Strategies to Transparently Make a Centralized Service Highly-Available

Adrian Coleşa, Bogdan Marincaş, Iosif Ignat, Cosmin Ardelean
Computer Science Department
Technical University of Cluj-Napoca (UTCN), Romania
Email: adrian.colesa@cs.utcluj.ro, bmarincas@student.utcluj.ro, iosif.ignat,cosmin.ardelean@cs.utcluj.ro

***Abstract*—Services' high-availability has received great attention for years, yet it is not clear how this can achieved efficiently for different types of services. Current research focuses on finding strategies to provide high-availability using commodity hardware and as independent of the service and transparent to the clients as possible.**

This paper investigates strategies to provide high-availability for centralized services, in a transparent way to the service itself. We describe a replication system that provides fault-tolerance for any type of centralized service. Our system runs each service instance into a virtual machine and dynamically replicates the entire virtual machine.

The testing prototype was implemented using the Xen hypervisor. It proved functional, totally transparent to service and clients and efficient.

***Keywords*-High-Availability, Replication, Fault-Tolerance, Scalability, Virtualization**

I. Introduction

High-availability [1] is an important requirement of many public services provided today. Therefore, it received great research interest over the years. The main technique used to get high-availability is replication [2]. It provides the basis for fault-tolerance and scalability.

Most existing solutions are based on specialized hardware and require the service to be designed for a distributed environment. Research has focused on finding solutions requiring minimal contribution of the service in all the complex distributed mechanisms needed to provide high-availability. That resulted in specialized frameworks [1] over which the services can be easily implemented. Such solutions require the service to be specifically designed to fit into the framework, so they do not provide total transparency to the service.

Other methods concentrate on providing high-availability as transparent as possible to the service [3], but they are generally developed for specific applications like Web servers, whose client communication protocol is known. In order to provide transparency to the clients, they require service's collaboration, so they impose special design requirements on the service. That way however they support active replication which accounts for service scalability.

Total transparency to both clients and service itself is obtained by systems like Remus [4] and Kemari [5], which run the service in a virtual machine and replicate the entire virtual machine to provide fault-tolerance [6]. Their method supports only the primary-backup configuration, which can provide for fault-tolerance, but not for load-balancing.

Our work is aimed at finding strategies to provide high-availability to existing centralized services requiring no modification of them. We want to make the system transparent for the service itself and as general as possible. Our methods could be used for legacy services that are impossible or too expensive to be reimplemented as distributed ones.

The two techniques we tried to combine in order to achieve high-availability are replication and load-balancing. We used replication of virtual machines in a primary-backup configuration. In order to extend this method to also support active-replication and consequently load-balancing, we note that this is possible only for specific types of services. We propose a replication strategy to make such services highly-available.

The testing prototype of our replication system was implemented using the virtualization and live migration mechanisms provided by the Xen hypervisor [7]–[9]. The tests we performed proved our method's correctness and efficiency.

II. Transparent Replication of a Centralized Service Using Virtualization

In order to make a centralized service highly-available we considered two techniques: *replication* and *load-balancing*.

There are two classical ways of implementing replication [2]: *primary-backup* and *active-replication*. The primary-backup configuration supports fault-tolerance, but not load-balancing. Active-replication supports load-balancing, but only for read requests, requiring complex synchronization mechanisms in the case of update requests. Active-replication can also hardly cope with update requests having nondeterministic effect, while primary-backup inherently manages them successfully.

Thus, read requests can be balanced and run simultaneously in an active-replication configuration, while update requests can be executed only using a lock-step mechanism in a primary-backup configuration, allowing just one update to take place at one moment.

One of the goals we established for our research was to make a centralized service running in a distributed way to provide fault-tolerance (by replication) and scalability (by load-balancing) totally transparent to the service itself.

978-1-4244-5007-7/09/$26.00 ©2009 IEEE

Having no support from the service's application level, our replication system must deal itself with anything that normally implies service's actions, e.g. make distinction between read and update requests and replicate changes.

We considered the primary-backup strategy as the single one providing total transparency to the service. Load-balancing can be used in the primary-backup configuration only when some details about the service are known in advance. For example, services handling only read requests or services for which the type of requests can be identified by the replication system before being handled by the service.

The state of the service's primary instance that is replicated on the other nodes as backup instances (replicas) must include not only the primary's memory space, but also all its corresponding resources and data structures the operating system it runs on maintains for it. For example, the current network connections between the primary instance and the clients it serves must be replicated.

Even if we consider that the primary replica's in-memory state does not change, the operating system's data structures do change due to new requests associated with network connections. So, in order for a backup replica to replace a failed primary it is not sufficient just to have the same in-memory state but also the same operating system state.

The technique we used to replicate the state of the service's primary instance and that of the operating system it runs on is called *replication using virtualization*. We place our service's primary instance in a virtual machine on a (primary) node and entirely replicate that virtual machine on other (backup) nodes.

Replication using virtualization makes our system totally independent from the service it replicates and also from the operating system that service runs on. It provides us with the mechanism to run a centralized service in a distributed system creating more instances (replicas) of that service, keeping them consistent with each other and, therefore, making the original service tolerant to node failures. Note that this replication schema considers only the situations where just one primary is running at one moment, but load-balancing cannot be used in such a configuration.

Load-balancing can be used only in a configuration where each service's instance has its state replicated. This seems to bring us back to the primary-backup configuration, which cannot be used for load-balancing. Still, this can be done, for services that do not change their state during request handling. The replication schema for such services is applied in parallel on each primary-backup tuple. Such a configuration is illustrated in Figure 1.

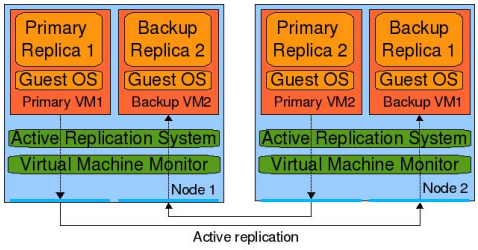

Figure 1. Active Replication of Multiple Tuples of Primary-Backups

III. Active Replication of Virtual Machines

Replicating a virtual machine can be obtained in two ways.

The first one starts with two virtual machines which are suspended and in the same state, i.e. identical. The virtual machine we are replicating is activated for a period of time before it is again suspended. During its running period all of its nondeterminism is recorded. Starting the other virtual machine and playing back the recorded nondeterminism at the appropriate moments will yield an identical virtual machine. It is important to note that the outputs from the replayed machine will be blocked. Recording and replaying is troublesome when dealing with multi-threaded processes and possible interactions between processes and is constrained to one processor, as there is the difficulty of handling the SMP multiprocessor computer architecture.

In the second method the primary is the only machine allowed to run. After the primary was active for a period of time it is suspended and its state is captured, that is, a checkpoint is created, and is copied to the backup. To maximize efficiency with each new round only the differences, mostly modified memory pages, from the previous checkpoint are transferred. Although this technique is highly dependent on the locality memory modifications and therefore on network throughput, it is the preferred one as it offers a higher degree of generality than the first method.

The architecture we are proposing relies on the Xen Hypervisor [7], [8] (Virtual Machine Monitor). Xen's architecture has two types of guests (also called domains): privileged (`Dom0`) and underprivileged (`DomU`). While multiple underprivileged domains are allowed, only one privileged domain can exist. The privileged domain is used both for control and access to hardware, through its drivers. Our replication system runs in `Dom0` and protects a `DomU` guest.

The term replication we are going to use from now on refers to the active process of maintaining an exact copy (replica) of a virtual machine (`DomU` guest) and its resources in the context of another Xen hypervisor, on another physical machine, over the network. We will also be using the terms primary and backup to refer to the entire virtual machine we are copying and its copy.

Our replication system allows transparent recovery from failure, meeting the following constraints:

- No output is released until the protected virtual machine is replicated.
- The time needed to detect and recover from failure is small, a reasonable value being 1 second.
- Failure of either the primary or the backup is tolerated.

The whole virtual machine replication strategy consists of the following steps:

1) Initially copy the whole state of the primary to the backup.
2) Run the primary for a certain period of time, blocking but buffering its network output.
3) Pause the primary and send to the backup the differences between this and previously transferred state.
4) Resume the execution of the primary and release its formerly blocked output, once the backup confirms that differences between states have been received.

The whole state of the virtual machine is composed of memory, CPU and local hard disks it uses. The majority of the differences between two states comes from the memory pages that were written while the virtual machine was running and local hard disk writes. The memory pages and the CPU's state are addressed as follows.

The actual process is an extension of a feature provided by Xen, namely live migration [9], which allows a virtual machine to be moved to another Xen hypervisor while the machine is still running with little or no interruption. This is achieved by transferring the state of its memory and CPU in rounds. With each new round, pages that have been modified by the active virtual machine are sent. This is done until the set of pages to be transmitted is modified far too quickly for the process to end. At this point the machine is suspended and, in this final round, the remaining pages and the CPU state are sent.

In essence our process is an uninterruptible live migration, which starts with a whole memory and CPU state transfer and subsequently repeats the final round.

The changed pages are detected with the use of Xen's *shadow page tables*. When this is activate, Xen marks the guest's pages as read-only and propagates any write attempt to the shadow page table, before allowing the actual write.

Network output buffering is done by implementing a new Linux queuing discipline attached to the virtual network interface of the protected guest in `Dom0`.

Hard disk consistency is currently maintained in our system with the help of DRBD [10] a distributed and replicated block device.

In order to meet the second constraint of transparency, primary's failure must be detected rather quickly. Therefore we deployed a heartbeat client-server configuration, which is sensible enough to withstand network load, but promptly reports a timeout in case the primary fails. Furhermore, during the replication process the backup commits the state once it has all been received. An accumulation of states would be wasteful both in terms of memory and recovery time. It follows that, once failure is detected, the backup will be restored to the last committed state quite fast. Also, after restoring, a gracious ARP packet will be sent in order to notify the network devices of the change.

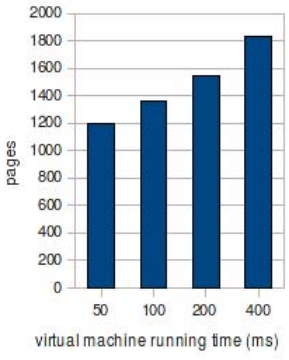

Figure 2. Average dirtied pages

In case the backup fails, the synchronization process will be interrupted, but the primary will continue to run, albeit without being protected.

IV. Tests and Results

All tests we performed on our replication system were run on a pair of computers with an AMD Athlon 2000 XP processor, 1GB of RAM, 40GB of hard disk space and a 100Mbit Ethernet interface. The protected virtual machine was allocated 64MB of RAM. During the synchronization the protected guest was run for variable amounts of time before checkpointing and transmitting its state.

First, in order to prove the correctness of our implementation, we established a SSH connection to the protected guest and severed the connection between the primary and the client by disconnecting the ethernet cable from primary's network interface. The SSH connection between the client and the backup was reestablished in under one second.

Our first performance test was the compilation of Busybox's source code. We felt that this test would best represent real world utilization of hard disk, memory and CPU resources.

Figure 2 informs us that the number of dirtied memory pages increases if the virtual machine is active for a longer period of time. In this test the number of dirty pages does not increase linearly with the running time, so one gains computational time by not having to transfer many pages on each iteration.

The iteration time in Figure 3 is the total time needed to run a virtual machine, suspend it, checkpoint its state, and resume the virtual machine. It can be seen that the time needed to checkpoint, that is, capture the state and send it over the network, i.e. replicate it, is considerably larger then the running time. This is in part due to the increased number of pages, as it was seen in Figure 2. However, it is clear that the more frequent each iteration is, the larger the resume time, which can even be greater than the actual running time.

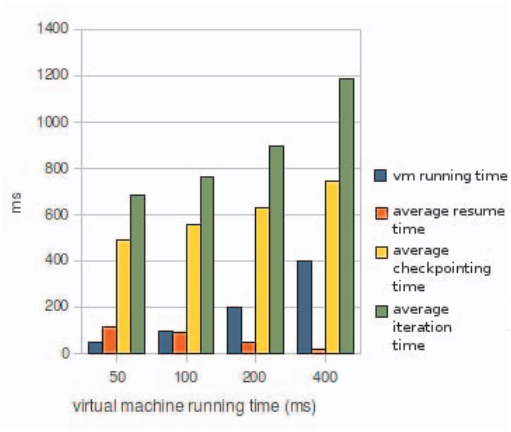

Figure 3. Average times that compose an iteration

It is important to note that the average iteration time is also a good measurement of the average network delay clients are expected to experience.

V. Related Work

Replication as a method to provide fault-tolerance and its possible configurations — primary-backup and active replication — are described in [2]. Systems like [1] propose different frameworks, services can be built on, in order to attain high-availability transparently to clients. They do not provide however transparency to the service.

Problems encountered when deploying in practice highly-available services are analyzed in [3]. They use a record-and-replay technique to transparently mask service replicas crashes from clients. Their system supports active replication and faces with random replicas' states based on a synchronization strategy implemented in the service itself, a requirement that our system cannot impose.

Virtualization as a method to provide fault-tolerance was first introduced in [6]. It was used in [4], [5], [9] to transparently run centralized serviced in a distributed manner to mask replicas' crashes from clients. Their solutions are based on the primary-backup configuration, which successfully deals with nondeterminism in the service's state, but does not support load-balancing. We proposed a way to avoid this limitation for specific types of services.

VI. Conclusion

We explored a replication strategy to transparently make a centralized service fault-tolerant to node crashes and scalable, i.e. highly-available.

Transparency is achieved by placing the service in a virtual machine. Fault-tolerance is provided by replicating that virtual machine, based on the primary-backup strategy. Scalability is obtained by load-balancing, proposing a replication schema with more primary-backups tuples, but it functions only for specific types of services or client requests.

The testing prototype we implemented over the Xen hypervisor proved our replication system's correctness. Its efficiency is closely dependent on those of the network link between primary an backups, since the state's transfer time has the greatest weight in the total replication time.

Future work will aim to maximize the running time of the primary while reducing the amount of data transfered during state replication. We also want to investigate further strategies to apply load-balancing for other types of services.

References

[1] A. Fekete and I. Keidar, "A general framework for highly available services based on group communication," in *Distributed Computing Systems Workshop, 2001 International Conference on*, 2001, pp. 57–62.

[2] R. Guerraoui and A. Schiper, "Fault-tolerance by replication in distributed systems," in *Reliable Software Technologies - Ada-Europe'96*. Springer-Verlag, 1996, pp. 38–57.

[3] M. Marwah, S. Mishra, and C. Fetzer, "Enhanced server fault-tolerance for improved user experience," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, 2008, pp. 167–176.

[4] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "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, pp. 161–174.

[5] Y. Tamura, "Kemari: Virtual machine synchronization for fault tolerance using domt," June 2008.

[6] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," in *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, vol. 29, no. 5. ACM Press, December 1995, pp. 1–11.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM Press, 2003, pp. 164–177.

[8] K. Fraser, H. Steven, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe hardware access with the xen virtual machine monitor," in *In Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[9] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*. USENIX Association, 2005, pp. 273–286.

[10] L. Ellenberg, "Drbd 9 and device-mapper," in *In Proceedings of the 15th International Linux System Technology Conference*, October 2008.