Availability in the KVM Virtualization Environment

Paper #10

Abstract

By allowing multiple servers to be consolidated on a smaller number of physical hosts, virtualization is widely used in computing environments of various kinds and scales.

In this paper we present an high-availability (HA) solution for the applications running in the virtual machines. In our solution, the key to achieving high availability efficiently is to leverage an State Machine Replication system in the network layer which replicates programs using RDMA. Evaluation on four widely used server programs (e.g., `MySQL` and Redis) shows that this solution is easy to use and has low overhead.

1 Introduction

Virtualization is used to create one or more Virtual Machines (VMs) that act like real machines in a physical host. It facilitates flexible partitioning and dynamic allocation of computing resources, and is widely used in computing environments of various kinds and scales.

In a virtualized environment, applications run in VMs, and multiple VMs may be consolidated in a single physical server. Server consolidation has in fact been the most common reason for using virtualization [3]. It is most useful when applications require a certain of isolation, e.g., isolation of configurations, faults, and so on, yet each of them does not need the full capacity of a single server. Running these applications in separate VMs on a single physical server enhances server utilization and reduces various operational costs, including management cost, power, space, etc.

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

As host failures are inevitable, even common in large systems [1, 4], maintaining highly available VMs despite the occurrence of host failure has become a crucial task.

A common approach to implementing fault-tolerance is replication, where two or more replicas keep nearly identical state at all times. If one replica fails, others can continue, and in such a way that the failure is hidden to external clients and no data is lost.

There are currently two main approaches for replication. The first one is state machine replication (SMR). SMR models the server programs as deterministic state machines that are kept in sync by starting them from the same initial state and ensuring the same input requests in the same order. The second method for replication is referred to as the state transfer approach. It ships changes to all state of the primary, including CPU, memory, and I/O devices, to the backup nearly continuously.

However, since server programs have some operations that are not deterministic, SMR is complex to get right. On the other hand, in state transfer approach the bandwidth needed to send state, particular changes in memory, can be very large.

To tackle nondeterminism, we incorporate the lightweight state transfer.

2 FALCON Background

FALCON's deployment model is similar to a typical SMR's. In a FALCON-replicated system, a set of $2f + 1$ machines (nodes) are set up in a InfiniBand cluster. Once the FALCON system starts, one node becomes the *primary* node which proposes the order of requests to execute, and the others become backup nodes which follow the primarys proposals. An arbitrary number of clients in LAN or WAN send network requests to the primary and get responses. If failures occur, the nodes run a leader election to elect a new leader and continue.

On receiving a client network request, it invokes a RDMA-based consensus process on this request to enforce that all replicas see the same sequence of input requests. This process has three steps. In the first step, the leader assigns a global, monotonically increasing viewstamp to this request, stores this request into an entry that is appended to the consensus log, and does a forced write to the local disk. The second step is to replicate the log entry on remote servers using a one-sided RDMA Write operation. Usually when the RDMA NIC (RNIC) completes the network steps associated with the RDMA operation, it pushes a completion event to the queue pair's associated completion queue (CQ) via a

Submitted to APSys '16, August 4-5, 2016, Hong Kong, China.

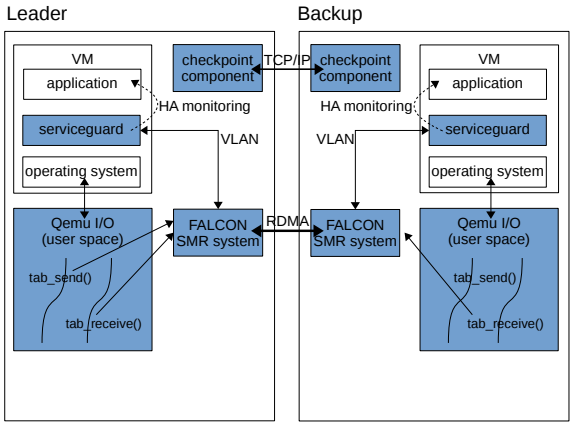

Figure 1: ***Proposed architecture in KVM virtualization environment.*** **Key components are shaded (and in blue).**

DMA write. Using completion events adds extra overhead. Since PAXOS could help handle the reliability issues, FALCON takes advantage of unsignaled RDMA write operations, i.e., a completion event will not be pushed for these operations, to reduce that overhead. In the last step, the leader thread waits for acknowledgments from a majority of nodes.

In addition to the distributed consensus protocol for coordinating the sequence of input requests, FALCON also runs an output checking protocol which compares each replica's network outputs occasionally to detect the divergence of execution states.

3 Design of Architecture

We chose the Linux Kernel Virtual Machine (KVM) [2] as the platform of our study. KVM takes advantage of the hardware virtualization extensions so that it achieves nearlythe same performance with the underlying physical machine. A major advantage of the KVM architecture is the full availability of user-space tools in the QEMU process, such as threading, libraries and so on.

Figure 1 shows our proposed architecture in KVM virtualization environment. Only the primary VM advertises its presence on the network, so all network inputs come to the primary VM.

It contains three main components: the Serviceguard, a tailored version of QEMU, and the SMR system FALCON.

The Serviceguard is deployed on the virtual machine, and provides monitoring capabilities for applications running inside virtual machines in the KVM virtualization environment.

4 Input Consensus

In order to provide the server programs running inside the virtual machine the exact same inputs in the same order, we interpose on the `tap_send()` function in QEMU and invoke the PAXOS consensus component of FALCON on the received networking packets. Once the consensus process is finished, QEMU continues with its normal execution.

5 Checking Network Outputs

This section presents XXX's network outputs checking mechanism for detecting and recovering from replicas' divergence of execution states.

To capture network outputs of the server programs running in the virtual machines, we modified the `tap_receive()` function in QEMU to maintain a packet queue for each application that captures the corresponding outgoing packets. The network outputs are pushed into the queue and whenever the queue is full, a new hash value is calculated by $h_i = H(h_{i-1}||H(queue))$ where $H()$ is a hash function and $||$ stands for concatenation. Such a computation links the hash value to all the previous network outputs. Then, after every T_{comp} hash values are generated, the latest one is passed to FALCON and the output checking protocol is invoked. The index of this hash value in the hash chain is consistent across replicas because each replica implements the same mechanism.

Once divergence of execution states is detected, we do a light-weight state transfer to handle it. For memory states, we make use of the *dirty page log* facility provided by KVM. The KVM kernel subsystem interfaces *KVM_SET_USER_MEMORY_REGION* and *KVM_GET_DIRTY_LOG* allow the caller to keep track of dirty page state changes for the given virtual machine. KVM_SET_USER_MEMORY_REGION allows the user to create or modify a guest physical memory slot, and KVM_GET_DIRTY_LOG returns a bitmap with all dirty pages since the last call. With these APIs, we can compare dirty memory pages and sync the different ones only.

6 Leadership Transfer

In some cases, one or more servers may be more suitable to lead the cluster than others. For example, a server with high load would not make a good leader, or in a WAN deplolyment, servers in a primary datacenter may be preferred in order to optimize the latency between clients and the leader. Since FALCON needs a server with a sufficiently up-to-date log to become leader, which might not be the most preferred one. Instead, a leader in XXX can periodically check to see whether one of its available followers would be more suitable, and if so, transfer its leadership to that server. To transfer leadership in XXX, the prior leader sends its log entries to the target server, then the target server runs an election without waiting for an election timeout to elapse. The following steps describe the process in more detail:

1. The prior leader stops accepting new client requests.

2. The prior leader fully updates the target server's log to match its own, using the normal log replication mechanism in FALCON.

3. The prior leader sends a *TimeoutNow* request to the target server. This request has the same effect as the target server's election time firing: the target server starts a new election.

Once the target server receives the *TimeoutNow* request, it is highly likely to start an election before any other server and become leader. At this point, leadership transfer is complete.

7 Evaluation

8 Related Work

9 Conclusion

We have presented the design of XXX.

References

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

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

[3] R. McDougall and J. Anderson. Virtualization performance: perspectives and challenges ahead. *ACM SIGOPS Operating Systems Review*, 44(4):40–56, 2010.

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