Lightweight Live Migration for High Availability Cluster Service *

Bo Jiang[1], Binoy Ravindran[1], and Changsoo Kim[2]

[1] ECE Dept., Virginia Tech {bjiang,binoy}@vt.edu
[2] ETRI, Daejeon, South Korea cskim7@etri.re.kr

Abstract. High availability is a critical feature for service clusters and cloud computing, and is often considered more valuable than performance. One commonly used technique to enhance the availability is live migration, which replicates services based on virtualization technology. However, continuous live migration with checkpointing will introduce significant overhead. In this paper, we present a lightweight live migration (LLM) mechanism to integrate whole-system migration and input replay efforts, which aims at reducing the overhead while providing comparable availability. LLM migrates service requests from network clients at high frequency during the interval of checkpointing system updates. Once a failure happens to the primary machine, the backup machine will continue the service based on the virtual machine image and network inputs at their respective last migration rounds. We implemented LLM based on Xen and compared it with Remus—a state-of-the-art effort that enhances the availability by checkpointing system status updates. Our experimental evaluations show that LLM clearly outperforms Remus in terms of network delay and overhead. For certain types of applications, LLM may also be a better alternative in terms of downtime than Remus. In addition, LLM achieves transaction level consistency like Remus.

1 Introduction

High availability (HA) is a critical feature of modern enterprise-scale data and service clusters. Any downtime that a server cluster experiences may result in severe loss on both revenue and customer loyalty. Therefore, high availability is often considered more valuable than performance [1]. Especially along with the development of cloud computing—one of the most remarkable development opportunities for the Internet—computation and storage are gradually moving from clients to cluster servers in a cloud [2]. Thus the availability of the resources in a cloud is essential to the success of cloud computing. Nowadays, high availability is still a very challenging problem [3],

* This work was supported by the IT R&D program of MKE/KEIT, South Korea [2007S01602, Development of Cost Effective and Large Scale Global Internet Service Solution]

because there are many failure categories to handle so as to guarantee the continuous operation. Among the failure models, hardware fail-stop failure is one of the most commonly studied [4].

Naturally, replication is an important approach to increase the availability by providing redundancy—once a failure occurs to a replica, services that run upon it can be taken over by other replicas [5]. Replication may be realized as several redundancy types: spatial redundancy, temporal redundancy, and structural (or contextual) redundancy [6]. For example, service migration used in server clusters [7] provides spatial redundancy, since it requires extra hardware as running space of services.

For any redundancy type, the consistency among multiple replicas needs to be guaranteed, in a certain consistency level. Based on Brewer's CAP theorem [8], the consistency is a competing factor to the availability, i.e., there is a trade-off between them.

By running multiple virtual machines (VM) on a single physical machine, virtualization technology can facilitate the management of services, such as replication via migration. Virtualization technology separates service applications from physical machines with a virtual machine monitor (VMM), thus provides increased flexibility and improved performance [9]. With these advantages, virtualization technology makes it easy to migrate services across physical machines. Usually we call the machine which provides regular services as the *primary machine*, and the one which takes over the services at a failure as the *backup machine*.

To achieve high availability, live migration is typically used to minimize the downtime. Here live migration means executing the migration without suspending the primary machine. Instead, the primary machine keeps running until the migration is completed. Live migration was first studied in [7], where the migration is executed only once and triggered on demand of users. Such a one-time live migration is suitable for data processing or management purposes. However, it does not work for disaster recovery because the migration cannot be triggered by a failure event.

In [10], the authors introduced the idea of checkpointing to live migration by presenting Remus—a periodical live migration process for disaster recovery. Using checkpointing, the primary machine keeps migrating a whole system, including CPU/memory status updates as well as writes to the file system to the backup machine at configured frequency. Once a failure happens so that the migration data stream is broken, the backup machine will take over the service immediately starting from the latest stop point of checkpointing. However, checkpointing at high frequency will introduce significant overhead, as plenty of resources such as CPU and memory are consumed by the migration. In this case clients that request services may experience significantly long delays. If on the contrary the migration runs at low frequency trying to reduce the overhead, there maybe many service requests that are duplicately served. Actually this will produce the same effect of increasing the downtime from the perspective of those new requests that come after the duplicately served requests.

In fact, there is another approach for service replication—input replay [11]. With input replay, the data to replicate will be much less than whole-system replication. Although input replay cannot replicate the system status exactly, such a Point-in-Time consistency is actually very challenging equally for all the replication approaches in real implementations [12].

Based on input replay, the objective of this paper is to reduce the overhead of whole-system checkpointing when achieving comparable downtime and the same level consistency as those of Remus. In this way, we will be able to leverage the advantage of input replay without suffering from its flaw on consistency.

Based on the checkpointing approach of Remus, we developed an integrated live migration mechanism, called Lightweight Live Migration (LLM), which consists of both whole-system checkpointing and input replay. The basic idea is as follows:

1) The primary machine keeps migrating to the backup machine: a) the guest VM image (including CPU/memory status updates and new writes to the file system) at low frequency; and b) service requests from network clients at high frequency; and

2) Once a failure happens to the primary machine, the backup machine will continue the service based on the guest VM image and network inputs at their respective last migration rounds.

Especially when the network service involves a lot of computation or database updates, CPU/memory status updates and writes to the file system will be a big bulk of data. Compared with past efforts such as Remus, migrating the guest VM image at low frequency with input replay as an auxiliary may significantly reduce the migration overhead.

We compared LLM with Remus in terms of the following metrics: 1) downtime, which demonstrates the availability; 2) network delay, which reflects the client experience; and 3) overhead, which is measured with kernel compilation time. The experimental evaluations show that LLM sharply reduces the overhead of whole-system checkpointing and network delay on the client side compared with Remus. In addition, LLM demonstrates a downtime that is comparable, or even better for certain type of applications, to that of Remus. We also analyzed that LLM achieves transaction level consistency, which is the same as Remus.

The paper makes the following contributions:

1) We integrate the idea of input replay with whole-system checkpointing mechanism. Such an integrated effort outperforms the existing work with a single effort of checkpointing, especially for applications with intensive network workload;

2) LLM migrates the service requests from the primary machine independently, instead of depending on a special load balancer hardware. This means we can apply LLM more generally in practical use; and

3) We developed a fully functional prototype for LLM, which can be used as a basis for further research and practical application.

The rest of the paper is organized as follows. Related work is discussed in Section 2. In Section 3, we describe the system model and assumptions. We then introduce the design and implementation of LLM in Section 4. In Section 5, we report our experiment environment, benchmarks and the evaluation results. In Section 6, we finally conclude and discuss the future work.

2 Related Work

In [13], high availability is defined as a system design protocol and associated implementation that ensures a certain degree of operational continuity during a given mea-

surement period. Based on this definition, the availability may be estimated with a percentage of continuously operation time in a year, also known as "X nines" (for example "five nines", i.e., 99.999%) [6]. In fact, this percentage is determined by two factors—the number of failure events and the downtime of each failure event. We mainly consider the downtime at a failure in this paper.

State migration was studied in many literatures such as [14, 15]. For Xen [16], live migration was added in [7] to reduce the downtime thereby increasing the availability. However, this one-time migration effort is not suitable for disaster recovery, which requires frequent checkpointing protection. The wide-area live migration was also studied in [17]. But it is out of the scope of this paper. We only study the live migration locally in a cluster.

Checkpointing is a commonly used approach for fault tolerance. The idea of checkpointing was introduced to live migration by Cully *et al.* in Remus [10]. Remus is a remarkable effort on live migration, which aims to handle hardware fail-stop failure on a single host with whole-system migration. By checkpointing the status updates of a whole system, Remus can achieve generality, transparency, and seamless failure recovery. It is designed to use pipelined checkpoints, which means the active VM is bounded by short pauses, in each of which the state change is quickly migrated to the backup machine. Moreover, both memory and CPU state backup and network/disk buffering were carefully designed based on live migration [7] and Xen's intrinsic services. In general, Remus is a practical effort based on Xen, and most of its functions have already been merged into Xen.

In terms of shortcomings, though the downtime of Remus can be controlled within one second, it experiences about 50% performance penalty such as on network delay and CPU execution time. This penalty comes from data migration at high frequency—it supports up to 40 times of migration per second. If we decrease the frequency, the backup machine may serve a lot of service requests that have already been served by the primary machine.

Input replay is also a commonly studied approach for high availability. In [11], Bressoud *et al.* provided fault tolerance by forwarding the input events and deterministically replay them. Another example is ReVirt [18], in which VM logging and replay were discussed yet for intrusion analysis instead of high availability. However like Remus, these efforts also involve a single approach only. On the contrary, we integrate the efforts of both checkpointing and input replay to provide high availability with reduced overhead.

Xen [16] is an open source virtual machine monitor under the GPL2 license, which makes it very flexible for the purpose of research. We evaluated LLM on the platform of Xen.

3 System Model

In this paper, we discuss a whole-system replication. Therefore, for each primary machine, we assume there is a backup machine as the replica, and there is a high-speed network connection between the two machines. We also assume that the primary ma-

chine and the backup machine share a single storage, so that we do not have to migrate the whole file system.

We only consider hardware fail-stop failure model. Fail-stop failure makes the following assumptions: 1) any correct server can detect whether any other server has failed; and 2) every server employs a stable storage which reflects the last correct service state of the crashed server. This stable storage can be read by other servers, even if the owner of the storage has crashed.

We implemented LLM based on the existing codes of Remus, which was developed on Xen [16]. With Xen, network services run in guest virtual machines (called domain U or domU in Xen terminology). There is a unique VM for management purpose, called domain 0 or dom0, which has direct access to all physical hardware. Therefore service requests go through the back-end driver (called netback) in dom0 first, and then are distributed to the front-end driver (called netfront) in a specific domU. This will facilitate our network buffer management and migration.

We do not make any assumptions about the load balancer in a cluster, since a load balancer is a special hardware which is out of scope of this paper. This means LLM migrates the service requests from the primary machine independently. Moreover, since we do not consider the application-level migration, we do not distinguish the services running on a single guest virtual machine. All the migrated service requests are managed in the same manner.

Finally, we assume that there is an algorithm to map each egress response packet to a specific ingress request packet. A straight-forward approach is to append a sequence number for each ingress request packet and egress response packet, and keep this sequence number during the service. Hence, it is easy to pare up the two types of packets and map them accordingly.

4 Design and Implementation

We design the implementation architecture of LLM as shown in Figure 1. Beyond Remus, we also migrate the change in network driver buffers. The entire process works as follows:

1) First, on the primary machine, we setup the mapping between the ingress buffer and the egress buffer, signifying which packets are generated corresponding to which service request(s), and which requests are yet to be served. Moreover, LLM hooks a copy for each ingress service request.

2) Second, at each migration pause, LLM migrates the hooked copy as well as the boundary information to the backup machine asynchronously, using the same migration socket as the one used by Remus for CPU/memory status updates and writes to the file system.

3) Third, all the migrated service requests are buffered in a queue in the "merge" module. Those buffered requests that have been served will be removed based on the migrated boundary information. Once a failure occurs on the primary machine that breaks the migration data stream, the backup machine recovers the migrated memory image and merges the service requests into the corresponding driver buffers.

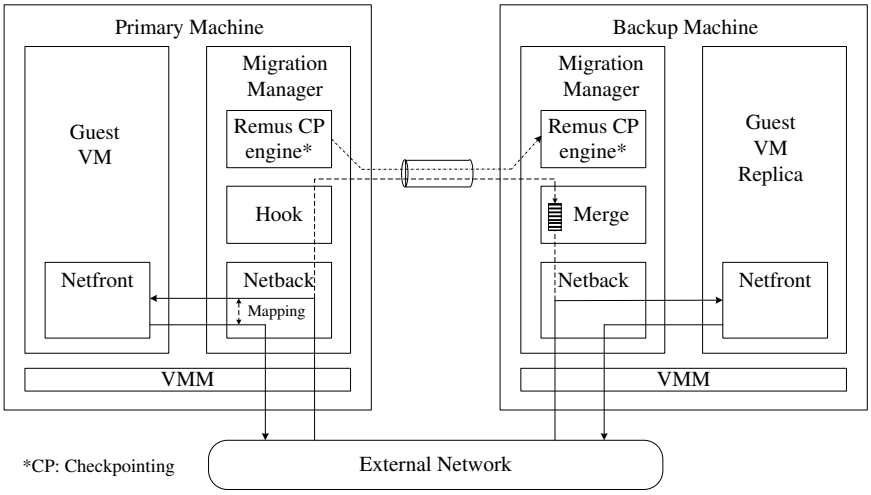

Fig. 1: LLM Architecture

In Figure 1, solid lines represent the regular input/output of network packets, dash-dotted lines show the migration of system status updates, and dashed lines mean the migration of network buffers.

Next, we will first introduce LLM's release of egress responses, analyze the consistency, then discuss the function modules in separate subsections: 1) egress response release and consistency analysis in Section 4.1; 2) mapping and hooking of services in Section 4.2; 3) asynchronous migration, especially its time sequence, in Section 4.3; and 4) buffering and merging of requests in Section 4.4.

4.1 Egress Response Release and Consistency Analysis

Unlike the block/commit case used by Remus, LLM releases the response packets immediately after they are generated on VMs. In the block/commit case, all the outputs are blocked using IMQ [19] until the migration in a checkpointing epoch is acknowledged. This can avoid losing any externally visible state, which helps to maintain the consistency. However, at low checkpointing frequency, network clients may experience very long delays with the block/commit mechanism of Remus.

In fact, the immediate release case and the block/commit case can achieve the same operation correctness on the client side, and the same consistency level on the server side. First on the client side: 1) for the block/commit case, there won't be any duplicated response packets. However, network clients may re-transmit requests after the timer expires; 2) for the immediate release case, on the contrary, there maybe duplicated response packets. Nevertheless, the re-transmission of service requests won't increment because of the increased delay. Given the fault tolerance of Internet, the duplication of either service requests or responses will be handled correctly. Therefore, both cases are correct and transparent to the clients.

Then on the server side, LLM achieves the same level of consistency, i.e., transactional consistency, as that of Remus. In Remus, all the service to request packets in the speculative execution phases will be re-transmitted by the client and re-served by the backup machine. In fact, this is equivalent to input replay, and the backup machine will have to recover the status at the failure point by replaying these re-transmitted requests. Though LLM migrates the requests directly, it recovers the status at the failure point in the same fashion. The only difference in consistency of LLM from Remus is the workload of input replay: as LLM usually runs at low frequency, it produces more requests to replay.

Therefore with the immediate release of egress response packets, LLM achieves the same operation correctness on the client side, and the same consistency level on the server side.

4.2 Hooking and Mapping of Service Requests

Without a special load balancer hardware, LLM has to migrate the service requests itself. Obviously, the first step of this migration is to make a copy.

Linux, which Xen is built on, provides a netfilter system consisting of a series of hooks in various points in a protocol stack. Figure 2 shows the netfilter system in the IPv4 protocol stack. This system makes it easy to copy or filter network packets to and from a specific guest VM [20].

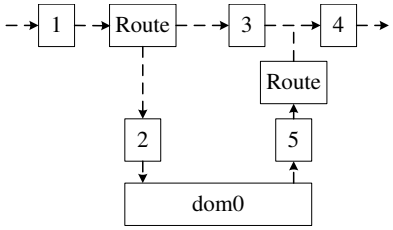

Fig. 2: Netfilter System

There is a netback driver in domain 0 of Xen, which is responsible for routing the packets to and from the guest VM. Domain "U"s are considered as external network devices. Thus, all the packets to and from guest VMs are routed through the path from 1 to 3 to 4 in the figure. Along this path, we choose point 3—NF_IP_FORWARD—to hook both ingress requests and egress responses.

We implemented this hook function module in two parts: 1) a hook module in the kernel that copies sk_buff and sends it up to the user space, and 2) a separate thread in the user space that receives copies and analyzes (for egress responses) or write them into the migration buffer (for ingress requests).

In Linux kernel, network packets are managed using sk_buff. The information in the sk_buff header is specific to the local host. Therefore we only copy the contents between the head pointer and the tail pointer. To recognize the packet header offsets

when the sk_buff header is absent, we append a metadata in front of the packet content, which includes the header offsets of each layer as well as the content length. This metadata will help the backup machine to create a new sk_buff header.

LLM manages a mapping table in the user space on the primary machine. For each hooked ingress request, we append an entry in the mapping table including 1) a sequence number, 2) a completion flag, and 3) a pointer to memory in the migration buffer. For each entry, the sequence number helps to distinguish requests from each other, and setting the completion flag as "True" means the service for this request has been completed. Then for each hooked egress response packet, we decide which request packet should be matched with using the algorithm that we assumed. Then we will set the completion flag of this request packet in the mapping table as appropriate.

4.3 Asynchronous Network Buffer Migration

Checkpointing was used to migrate the ever-changing updates of CPU/memory/disk to the backup machine by Remus. Only at the beginning of each checkpointing cycle, the migration occurs in a burst mode after the guest virtual machine resumes. Most of the time, there is no traffic flowing through the network connection between the primary machine and the backup machine. During this interval, we can migrate the service requests at higher frequency than that of checkpointing.

Like the migration of CPU/memory/disk updates, the migration of service requests is also in an asynchronous manner, i.e., the primary machine can resume its service without waiting for the acknowledgement from the backup machine.

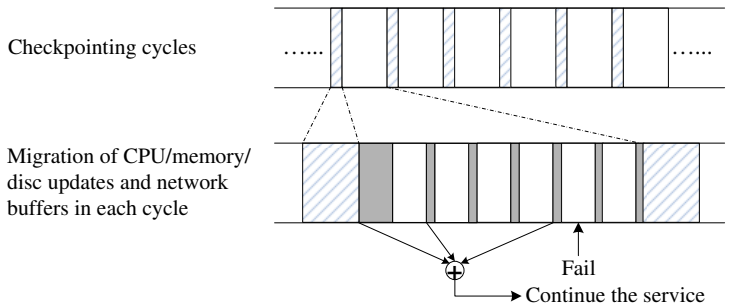

Fig. 3: Checkpointing Sequence

Figure 3 shows the time sequence of migrating the checkpointed resources and the incoming service requests at different frequencies on a single network socket. The entire sequence within an epoch is described as follows:

1) The dashed blocks represent the suspension period when the guest virtual machine is paused. During this suspension period, all the status updates of CPU/memory/disk are collected and stored in a migration buffer.

2) Once the guest VM is resumed, the content stored in the migration buffer is migrated first (shown as a block shaded area that is adjacent to the dashed area in the figure).

3) Then, the network buffer migration starts at high frequency until the guest VM is suspended again. At the end of each network buffer migration cycle (the thin, shaded strips in the figure), LLM transmits two boundary sequence numbers for the moment: one is for the first service request in the current checkpointing period, and the other is for the first service request that has a "False" completion flag. All the services after the first boundary need to be replayed on the backup machine for consistency, but only those after the second boundary need to be responded to the clients. If there is no new requests, LLM transmits the boundary sequence numbers only.

Anytime a failure happens to a primary machine, the backup machine will 1) continue the execution of VM from the latest checkpointed status; 2) replay the requests after the first boundary to achieve consistency; and 3) respond to those un-responded requests after the second boundary. This recovery process is shown with the "+" combination signal in the figure.

4.4 Buffering and Merging of Requests

The migrated service requests are first stored in a queue (implemented with a double linked list as shown in Figure 1) in the user space on the backup machine. With this double linked list, the storage can be allocated dynamically and the complexity of insertion at tail and removal from head will be constant.

Everytime when a network buffer migration burst arrives, the backup machine will first enqueue the incoming new requests at the tail, then dequeue and free requests from the head until the one with the first boundary sequence number. In this way, the queue only stores those needed to recover the system state. These requests will not be released to the kernel space until the migration data stream is broken.

As on the primary machine, there is also a module running in the kernel space on the backup machine. This module is responsible for creating a sk_buff header based on the metadata and inserting the requests into the queue in the kernel space. As shown in Figure 2, the requests will also be inserted to point 3, i.e., NF_IP_FORWARD. In this way, the protocol stack in the kernel will be able to recognize these migrated requests, just like local ingress request packets.

5 Evaluation

We evaluated LLM and compared it with Remus in terms of its correctness, downtime, delay for clients, and overhead under various checkpointing periods. The downtime is the primary factor to estimate the availability of a cluster, whereas network delay mainly represents clients' experience. Finally, the overhead must be considered in the picture so that the effectiveness of the service will not be overly compromised by checkpointing.

5.1 Experiment Environment

The hardware experiment environment included two machines (one as primary and the other as backup), each with an IA32 architecture processor and a 3 GB RAM. We set up a 100 Mbps network connection between the two machines specifically used for migration. In addition, we used a third PC as a network client to transmit service requests and examine the results based on responses.

As for the software environment, we built Xen from source which was downloaded from its unstable tree [21], and let all the protected virtual machines run PV (i.e., paravirtualization) guests with Linux 2.6.18. We also downloaded Remus version 0.9 codes from [22]. Then we allocated 256 MB RAM for each guest virtual machine, the file system of which is an image file of 3 GB shared by two machines using NFS (i.e., Network File System).

5.2 Benchmarks and Measurements

We utilized three network application examples to evaluate the downtime, network delay and overhead of LLM and Remus:

1) Example 1 (HighNet)—The first example is flood ping [23] with the interval of 0.01 second, and there is no significant computation task running on domain U. In this case, the network load will be extremely high, but the system updates are not significant. We named it "HighNet" to signify the intensity of network load.

2) Example 2 (HighSys)—In the second example, we designed a simple application to taint 200 pages (4 KB per page on our platform) per second, and there are no service requests from external clients. Therefore, this example involves a lot of computation workload on domain U. The name "HighSys" reflects its intensity on system updates.

3) Example 3 (Kernel Compilation)—We used kernel compilation as the third example which involves all the components in a system, including CPU/memory/disk updates. As part of Xen, we used Linux kernel 2.6.18 directly. Given the limited resource on domain U, we cut the configuration to a small subset in order to reduce the time required to run each experiment.

Here example 1 and 2 represent opposite types of network application, whereas example 3 is a typical application type entailing almost all aspects of system workload.

We measured the downtime and network delay under example 1 and 2, and the overhead under example 3. The details of each measurement are described below.

The downtime and network delay were measured using ping program on the client side, and the key index we selected here is the round-trip time of ping packets. We believe it makes more sense to measure on the client side since the client experience during a downtime is what actually matters. The flood ping used by example 1 is in itself a way of measurement. Yet for example 2, since it does not involve network activities, we have to use ping as an additional measure. To avoid the extra migration load, we increased the ping interval to 0.1 second and disabled the hooking function of LLM for ping packets. For each test case, we stopped the ping program after breaking the migration data stream. Then, in each ping program log file, we record the last peak value of the round-trip time as the downtime, since it represents the delay of the first

ping packet at the beginning of the disruption, therefore reflects the wait time of network clients. Lastly, we calculated the average value of round-trip times in a checkpoint period as the network delay.

Though there is no response for ping requests before the VM that fails is recovered completely, we are still able to guarantee that no ping packets are lost during the downtime. This is based on the configurable timer that ping program provides. As long as this timer does not expire, ping program will wait for the response (the transmission of following requests will not be influenced) without acknowledging a ping failure. In the experiments, we configured this timer long enough regarding the downtime that we may experience, so that each ping request will be responded, sooner or later. In this way, the response with the longest round-trip time could be used to estimate the downtime.

The overhead was measured using the incremental time (as a percentage) of kernel compilation. Specifically, the baseline, i.e., a 0% overhead, is the kernel compilation time without checkpointing, whereas a 100% overhead, for example, stands for a doubling of kernel compilation time when checkpointing exists. We measured kernel compilation time using a stop watch, so that it includes both the execution time and suspension time of domain U.

Finally, we measured the performance and the overhead under various checkpointing periods. For Remus, the checkpointing period is the time interval of system updates migration, whereas for LLM, the checkpointing period represents the interval of network buffer migration. By configuring the same value for the checkpointing frequency of Remus and the network buffer frequency of LLM, we are able to guarantee the fairness of the comparison to the greatest extent. Furthermore, we executed our experiments starting from one second of checkpointing period for two reasons. One is that the network connection specifically between the primary and backup machines has limited bandwidth in our experiment environment, thus will increase the migration time in each period. The other is that the timer in the existing migration implementation used by Remus and Xen is still under experiment. Therefore, at high checkpointing frequency, the actual checkpointing period is highly likely to exceed what we configured, thus it does not make sense to set checkpointing period of less than one second in the experiments.

5.3 Evaluation Results

First, we verified the correctness of LLM using two approaches:

1) We verified that the flood ping can be served continuously by the VM which is taken over by the backup machine after a failure occurs. Moreover, we carefully examined the sequence numbers, and observed that there was no disruption in the ping flood; and

2) We verified the results of kernel compilation by installing and booting the compiled kernel in domU successfully.

These two approaches can fully prove that LLM functions correctly after the migration.

Secondly, we measured the downtime under HighNet and HighSys, the results of which are shown in Figures 4 and 5.

We observe that under HighSys, LLM demonstrates a downtime that is longer than, yet comparable to, that of Remus. The reason is that LLM runs at low frequency, hence

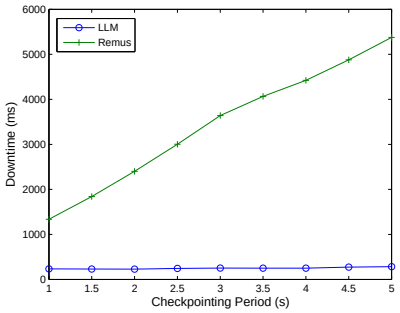

Fig. 4: Downtime under HighNet

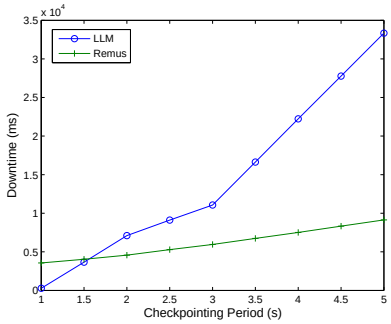

Fig. 5: Downtime under HighSys

the migration traffic in each period will be higher than that of Remus. Under HighNet, the downtime of LLM and Remus show a reverse relationship where LLM outperforms Remus. This is because, from the client side, there are too many duplicated packets to be served again by the backup machine in Remus. In LLM, on the contrary, the primary machine migrates the request packets as well as boundaries to the backup machine, i.e., only those packets yet to be served will be served by the backup. Thus the client does not need to re-transmit the requests, therefore will experience a much shorter downtime.

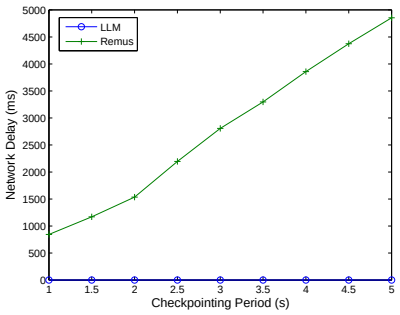

Fig. 6: Network Delay under HighNet

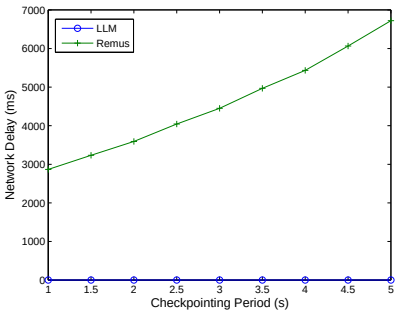

Fig. 7: Network Delay under HighSys

Thirdly, we evaluated the network delay under HighNet and HighSys as shown in Figures 6 and 7. In both cases, we observe that LLM significantly reduces the network delay by removing the egress queue management and releasing responses immediately.

In Figures 6 and 7, we only recorded the average network delay in a migration period. Next, we show the details of the network delay in a specific migration period in Figure 8, in which the interval between two adjacent peak values represents one migration period. We observe that the network delay of Remus decreases linearly within a period but remains at a plateau. In LLM, on the contrary, the network delay is very

high at the beginning of a period, then quickly decrease to nearly zero after a system update is over. Therefore, most of the time, LLM demonstrates a much shorter network delay than Remus.

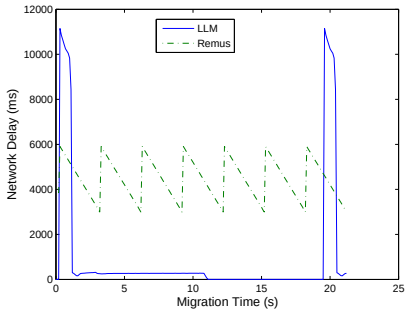

Fig. 8: Detailed Network Delay

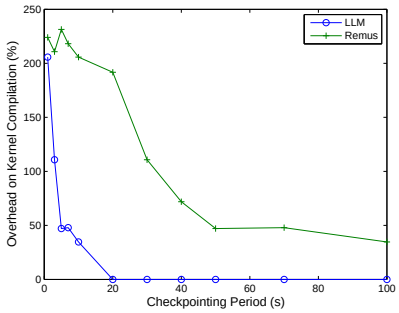

Fig. 9: Overhead under Kernel Compilation

Finally, Figure 9 shows the overhead under kernel compilation. Actually, the overhead significantly changes only in the checkpointing period interval of $[1,60]$ seconds, as shown in the figure. For checkpointing with shorter periods, the migration of system updates may last longer than a configured checkpointing period, therefore the kernel compilation time for these cases are almost the same with minor fluctuation. For checkpointing with longer periods, especially when it is longer than the baseline (i.e., kernel compilation without any checkpointing), a VM suspension may or may not occur during one compilation process. Therefore, the kernel compilation time will be very close to the baseline, meaning a zero percent overhead. Right in this interval, LLM's overhead due to the suspension of domain U is significantly lower than that of Remus, as it runs at much lower frequency than Remus.

In the experiments above (except for the specific sample shown in Figure 8), each data point was averaged from five sample values. The reason that we did not provide standard deviations is that the sample values remain very stable for given application examples. Generally the standard deviation is less than 5% of the mean value.

In summary, LLM clearly outperforms Remus in terms of network delay and overhead. For certain types of applications, LLM may also be a better alternative in terms of downtime than Remus.

6 Conclusions

In this paper, we designed an integrated live migration mechanism consisting of both whole-system checkpointing and input replay. LLM achieves transactional consistency, which is in the same level as Remus. From the experimental evaluations, we observed that LLM can successfully reduce the overhead of whole-system checkpointing and network delay on the client side while providing comparable downtime. Especially for

HighNet-like application types with intensive network workload, LLM demonstrates an even shorter downtime than the state-of-the-art efforts. As most services that require high availability usually involve a lot of network activities, LLM will be more efficient than other high availability approaches.

Finally, we want to provide some thoughts and clarifications for further discussion in this topic, namely, load balancer and multiple backup:

1) Load balancer—We did not depend on a special load balancer hardware to migrate the requests. If we do, a load balancer may duplicate the ingress request packets at the gateway, and distribute them to the primary and backup machines at the same time. In this way, what we need to migrate besides the system updates will simply be the boundary information. However, since the network buffer migration happens in the interval between system updates migration, the savings from the migration traffic may be negligible compared to the required investment on a load balancer.

2) Multiple backup—There are many scenarios for multiple backup, which involves the internal architecture of clusters. This is out of scope of this paper, and that is why we did not evaluate it in the experiment. If there is need to support multiple backup, we expect the changes to the prototype will not be significant.

The directions of future work include: 1) evaluate the impact of LLM on the consistency; and 2) compare LLM's performance and overhead in an environment with a load balancer.

References

1. Kopper, K.: The Linux Enterprise Cluster: build a highly available cluster with commodity hardware and free software. No Starch Press (2004)
2. Michael Armbrust, Armando Fox, R.G.A.D.J.R.H.K.A.K.G.L.D.A.P.A.R.I.S., Zaharia, M.: Above the clouds: A berkeley view of cloud computing. Technical report (2009)
3. Blake, V.: Five nines: A telecom myth. Communications Technology (2009)
4. Poledna, S.: Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism. Kluwer Academic Publishers (1996)
5. Mullender, S.: Distributed Systems. Addison Wesley Publishing Company (1993)
6. Carwardine, J.: Providing open architecture high availability solutions. HA forum (2005)
7. Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live migration of virtual machines. In: NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation, Berkeley, CA, USA, USENIX Association (2005) 273–286
8. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2) (2002) 51–59
9. Mergen, M.F., Uhlig, V., Krieger, O., Xenidis, J.: Virtualization for high-performance computing. SIGOPS Oper. Syst. Rev. **40**(2) (2006) 8–11
10. Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., Warfield, A.: Remus: high availability via asynchronous virtual machine replication. In: NSDI'08: Proceedings of the

5th USENIX Symposium on Networked Systems Design and Implementation, USENIX Association (2008) 161–174
11. Bressoud, T.C., Schneider, F.B.: Hypervisor-based fault tolerance. In: SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles, ACM (1995) 1–11
12. Aguilera, M.K., Spence, S., Veitch, A.: Olive: distributed point-in-time branching storage for real systems. In: NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation, Berkeley, CA, USA (2006) 27–27
13. Wikipedia: High availability, http://en.wikipedia.org/wiki/High_availability
14. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. In: SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data, ACM (1996) 173–182
15. Milojíčić, D.S., Douglis, F., Paindaveine, Y., Wheeler, R., Zhou, S.: Process migration. ACM Comput. Surv. **32**(3) (2000) 241–299
16. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, ACM (2003) 164–177
17. Bradford, R., Kotsovinos, E., Feldmann, A., Schiöberg, H.: Live wide-area migration of virtual machines including local persistent state. In: VEE '07: Proceedings of the 3rd international conference on Virtual execution environments, ACM (2007) 169–179
18. Dunlap, G.W., King, S.T., Cinar, S., Basrai, M.A., Chen, P.M.: Revirt: enabling intrusion analysis through virtual-machine logging and replay. SIGOPS Oper. Syst. Rev. **36**(SI) (2002) 211–224
19. MCHARDY, P.: Linux imq, http://www.linuximq.net/
20. Russell, R., Welte, H.: Linux netfilter hacking howto, http://www.iptables.org/documentation/HOWTO/netfilter-hacking-HOWTO.html
21. Xen Community: Xen unstable source, http://xenbits.xensource.com/xen-unstable.hg
22. Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., Warfield, A.: Remus source code, http://dsg.cs.ubc.ca/remus/
23. Wikipedia: Ping program, http://en.wikipedia.org/wiki/Ping