2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing

Improving The Responsiveness of Replicated Virtualized Services in Case of Overloaded Replicas Connectivity

Adrian Coleşa
Computer Science Department
Technical University of Cluj-Napoca, Romania
Email: adrian.colesa@cs.utcluj.ro

Ioan Stan
Computer Science Department
Technical University of Cluj-Napoca, Romania
Email: stanioan87@gmail.com

***Abstract*—One approach to provide high availability for services is to encapsulate the service in a virtual machine and replicate, in checkpoints, the entire machine.**

The replication protocol can overload the network link between the replicas, fact that will increase the latencies experienced by the clients of the service.

We implemented a context-aware replication protocol that adapts the length of a checkpoint phase in order to get low values for the latencies experienced by the service's clients. The length of a checkpoint phase is obtained by using predicted parameters that describe the service behavior.

The results show us that the latencies experienced by the service's clients were reduced, especially in case the network link between replicas was overloaded.

Our replication protocol extends Remus replication protocol, which is provided with Xen Hypervisor, by improving its asynchronous strategy and by taking into account the context of the replicated service.

***Keywords*-asynchronous, context-aware, high availability, prediction, replication, service**

I. Introduction

The service responsiveness represents the capacity of a service to ensure client requests, according to specifications, during normal and abnormal functioning. The abnormal functioning can be determined for example by electric power outages, hardware dis-functionalities, software attacks [1], [2] or resources overloading.

Service high availability requirement emerges from the fact that its clients need a permanent access to the service that fulfill their specifications. The unavailability of critical services would have a negative impact for their clients. This is the reason the research in this field has received a great attention in the latest two decades [3], [4].

Most proposed solutions require design changes of the service and the protection system, because they are based on special properties of the service they support [5], [6], [7]. Being integrated within the service, the main advantage of such solutions is their efficiency. Their main problem though is that any change in the service's properties leads to the redesign and reimplementation of the whole system in order to maintain initial requirements. Also, they cannot be applied to services that cannot be reimplemented.

Other solutions try to provide high availability independently of the service. Some of them [8], [9] are capturing the state of the service at the application level and are based on the record-and-replay technique. They are dependent on the operating system the service is running on and do not support nondeterministic behavior of the service.

In order to avoid such drawbacks the service to be protected was encapsulated in a virtual machine and the entire virtual machine replicated. Such a strategy can be applied to any service and provides transparency for both service's clients and the service itself. In [10] a synchronous replication strategy is presented. An improvement to this strategy is provided in [11], where the states are transfered asynchronously from other operations of the replication. These two strategies have the drawback of not controlling the replication of virtual machine's disks states. A complete replication protocol was presented in Remus [12], where, in order to have an asynchronous strategy they used the speculative execution approach, in which the transfer operation can serialize the replication protocol in case the bandwidth is overloaded.

The generality advantage of the service virtualization results in the solution not being efficient for any type of service. Also, the existing solutions do not adapt to the service's specific properties or running behavior, nor even to other environment characteristics. In order to improve the replication protocol in [13] is proposed an adaptive replication protocol implemented on the base of [11], which is not fully complete because it doesn't control disk replication.

In this paper we describe a complete adaptive replication protocol. We improved the adaptive replication algorithm proposed in [13] by integrating disk replication with memory replication and queuing discipline. We introduced the resulting algorithm in the Xen hypervisor 4.1.2, by changing the Remus [12] replication mechanism, obtaining the following benefits: firstly, we improved Remus, making it adaptive and avoiding serialization induced by the speculative execution in case of overloaded replicas connectivity and, secondly, take the advantage of the optimizations introduced by Remus in Xen, like reduced buffering time and avoidance of VM suspension during phase evaluation.

978-0-7695-4934-7/12 $26.00 © 2012 IEEE
DOI 10.1109/SYNASC.2012.13

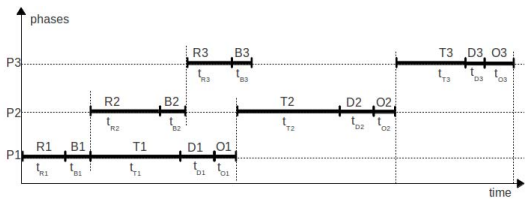

Figure 1. Phases and Steps of the Asynchronous Replication Strategy

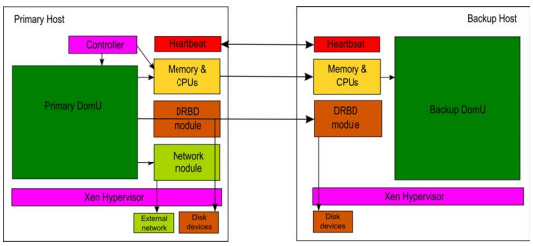

Figure 3. System Design

We analyzed, both theoretically and practically, the behavior of the resulting replication protocol on different kind of real services (a Web server and an intensive resource usage server) in different scenarios. The analyze and experimental results proved that the protocol adapts efficiently to the service behavior and improves the responsiveness of the replicated and virtualized service in case the connection between replicas becomes overloaded.

II. Strategies to Protect a Virtualized Service

This section briefly presents the replication strategy and the queuing protocol used to control network outputs and disk operations on primary and on backup, respectively.

A. Replication Strategy

The replication protocol we use is based on the adaptive one presented in [13], but improves it to also control the disk operation replication. It replicates asynchronously the virtual machine (VM) that runs on a *primary* to a *backup* host.

The replication takes place in phases, each one consisting in the following steps, illustrated in Figure 1: (*1*) *running* (R_i), lasting t_{R_i}, during which the primary VM is running, accepting inputs, but having its network outputs blocked, (*2*) *buffering* (B_i), lasting t_{B_i}, when the VM is suspended and its state buffered, (*3*) *transfer* (T_i), lasting t_T, when the previously buffered VM's state is transfered from the primary host on the backup one, (*4*) *disk writes release* (D_i), lasting t_{D_i} and corresponding to the act of applying the write operations that occurred on the primary VM's disks to the backup VM's disks, (*5*) *output release* (O_i), lasting t_{O_i} and corresponding to the act of releasing the outputs of the primary VM corresponding to its already replicated state.

As can be observed in Figure 1, during the replication, the transfer, disk writes release and network output release steps corresponding to previous phases run in parallel with running and buffering steps of newer phases.

The running step in different phases has different lengths, because it is adapted to the behavior of the virtualized running service. Also, during the running step, the service behavior is analyzed without suspending the VM, and is determined if a phase will continue or not.

We can note that disk writes are replicated when they happen, but are buffered on the backup and applied only after the corresponding memory and CPUs states were replicated, i.e. at the end of the transfer step.

B. Queuing Strategy for Disk Operations and Network Outputs

During the replication of a phase we need to ensure consistency between memory, CPUs, disks and network states. This means that inside the asynchronous transfer of a phase we need to synchronize the state transfer of different kind of devices.

Similar to the replication of disk operations are treated the network output packets that correspond to a phase: they are buffered on the primary host until the memory and CPUs states are replicated. Therefore the same buffering strategy (for now on named *queuing strategy*) will be used for both disk writes and network outputs.

In Figure 2 is represented the queuing strategy which is described below:

- When the VM starts to run during a phase $Phase_i$, a barrier B_i is added at the end of the queue.
- All the packets p_i that arrive will be added to the queue after this barrier.
- When the memory and CPUs states that correspond to a phase were replicated to the backup, the first barrier is removed and the queued packets, until next barrier, are released.
- We can be sure that the released packets corresponds to the replicated memory and CPUs states because these states are buffered on the primary and transfered then on the backup in the same order the packets are enqueued.

We use two such queues during replication: one for disk writes on the backup and one for network outputs on the primary. In order to release network outputs that correspond to complete replicated VM's state of a phase, we first remove the barrier on the backup queue and apply disk writes and then, remove the barrier that blocks network outputs on the primary.

III. Protection System's Design

The state of the VM that encapsulates the service and runs on the primary host is replicated to a backup host when the adaptive replication protocol decides, based on the

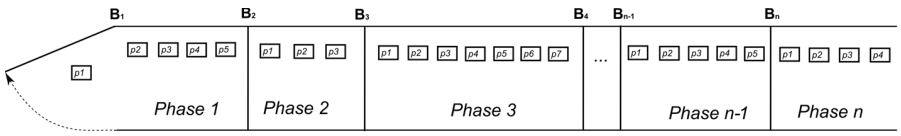

Figure 2. Queuing Strategy used to buffer network output packets and disk writes

service behavior regarding the changed pages of memory and number of output network packages.

Our replication system, shown in Figure 3, is integrated into the Remus replication mechanism provided with the Xen Hypervisor. It improves the adaptive replication protocol in [13] by taking into account disk states during replication. For some kind of services it is more efficient than Remus implementation, because it adapts the running time based on the service behavior. In this section we describe the modules that were used to design a system proved to be complete and efficient for different kind of services or different scenarios.

The *Controller* contains the implementation of the adaptive protocol and, based on analysis of the service behavior, decides when is better to continue a phase and when is efficient to stop it and start a new one.

After the VM is running for a period of time in a phase, it is suspended and its memory and CPUs states are buffered. The *Memory & CPUs* module represent the buffer and the operations on buffer. The state of the buffer is used during analysis process in the *Controller*.

The *DRBD module* is a modified version of Distributed Replicated Block Device tool according to our needs. It is used to replicate disk writes from the primary VM to the backup one and buffer them there until the corresponding memory and CPUs states are replicated.

For buffering network outputs on the primary VM, until the corresponding state is replicated, the *Network module* from Figure 3 implements a Linux queuing discipline based on the queuing strategy presented above.

In case of a primary host failure the *Heartbeat* module detects it and the VM from the backup host will take place of its counter part from the primary host. The failure will be transparent to the clients of the virtualized service because the released output will correspond to phases that were replicated on the backup host.

The adaptive replication protocol tries to adapt the running time of the VM in order to increase the responsiveness of the virtualized service both during replication and in case of failure, especially when the connectivity between hosts becomes overloaded.

IV. Analysis of The Adaptive Replication Protocol in Different Replication Scenarios

Using the algorithms and relations presented in [13] we will analyze different scenarios for the adaptive replication protocol in order know for which kind of services and in which contexts it is efficient. Bellow we will show the algorithms from [13], describe the conditions used in the algorithms and analyze exceptional and normal scenarios in order to know if a phase continues or not.

First algorithm is run in case the client requirements are met by the service and the second one runs in case of degraded functioning of the service.

```
   task VM_replication;
   function normal_functionality () : boolean
       t_chk ← τ;
C2     if "it is possible to exceed D_max^req" then
           return TRUE;
       else
C3         if "the state buffer is full" then
               return TRUE;
           else
C4             if "required η_min provided" then
C5                 if "it is more efficient extending the
                   current phase than starting the next
                   one" then
                       return FALSE;
                   else
                       return TRUE;
                   end
               else
                   return FALSE;
               end
           end
       end
```

Algorithm 1: Replication algorithm in case of *normal functionality*

The following relations and conditions are used during the replication protocol.

CPU usage efficiency is denoted by η and defined by the relation:

$$\eta = \frac{t_R}{t_R + t_B} \tag{1}$$

Maximum estimated delay introduced by our protocol in case the service need *n* phases to handle the client request depends on the buffering time (t_B), transfer time (t_T) and running time (t_R).

$$D_{max} = (n+2)t_B + t_T + t_R \tag{2}$$

During the replication protocol, in order to know which algorithm to run (i.e. normal or degraded), we check if the

```
task VM_replication;
function degraded_functionality () : boolean
    t_chk ← τ;
C6  if "there is available bandwidth" then
        return TRUE;
    else
C3      if "state buffer is full" then
C7          if "it is more efficient extending the current
            phase than waiting for saving space in state
            buffer and starting the next phase" then
                t_wait ← max(0,size(Δ_i)+size(state_buf)−MAX_BUF) / B_crt ;
                t_chk ← min(t_wait, τ);
                return FALSE;
            else
                return TRUE;
            end
        else
C5          if "it is more efficient extending the current
            phase than starting the next one" then
                return FALSE;
            else
C8              if "not enough space in state buffer"
                then
C9                  if "state buffer is almost full" then
                        t_chk ← max(t_Tk | Δ_k ∈
                        state_buf) − t_B;
                    end
                end
                return TRUE;
            end
        end
    end
```

Algorithm 2: Replication algorithm in case of *degraded functionality*

delay introduced by our system meets client's requirements in terms of maximum accepted delay.

$$D_{max} \leqslant D_{max}^{req} \tag{C1}$$

In case we run normal functioning, we check if the accepted delay will not be exceed if the current phase is continued.

$$D_{max} + \frac{\Delta_i^{add}}{B_{crt}} > D_{max}^{req} \tag{C2}$$

If the accepted delay will not be exceeded, we must check in Condition C3 if the buffer will become full in case we continue the phase. In such a case a new phase will be started.

$$size(state\_buf) + \Delta_i + \Delta^{add} - B_{crt}\tau > MAX\_BUF \tag{C3}$$

If the buffer is not to be filled, the next checking is whether the required CPU usage efficiency η_{min} is met.

$$t_{R_i} \geqslant \frac{\eta_{min} t_B}{1 - \eta_{min}} \tag{C4}$$

In case the η_{min} is reached, we need to decide in Condition C5 if the phase will continue or will start a new phase.

$$P_{out}(\frac{\Delta^{add}}{\tau} + B_{crt}) \leqslant P_{out}^{add}\frac{\Delta^{com}}{\tau} \tag{C5}$$

For Condition (C5), we consider the following exceptional cases, that must be considered in the given order:

1) If $P_{out} = 0$, i.e. there is no output packet in the current phase waiting to be released (so, there is no one waiting for a response and perceiving a delay in response), then *the current replication phase is extended*. This is because the additional output packets (P_{out}^{add}), i.e. output packets in the new phase, will be delayed anyway until the pages in the current phase will be transfered, being them included in the previous phase (when decides to go to a new phase), or the current one (when decides to extend the current phase). In the latter case, however, the number of pages that must be replicated could even be smaller than in the former case, if Δ^{com} would be big (high locality of memory changes).
2) If $P_{out}^{add} = 0$, i.e. there will be no additional output packets in the next time period τ (the one considered to extend the current phase), then the current phase is terminated and the replication algorithm *starts a new phase*. A particular sub-case of this one is when $\Delta^{add} = 0$, i.e. there is no new distinct page in the next period τ. This latter case also leads to a new phase, because there will be however no packet delayed in the next phase because of new pages, but on the other hand, current packets will be delayed more if extending the current phase.
3) If $\Delta^{com} = 0$, i.e. there will be no new page changed in the next period τ to be the same with anyone changed until now in the current phase (only new additional pages), then the algorithm *starts a new phase*. An explanation of this decision could be the following: the additional output packets would however have to wait until pages in current phase are replicated; as long as there is no such page changed again in the next period τ (such that an additional output packet to wait two times for a page instead of one time), only additional pages will count for additional delay of the additional output packets in the next period τ. This would not be, however the case, for existent output packets if current phase would be terminated and a new one would be started (to correspond to the next period τ). So, no matter as many additional output packets would be in the next period τ, they have no reduction in terms of

delay if current phase would be extended, but instead the current output packets would be delayed more.

For the normal case, not belonging to the exception cases above, we will use the following notations:

$$\begin{aligned} \Delta^{add} &= \alpha \Delta^{com} \\ B_{crt} &= \beta \frac{\Delta^{com}}{\tau} \\ P_{out}^{add} &= \gamma P_{out} \end{aligned} \tag{3}$$

Replacing the above notations in Condition (C5) we will have it in the following form:

$$\alpha + \beta \leqslant \gamma \tag{4}$$

We analyze firstly the possible values for β. We considered a common, limited and very possible easily overloaded connection of $100Mb/s$, i.e. $12.5MB/s$. In that case, to have $\beta = 1$, the number of pages of the VM to be modified per second in the current replication phase should be $3200 pages/s$ (coming from $\frac{12.5*1024*1024}{4096}$, considering pages of $4KiB$). We measured however the number of distinct pages that could be modified on a real common case system (CPU: Intel Core2 Duo T5750, $2.0GHz$, RAM: $2GiB$, DIMM DDR2 Synchronous $800MHz$, $1.2ns$) and we noted the following value: about $40000 pages/s$. This corresponds to the maximum number of distinct pages that could be modified in a second. So, in this case, the minimum value for β would be $\beta_{min} = \frac{3200}{40000} = 0.08$. So the Equation (4) will be:

$$\alpha + 0.08 \leqslant \gamma \tag{5}$$

This gives us an idea of when the current phase is extended. For instance if $\alpha = 0$, i.e. there will be no additional new pages, but only the ones already changed in the current phase, then the number of additional output packets in the next period τ must be at least 0.08 more times then those already existent at the evaluation moment in the current phase, such that the current phase to be extended. When, for instance, $\alpha = 0.5$, then γ should be more than 0.58, such that the current phase to be extended.

In practice, when the link between primary and backup becomes even more overloaded, in extreme case zero, i.e. $\beta = 0$ (no connection between nodes), then the Equation (4) becomes:

$$\alpha \leqslant \gamma \tag{6}$$

In case of a degraded connectivity between the replication nodes, two more conditions are relevant for our analysis. Condition (C6) checks if there is available bandwidth, by verifying if the transfer of the current state can start immediately or not.

$$t_B \geqslant size(state\_buf)/B_{crt} \tag{C6}$$

In case there is no available bandwidth and the state buffer is full, in Condition (C7) we check if it is more efficient to continue the phase or not.

$$P_{out} t_T^{add} \leqslant P_{out}^{add}(t_T^{com} - t_{wait}) \tag{C7}$$

For Condition (C7), we will analyze the following exceptional cases, that must be checked in the mentioned order:

1) If $P_{out}^{add} = 0$, i.e. there will be no additional output packets in the next time period τ (the one considered to extend the current phase), then, like in the second exceptional case of Condition (C5), the current phase is terminated and the replication algorithm *starts a new phase*.
2) If $t_{wait} \geqslant t_T^{com}$, i.e. the necessary buffer space to be released in order to continue the phase is bigger than the space necessary to store dirtied pages during first τ period of a new phase then, the protocol will *start a new phase*.
3) If $P_{out} = 0$, i.e. there is no output packet in the current phase waiting to be released (so, there is no one waiting for a response and perceiving a delay in response), then, like in the first exceptional case of Condition (C5), *the current replication phase is extended*.
4) If $t_T^{add} = 0$, i.e. there will be no new dirtied page in the next period τ of the phase. This case leads to *start a new phase*, because there will be however no packet delayed in the next phase because of new pages, but on the other hand, current packets will be delayed more if extending the current phase.

For the normal case, not belonging to the exception cases above, we will use the notations from Condition (C5) and the next one:

$$t_{wait} = \frac{\delta \Delta^{com}}{\gamma B_{crt}} \tag{7}$$

Replacing the notations from Condition (C5) and the above notation in Condition (C7) we will have it in the following form:

$$\alpha \leqslant \gamma - \delta \tag{8}$$

In case it is not necessary to free space in buffer in order to continue the phase, i.e. $\delta = 0$, and the Equation (8) becomes:

$$\alpha \leqslant \gamma \tag{9}$$

On the other extreme, the maximum value for δ appears when the size of the necessary space to be released in order to start a new phase is maximum. Therefore, the following relation happens:

$$\delta \leqslant \frac{P_{out}^{add}}{P_{out}} \tag{10}$$

The upper bound of the γ corresponds in practice to the case in which the amount of space necessary to be released

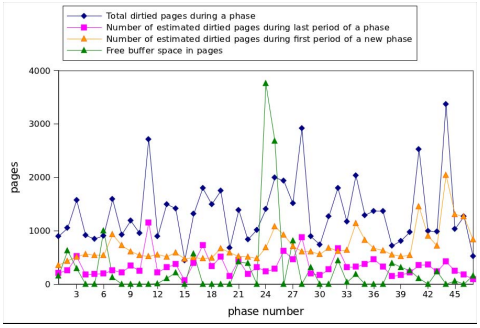

Figure 4. "Linux Kernel Compilation" behavior in terms of memory pages

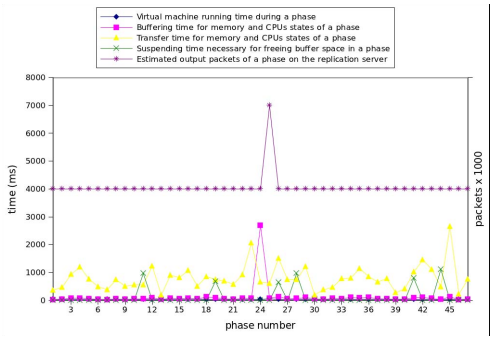

Figure 5. "Linux Kernel Compilation" behavior in terms of time and estimated output packets on replication server

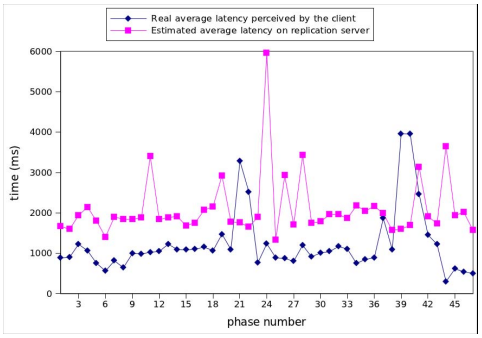

Figure 6. "Linux Kernel Compilation". Client perceived latencies and replication server estimated latencies

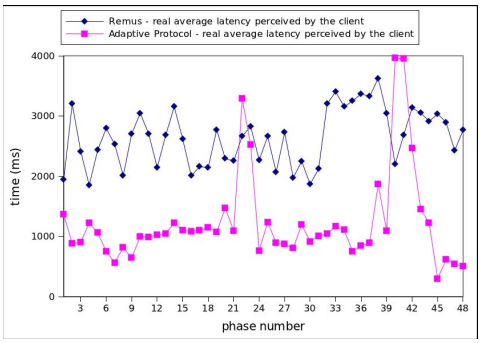

Figure 7. "Linux Kernel Compilation". Remus vs. Adapted Replication in terms of client perceived latencies

in order to continue the phase is equal with the amount of space that will be occupied during first τ period of a new phase. This happens because of the low locality of memory changes during the running of the service and therefore the network link between replicas will become overloaded.

V. Experimental Results

Trying to prove the benefits of the adaptive strategy in case of overloaded replicas connectivity we tested our adaptive replication protocol on two kinds of services. The same tests were also run comparatively on Remus. For both tests, the Remus phase running time was fixed to $30ms$ in order to reduce latencies.

These tests run on a pair of computers with an Intel Core 2 Duo 2.0 GHZ processors, $2GB$ of RAM, $160GB$ of hard disk space and a $100Mbit$ Ethernet interface. The protected VM was allocated $128MB$ of RAM and $4GB$ for hard disk. Xen 4.1.2 was installed on each machine, with 32-bit Linux kernel 2.6.32.40 running Ubuntu 11.04 as primary operating system for Dom0. In DomU was installed Linux Debian Squeeze.

The memory buffer size was $16MiB$ (e.g. 4096 memory pages for page size of $4KiB$).

A. Test 1: Linux Kernel Compilation as Virtualized Service

For the first test we used a service considered to provide Linux kernel compilation at clients requests, because it dirties memory pages and executes writes on disks faster than the speed of the network connectivity between replicas, which becomes this way overloaded in time. Also, the CPUs are used intensively for such a service. While the service was running, a client request generates 100 output packets per second on the primary VM.

In Figure 4 we see that the buffer tends to become full because memory pages are changed faster than the network bandwidth (in Figure 5 $t_T \geqslant t_R + t_S$). This means that the adaptive replication protocol runs as short as possible phases (i.e. 30 ms), in order to reduce the latencies experienced by the client. If is started a new phase, the number of estimated dirtied memory pages during a period would be greater than the number of estimated dirtied memory pages during a period in case phase is continued, because in the former case it would include also the dirtied pages from the previous phase.

Because the buffering time in Phase 24 in Figure 5 takes more, the states in the buffer can be transferred in time and the buffer becomes almost empty (see Phase 24 in Figure 4). This fact is observed in the Phase 25 from Figure 5, when the number of estimated output packets are bigger because the phase is running more time.

We observe in Figure 6 the maximum estimated latencies on the replication server (see relation 2) are bigger than

the client perceived latencies. The peeks in the perceived latencies happens because of network blockage between client and server. Therefore, for our analysis that peek values are not relevant.

In Figure 7 we observe that, overall, the clients experienced latencies are smaller in case our protocol is running for replicating Linux kernel compilation service than the case when the pure Remus implementation is used to replicate the service. The observation that was mentioned before regarded to peeks is also valid here.

Also, in case of a primary failure, when the replicated service is a Linux kernel compilation, the client experienced about 1400 packets lost when using adaptive replication protocol and about 3600 packets lost when using the Remus implementation (during a second).

B. Test 2: Apache2 Web Server as Virtualized Service

Second tested service was Apache2 Web Server, which does not change too much the VM state when is running. While the service was running, a client request generates 1000 output packets per second on the primary VM.

Figure 8 shows us that the buffer is all the time almost empty because the pages of a previous phase can be all transfered during the running step of the current phase (see Figure 9 in which the transfer time is smaller than the running time).

From Figure 9 we observe that the number of estimated output packets on the replication server are about 20 times greater than in case of the Linux kernel compilation service replication. This is because we have 10 times more requests per second and also because the number of dirtied memory pages is small and we can have longer phases (buffer will not become full) that means more estimated output packets. Overall, the latencies experienced by the client shown in Figure 10 are in the range of estimated latencies on replication server because the dirtied memory pages doesn't influence them. Again, the peeks represent network blockages and are not relevant in our analysis.

Comparing with Remus in terms of experienced client latencies, as shown in Figure 11, our protocol is a little bit more efficient, but we can say that the latencies are in the same range because the number of dirtied memory pages is small and the connectivity between replicas does not become overloaded. As before, the peek values of experienced latencies are not relevant in this graph.

For both implementations, in case of primary failure, the client experienced about 600 packets lost.

Another test for this service is to run *wget* command. We get about $350MiB/s$ when the Remus replication protocol was run and about $500MiB/s$ when was run the adaptive replication protocol.

Analyzing the results of these two tests we can say that, comparing with pure Remus implementation, the adaptive

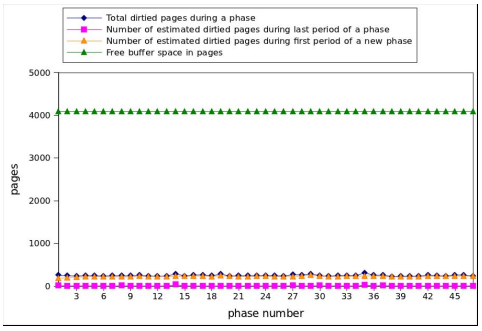

Figure 8. Apache2 Web Server behaviour in terms of memory pages

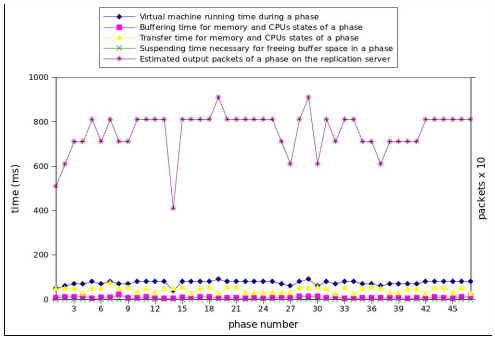

Figure 9. Apache2 Web Server behavior in terms of time and estimated output packets on replication server

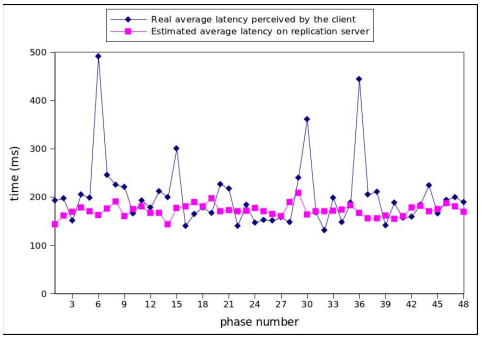

Figure 10. Apache2 Web Server - Client perceived latencies and replication server estimated latencies

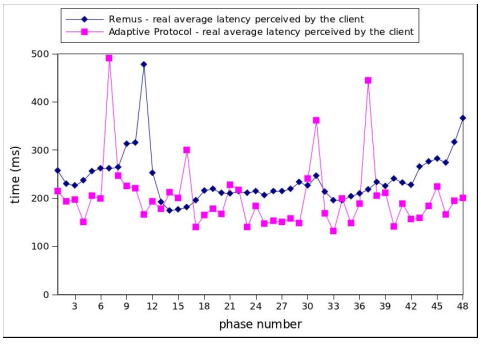

Figure 11. Apache2 Web Server - Remus vs. Adapted Replication in terms of client perceived latencies

replication protocol improves the responsiveness of the virtualized service, especially in case the connectivity between replicas is overload.

Comparing our protocol with that in [13], in case there is a high locality of memory reference, the buffering time is decreased considerably (for such a scenario, in their implementation the buffering time is greater than transfer time)

VI. Related Work

High availability based on VM replication using Xen has been explored in [10], [11], [13], Remus [12] and Kemari [14]. The main improvement our system brings over this solutions is the complete adaptive replication protocol that improve the service responsiveness in case of overloaded connectivity between replicas.

The implementation described in [12], which was used to design complex systems like the ones described in [15] and [16], provides a complete replication protocol but doesn't adapts to the service behavior. Therefore, after the replication process is started with a fixed running time for a phase it cannot adapt the running time in order to reduce the client experienced latencies in case the service response is delayed.

On the other hand, such an adaptive replication protocol was prosed in [13] but it has the drawbacks of not controlling the disk writes operations (which means we can have inconsistency between replicated disks states and memory and CPUs replicated states), suspending the VM all the time a phase is analyzed (it can take between $20ms$ and $500ms$) and too large buffering time.

Our proposed solution fills the gaps of systems described in [12] and [13] by changing the replication protocol from [12] to be adaptive and improving the adaptive protocol from [13] in terms of buffering time, suspension avoidance and disk writes replication control.

VII. Conclusion

This paper proves the fact that the adaptive replication protocol improves the responsiveness of virtualized and replicated services in case the connectivity between replicas becomes overloaded.

An improvement to existing implementation can be done by taking into account the time required for disk writes to be applied, on backup machine, when we decide to continue the phase or not.

References

[1] 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.

[2] A. Bartoli and O. Babaoglu, "Constructing highly-available internet services based on partitionable group communication," 2001. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.218

[3] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Trans. Comput. Syst.*, vol. 5, no. 1, pp. 47–76, February 1987.

[4] F. Cristian, B. Dancey, and J. Dehn, "Fault-tolerance in the advanced automation system," in *EW 4: Proceedings of the 4th workshop on ACM SIGOPS European workshop*, New York, NY, USA, 1990, pp. 6–17.

[5] T. Anker, D. Dolev, and I. Keidar, "Fault tolerant video on demand services," in *In Proceedings of the 19th International Conference on Distributed Computing Systems*, 1999, pp. 244–252.

[6] M. Marwah, S. Mishra, and C. Fetzer, "Fault-tolerant and scalable tcp splice and web server architecture," in *SRDS '06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, 2006, pp. 301–310.

[7] ——, "Enhanced server fault-tolerance for improved user experience," in *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008, pp. 167–176.

[8] Y. Saito, "Jockey: A user-space library for record-replay debugging," 2005.

[9] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, F. M. Kaashoek, and Z. Zhang, "R2: An application-level kernel for record and replay," 2008.

[10] A. Coleșa and B. Marincaș, "Strategies to transparently make a centralized service highly-available," in *IEEE International Conference on Intelligent Computer Communication and Processing (ICCP'09)*, 2009, pp. 339–342.

[11] A. Coleșa, I. Stan, and I. Ignat, "Transparent fault-tolerance based on asynchronous virtual machine replication," in *Proceedings of The 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC '10)*, 2010, pp. 442–448.

[12] 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.

[13] A. Colesa and M. Bica, "An adaptive virtual machine replication algorithm for highly-available services," in *Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS'11)*, 2011, pp. 941–948.

[14] Y. Tamura, "Kemari: Virtual machine synchronization for fault tolerance using DomT," *Xen Summit*, June 2008.

[15] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield, "RemusDB: Transparent high availability for database systems," *PVLDB*, vol. 4, no. 11, pp. 738–748, 2011.

[16] S. Rajagopalan, B. Cully, R. OConnor, and A. Warfield, "Secondsite: Disaster tolerance as a service," 2012.