocaml-findlib
libyajl-dev yajl-tools libglib2.0-dev libsdl-ttf2.0-0
libsdl-ttf2.0-dev texlive-latex-recommended texlive
texlive-base tetex-brev bin86 bcc -y

# get packages from 2.73
# first confirm git get latest rRVM on 2.73
echo -e "Begin get packages(Xen, rRVM, alpine.iso,
  config, etc.)\n"
./scp.exp 192.168.2.73 root cp-x2520
/root/create_alpine_vm/create_alpine_vm.tgz /root
pull 0 -1
```

```

cd /root
tar -xvf create_alpine_vm.tar.gz
tar -xvf xen-4.1.4.tar.gz
tar -xvf rRVM.tar.gz

# compile xen
echo -e "Begin compiling Xen...\n"
cd /root/xen-4.1.4/
make world -j8 && make install-tools
PYTHON_PREFIX_ARG= && make install

echo -e "Set Xen auto-start when booting...\n"
update-rc.d xencommons defaults 19 18
update-rc.d xend defaults 20 21
update-rc.d xenddomains defaults 21 20
update-rc.d xen-watchdog defaults 22 23

mv ./dist/install/usr/lib64/* ./dist/install/usr/lib/
mv ./dist/install/usr/lib64/xen/bin/*
./dist/install/usr/lib/xen/bin/
cd ./dist
./install.sh

echo -e "Copy Xen config file for xend-relocation...\n"
cd /root
cp xend-config.sxp_template /etc/xen/xend-config.sxp
# check echo
#grep '^ (xend-relocation ' /etc/xen/xend-config.sxp

cat << EOF > /etc/ld.so.conf.d/libc.conf
# libc default configuration
/usr/local/lib
ldconfig
EOF

ln -s /usr/lib/xen /usr/lib/xen-default

# generate grub.cfg
echo -e "Generate grub menu for Xen...\n"

```

```

cd /root
update-grub
./generate_grub.sh

# create alpine.img for xen-remus testing
#echo -e "Create alpine.img for xen-remus...\n"
#mkdir -p /var/lib/xen/images
#dd if=/dev/zero of=/var/lib/xen/images/xen_alpine.img
  bs=1M count=3000

# copy interface_normal template, need to be modified
  manually
echo -e "Copy Xen interface_normal template...\n"
cp /etc/network/interfaces /etc/network/interfaces_bak
cp /root/interfaces_normal /etc/network/interfaces

# need restart
#reboot

#!/usr/bin/expect —

proc Usage_Exit {myself} {
  puts ""
  puts "### USAGE: $myself ip user passwd sourcefile
  destdir direction bwlimit timeout"
  puts ""
  puts "          sourcefile: a file or directory
  to be transferred"
  puts "          destdir:    the location that the
  sourcefile to be put into"
  puts "          direction:  pull or push."
  puts "          pull: remote -> local"
  puts "          push: local -> remote"
  puts "          bwlimit:    bandwidth limit, kbit/s,
  0 means no limit"
  puts "          timeout:    timeout of expect, s,
  -1 means no timeout"
  puts ""
  exit 1
}

```

```

if { [llength $argv] < 8 } {
    Usage_Exit $argv0
}

set ip [lindex $argv 0]
set user [lindex $argv 1]
set passwd [lindex $argv 2]
set sourcefile [lindex $argv 3]
set destdir [lindex $argv 4]
set direction [lindex $argv 5]
set bwlimit [lindex $argv 6]
set timeoutflag [lindex $argv 7]

set yesnoflag 0
set timeout $timeoutflag

for {} {1} {} {

# for is only used to retry when "Interrupted system
  call" occured

# scp: -p Tells scp to preserve file attributes and
  timestamps
# scp: -r Copy directories recursively. Does not
  follow symbolic links

# rsync: -a, --archive, archive mode, equivalent to
  -rlptgoD
# rsync: -r, --recursive, recurse into directories
# rsync: -t, --times, preserve times
# rsync: -z, --compress, compress file data
# rsync: --progress show progress during transfer

if { $direction == "pull" } {

    if { $bwlimit > 0 } {
        spawn /usr/bin/rsync -artz --bwlimit=$bwlimit
    }
}
}

```

```

        -e "/usr/bin/ssh -l$user"
        $ip:$sourcefile $destdir
    } elseif { $bwlimit == 0 } {
        spawn /usr/bin/scp -r -p $user@$ip:$sourcefile
        $destdir
    } else {
        Usage_Exit $argv0
    }
} elseif { $direction == "push" } {

    if { $bwlimit > 0 } {
        spawn /usr/bin/rsync -artz --bwlimit=$bwlimit
        -e "/usr/bin/ssh -l$user"
        $sourcefile $ip:$destdir
    } elseif { $bwlimit == 0 } {
        spawn /usr/bin/scp -r -p $sourcefile $user@$ip:$destdir
    } else {
        Usage_Exit $argv0
    }
} else {
    Usage_Exit $argv0
}

expect {

    "assword:" {
        send "$passwd\r"
        break;
    }

    "yes/no)?" {
        set yesnoflag 1
        send "yes\r"
        break;
    }

    "FATAL" {
        puts "\nCONNECTERROR: $ip occur FATAL

```

```

        ERROR!!!\n"
        exit 1
    }

    timeout {
        puts "\nCONNECTERROR: $ip logon TIMEOUT!!!\n"
        exit 1
    }

    "No route to host" {
        puts "\nCONNECTERROR: $ip No route to
        host!!!\n"
        exit 1
    }

    "Connection Refused" {
        puts "\nCONNECTERROR: $ip Connection
        Refused!!!\n"
        exit 1
    }

    "Connection refused" {
        puts "\nCONNECTERROR: $ip Connection
        Refused!!!\n"
        exit 1
    }

    "Host key verification failed" {
        puts "\nCONNECTERROR: $ip Host key
        verification failed!!!\n"
        exit 1
    }

    "Illegal host key" {
        puts "\nCONNECTERROR: $ip Illegal host
        key!!!\n"
        exit 1
    }

    "Connection Timed Out" {

```

```

        puts "\nCONNECTERROR: $ip logon
        TIMEOUT!!!\n"
        exit 1
    }

    "Interrupted system call" {
        puts "\n$ip Interrupted system call!!!\n"
    }
}

}

if { $yesnoflag == 1 } {
    expect {
        "assword:" {
            send "$passwd\r"
        }

        "yes/no)?" {
            set yesnoflag 2
            send "yes\r"
        }
    }
}

if { $yesnoflag == 2 } {
    expect {
        "assword:" {
            send "$passwd\r"
        }
    }
}

expect {
    "assword:" {
        send "$passwd\r"
        puts "\nPASSWORDERROR: $ip PASSWORD
        ERROR!!!\n"
        exit 1
    }
}

```

```
    eof {  
        puts "ABS_OK_SCP: $ip\n"  
        exit 0;  
    }  
}
```



Declaration

Name of candidate: Muyang He

This Thesis/Dissertation/Research Project entitled: Using VM Replication to Build a Consistent, Fast HPA System is submitted in partial fulfillment for the requirements for the Unitec degree of

Principal Supervisor: Professor Pang

Associate Supervisor/s: _____

CANDIDATE'S DECLARATION

I confirm that:

- This Thesis/Dissertation/Research Project represents my own work;
- The contribution of supervisors and others to this work was consistent with the Unitec Regulations and Policies.
- Research for this work has been conducted in accordance with the Unitec Research Ethics Committee Policy and Procedures, and has fulfilled any requirements set for this project by the Unitec Research Ethics Committee.

Research Ethics Committee Approval Number:

Candidate Signature: 何慕昂 Date: 29/07/2017

Student number: 1434735



Institute of Technology

TE WHARE WANANGA O WAIRAKA

Full name of author: Muyong He

ORCID number (Optional):

Full title of thesis/dissertation/research project ('the work'):
Using VM Replication to Build a Consistent, Fast HA System

Practice Pathway: cloud computing

Degree: Master of Computing

Year of presentation: 2017

Principal Supervisor: Professor Pang

Associate Supervisor:

Permission to make open access

I agree to a digital copy of my final thesis/work being uploaded to the Unitec institutional repository and being made viewable worldwide.

Copyright Rights:

Unless otherwise stated this work is protected by copyright with all rights reserved.

I provide this copy in the expectation that due acknowledgement of its use is made.

AND

Copyright Compliance:

I confirm that I either used no substantial portions of third party copyright material, including charts, diagrams, graphs, photographs or maps in my thesis/work or I have obtained permission for such material to be made accessible worldwide via the Internet.

Signature of author: 何勇

Date: 29.07.2017