

# An Adaptive Virtual Machine Replication Algorithm for Highly-Available Services

Adrian Coleșa

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

Bica Mihai

Computer Science Department  
Technical University of Cluj-Napoca, Romania  
Email: bicamihai.m@gmail.com

**Abstract**—This paper presents an adaptive algorithm for the replication process of a primary virtual machine (VM) hosting a service that must be provided high-availability. Running the service in a VM and replicating the entire VM is a general strategy, totally transparent for the service itself and its clients. The replication takes place in phases, which are run asynchronous for efficiency reasons. The replication algorithm adapts to the running context, consisting of the behavior of the service and the available bandwidth between primary and backup nodes. The length of each replication phase is determined dynamically, in order to reduce as much as possible the latencies experienced by the clients of the service, especially in the case of a degraded connectivity between primary and backup nodes.

We implemented our replication algorithm as an extension of the Xen hypervisor's VM migration operation. It proved better than its non-adaptive variants.

**Index Terms**—high-availability, virtualization, replication, asynchronous, adaptive

## I. INTRODUCTION

THE AVAILABILITY of a service is given by the proportion of time that service is perceived by its clients as functioning according to its specifications, in both normal and abnormal conditions, the latter ones being determined for example by electric power outages, hardware dis-functionalities or software attacks [1], [2].

The high-availability requirement of a service emerges from the fact that its clients need a permanent access to the service. The unavailability of some services would have a negative impact for their clients, like in case of banking institutions, telecommunication companies, military applications or even hospitals. This is the main reason the research in this field has received a great attention in the latest two decades [1], [3], [4].

We will call in this paper the way a service is made highly-available *high-availability strategy*. The software infrastructure needed to provide high-availability for that service will be named *high-availability* or *protection system* and the service itself will be referred to as the *protected service*.

Most existing solutions require design changes to hardware or software components of 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 requires 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 in general does not support nondeterministic behavior of the service. Another strategy is used by [10], [11], [12], [13], [14], [15], [16]. They run the service in a virtual machine and replicate the entire virtual machine. Such a strategy can be applied to any service and provides transparency for both service's clients and the service itself. Yet, 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 this paper we describe an improved variant of the virtual machine replication algorithm proposed in [16]. The resulting algorithm is adaptive to different environment changes during the runtime of the protected service. It dynamically calculates each replication phase's duration based on the characteristic of locality of memory changes of the protected service and the number of output network packets generated during runtime. The algorithm also takes into account the available bandwidth between the primary and backup nodes. The resulted protection system aims to reduce the network traffic in case of degraded connectivity and still maintain the latencies experienced by the service's clients as close as possible to the required values. In cases of normal condition the algorithm tries to increase the efficiency of the CPU usage on the VM running the service.

We implemented the proposed algorithm over the migration mechanism of the Xen hypervisor [12]. We run the services we want to make highly-available on a Linux distribution. The tests we performed proved our adaptive algorithm's efficiency relative to its non-adaptive variants.

## II. HIGH-AVAILABILITY STRATEGY AND SYSTEM ARCHITECTURE

The high-availability system we use is the one proposed in [15], [16]. This paper contributes to the replication algorithm used by the system. This section briefly describes the general

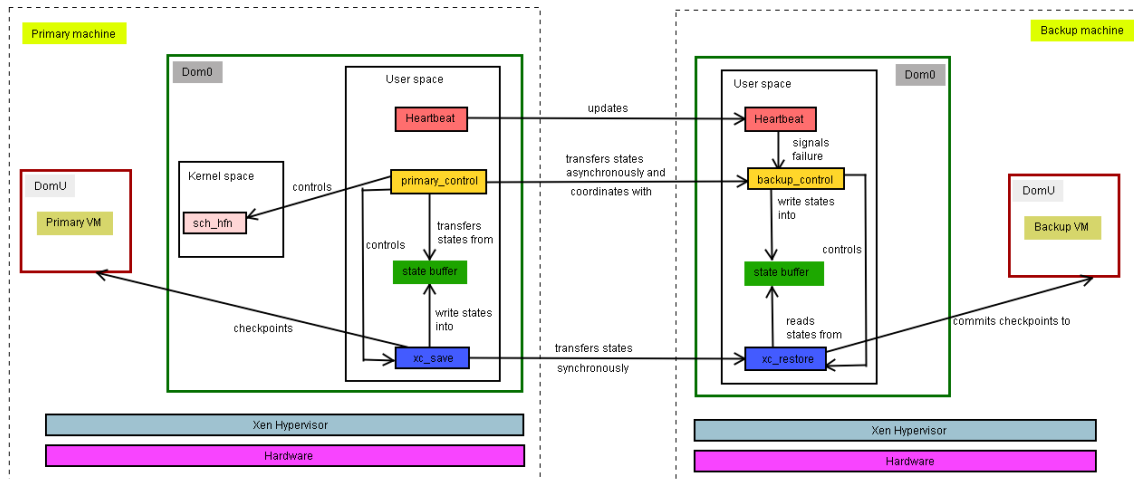


Fig. 1. Highly-Available System Architecture Built on the Xen Hypervisor

architecture and main functional characteristic of the protection system, as prerequisites for understanding the adaptive replication algorithm.

We considered centralized services that are already in use and cannot be modified in order to increase their availability level. High-availability implies tolerance to the service's components failure, which can be provided only based on some degree of redundancy. Though, the fact that the service is centralized means that it is implemented and provided by a single running server, which constitutes a single point-of-failure. In order to improve its high-availability, we must create some replicas of the service's server and run them altogether in a distributed manner. The replication strategy we consider is the passive one, with a primary and more backup replicas. It provides total transparency for the service's clients [1].

The only way of replicating a centralized service without modifying it is to place the service's server in a virtual machine (VM) and replicate the entire virtual machine [10], [14], [15]. This method is general because it can be applied to any service and operating system the service runs on.

Our solution is based on VM replication over the Xen hypervisor and extends the live VM migration operation supported by Xen. Xen allows several guest operating systems to execute on the same computer hardware in different VMs. The first VM which is booted, called *Dom0* (Domain 0), is a privileged one. It runs a Linux kernel and is used by the Xen hypervisor to interact with the hardware. This way Xen is independent of the hardware, letting this responsibility to the Linux kernel in *Dom0*. The other VMs are called *DomU* (Domain Unprivileged). User space Xen tools in *Dom0* allows the user to gain control rights over the other guest operating *DomU* VMs. To implement our high-availability system we modified some of the processes which are controlling *DomU* domains. We run the service we want to make highly-available in a *DomU* VM on a primary node and replicate that VM on a backup node in a corresponding backup *DomU* VM. The replication is controlled by processes placed in *Dom0* VM

on both nodes. The architecture of the resulting system is illustrated in Figure 1.

The main components of our system are the two user-space processes *primary\_controller* and *backup\_controller*, which controls the replication mechanism. They interact with the Xen's *xc\_save* and *xc\_restore* components, which normally implemented the Xen live migration operation, but which we modified to transform the one-time migration in a continuous replication of the VM machine running on the primary node to the backup one.

The replication takes place in phases, each one consisting in the following ordered stages: (1) *running*, lasting  $t_R$ , during which the primary VM is run, accepting inputs, but having its network outputs blocked, (2) *saving*, lasting  $t_S$ , when the VM is suspended and its state saved in the *state\_buffer*, (3) *transfer*, lasting  $t_T$ , when the previously saved VM state are transferred from the primary node on the backup one, (4) *output release*, lasting  $t_O$  and corresponding to the act of releasing the outputs of the VM corresponding to its already replicated state.

The running stage is controlled by our system and its length  $t_R$  is calculated dynamically for each stage in a way described in details in the next section. The VM state saved during a saving stage consists in the memory pages modified from the last saving point and current values of the VM's CPU registers. The corresponding  $t_S$  depends on the number of modified pages. Being small relative to the other times, we considered it constant. The transfer time  $t_T$  is dependent on the size of the VM's saved state and the available bandwidth between the primary and backup nodes. These two terms cannot be directly controlled by our system. Though, we will see below how we can indirectly reduce the transfer time of the overall replication process, trying to replicate as few pages as possible, when network bottlenecks occur.

Blocking the primary VM's outputs until its corresponding state is replicated, provides for its transparent replacement by

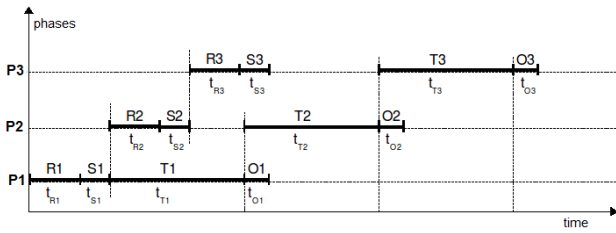


Fig. 2. Asynchronous Replication Strategy

a backup VM, in case it crashes. The kernel-space module `sch_hfn` controls the blocking and releasing of the primary VM's network outputs. The crashes of the primary VM are detected using a simple *heartbeat* mechanism.

The replication process is asynchronous, parallelizing as many replication stages as possible, in order to reduce the latencies experienced by the service's clients. Actually, only some of the stages can be performed in parallel. The way different stages of successive replication phases overlap is illustrated in Figure 2. We can note that during the transfer and output release stages of previous phases, running and saving stages of the following phases can take place.

### III. THE ADAPTIVE REPLICATION ALGORITHM

Experimental results in [16] have led us to the conclusion that an adaptive replication algorithm, which would calculate dynamically the running time  $t_R$  of each replication phase, based on the current runtime environmental conditions, could provide improvements in terms of decreasing the network load, increasing the CPU usage efficiency, and reducing the latencies perceived by the clients (response time).

We must note however that not all the mentioned parameters could be optimized simultaneously. For instance, the strategy of getting a better CPU usage could be in contradiction with the one trying to decrease the response time. We will see below the way our adaptive algorithm trades between these factors.

We denote the *CPU usage efficiency* by  $\eta$  and defined it by the relation:

$$\eta = \frac{t_R}{t_R + t_S} \quad (1)$$

The smaller the running time and, implicitly, more frequent the saving stages, the lower the efficiency of the CPU usage, because most of the time the CPU would be used by the saving process, which is time consumed additionally by the protection system and not by the service. Equation (1) can be thought as the fraction between the running times in cases the protection system is deactivated and respectively, activated.

One condition required to our replication algorithm is to try to keep  $\eta$  above a certain value  $\eta_{min}$ . For instance having a  $t_S = 50ms$ , in order to provide a  $\eta_{min} = 90\%$ , the algorithm must use a running time  $t_R$  of at least  $450ms$ . This could be too much for a response time of a client request. Just to have an idea of this problem, we will calculate the maximum delay our protection system could introduce in a client request's response

time in the particular favorable case, when we consider having enough network bandwidth available in each replication phase, such that the transfer stage of a phase could start immediately after the saving stage of that phase. This means that for any phase  $i \geq 1$ ,  $t_{T_i} \leq t_{R_{i+1}} + t_{S_{i+1}}$ . For simplicity we also consider the time of all similar stages to be identical in all phases and such we will have the following formula for the delay introduced by our system in the response time:

$$D_{max} = (n + 2)t_S + t_T + t_R \quad (2)$$

where  $n$  is the number of phases needed for the service to handle that client request and generate a response in terms of one or more output network packets.

Coming back to the above example, for  $t_R = 450ms$  and  $t_S = 50ms$ , considering a  $1Gbs$  network bandwidth, a VM state to be replicated of about  $10000pages$  of  $4KB$  each and  $n = 1$ , the maximum delay will be  $D_{max} \approx 855ms$ , which could be really too long for many of the client applications. So the running time should be decreased in such cases. In the case of not having enough bandwidth available, the transfer time becomes the dominant factor in the delay formula, and the result could be even worse.

From the example above we can understand intuitively the way our replication algorithm will act. Firstly, it will try to keep the response time as small as possible, because this is directly perceived by the service's clients. This depends on the size of the VM state that must be replicated and also on the available bandwidth. As long as the replication algorithm can provide

$$D_{max} \leq D_{max}^{req} \quad (C1)$$

where  $D_{max}^{req}$  is the maximum acceptable delay required by service's clients, it increases the running time of the current replication phase, in order to get a CPU usage efficiency as close as possible to the required  $\eta_{min}$ . This case is what we will consider to correspond to the *normal functionality* of the system. In case, the network bandwidth is not enough (e.g. network overloaded or a large VM state) to transfer in time the VM state, the replication algorithm will calculate the running time of the each replication phase such that to minimize the delay  $D_{max}$ . Such a situation we will call *abnormal* or *degraded functionality*.

The parameters taken into account to establish each  $t_{R_i}$  are: (1) the protected service's behavior regarding the locality of memory changes, measured as the number of new distinct pages that will be modified in the next future in current phase  $i$ , noted  $\Delta^{add}$ , (2) the available current network bandwidth between the primary and the backup nodes, noted  $B_{crt}$ , (3) the output network packets generated on the primary VM.

The first two parameters are predicted each  $\tau ms$  (a period established by the system administrator) using the exponential average. The third parameter consists in both the number of the output packets generated until the current moment, noted  $P_{out}$ , and that of the packets that will be generated in the next  $\tau ms$ , noted  $P_{out}^{add}$ .

The ideas the algorithm is based on are the followings: when there are output packets waiting to be released, it tries to reduce the running time, in order to reduce the response time of client requests. Reducing the  $t_R$  will result in a smaller VM state that must be replicated, so a smaller transfer time, so again a smaller delay in the response time. Nevertheless, the number of replicated states will be bigger, comparing with the case of longer replication phases. This means, firstly, a worse CPU usage and, secondly, a possible greater total network load. The latter especially occur when the replicated VM and implicitly the protected service manifests a greater locality of memory changes, i.e. modifies approximately the same set of pages in consecutive time intervals. This could result in the same set of pages being replicated more times consecutively, corresponding to more consecutive saved VM states. Such a situation must be avoided when the network is overloaded and the transfer of a saved state cannot be started immediately after the saving stage finishes.

Based on the above considerations, the first thing the algorithm does is to check whether there are output packets waiting to be released. If there is no one, then it makes no sense terminating the current phase, especially if there is no bandwidth available, because there will be no delay perceived by the service's clients. Furthermore, the shorter the running stages, the greater the number  $n$  of replication phases a client request handling lasts and, consequently, a larger response time.

The second thing the algorithm considers is the available network bandwidth. If there is enough available, then the current phase could be terminated and a new one started, because the replication of the current one could be started immediately. If not, maybe it could be more appropriate to enlarge the current phase (actually its running stage), especially if the service behaves locality of memory changes, just to avoid a greater amount of memory replicated and a greater overall delay.

The exact decision the algorithm takes at one moment regarding the continuation or finish of the current running stage (and implicitly current phase) depends on the number of output packets and the number of VM's memory pages already modified, i.e. the size of the VM state that must be replicated. It evaluates, based on the currently measured and predicted values, if the average delay of the output packets would be greater if the current phase is continued or terminated. The best case is always chosen. If the decision is to continue the current phase, then a new evaluation is made after a  $\tau$  period.

The detailed description of the algorithm will illustrate the above ideas. Algorithm 1 describes the strategy followed in case of normal functionality, actually when enough bandwidth is available. It returns *TRUE* if the current phase must be terminate and *FALSE* otherwise. Also, it establish the time after which a new estimation will be performed. Firstly, the algorithms checks if continuing the current phase could result in exceeding the  $D_{max}^{req}$ . Condition C2 take into account the currently introduced maximum delay  $D_{max}$  and also the additional time to transfer new distinct pages that will be

modified if continuing the current phase. Thus, we have the following relation for C2:

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

In case the required delay is not exceeded, the algorithm checks if the state buffer will be filled or not taking into account the current size of the buffer, the page already modified, i.e. the current size of the VM state  $\Delta^i$  that must be replicated and the number of new distinct pages that will probably be modified in the next period  $\tau$ . Thus, the condition C3 can be express like:

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

The term  $\Delta^{add} - B_{crt}\tau$  represents the number of pages that will be actually accumulated to the current state in the next period. If the buffer is not to be filled, the next checking is whether the required CPU usage efficiency  $\eta_{min}$  is met or not. Condition C4 is expressed based on Equation (1):

$$t_{R_i} \geq \frac{\eta_{min} t_S}{1 - \eta_{min}} \quad (C4)$$

In case the  $\eta_{min}$  is not reached, the current phase is continued. Otherwise, the algorithm does not decide to terminate the current state. If the average delay of currently waiting output could be greater than the estimated average delay of the overall output packets, then the current state will be extended. Intuitively, this could happen if in the immediate future (at least next  $\tau$  ms) the number of new generated output packets will be great relative to the number of new distinct modified pages that will be added to the current VM state, i.e. the VM manifest a good locality of memory changes. The condition C5 could be express using the formula:

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

where  $\Delta^{com}$  is the number of pages that would belong to both current phase and a possible next one, if the current one would be terminated.

Algorithm 2 describes the steps taken by the protection system in case the required maximum delay in the client response time cannot be provided. Some decisions are similar with those in Algorithm 1, so we detail only the others. Firstly, the algorithm checks if there is available bandwidth. Actually, it checks if the transfer of the current state can start immediately or not. So, the condition C6 can be written:

$$t_S \geq size(state\_buffer)/B_{crt} \quad (C6)$$

In case the current state cannot be replicated immediately, the algorithm try to find the best solution to get a minimum overall delay. In essence, this depends on the locality of memory changes manifested by the replicated VM and the number of generated outputs. The state buffer is tested for three other distinct situations, different by that of empty buffer

```

task VM_replication;
function normal_functionality () : boolean
   $t_{chk} \leftarrow \tau$ ;
  if “it is possible to exceed  $D_{max}^{req}$ ” then
    | return TRUE;
  else
    if “the state buffer is full” then
      | return TRUE;
    else
      if “required  $\eta_{min}$  provided” then
        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
  end

```

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

tested by condition C6: (1) buffer full, tested by condition C3, (2) available space above a safe threshold, tested by condition C8, and (3) available space below a safe threshold, i.e. the trend is to rapidly fill the state buffer, tested by condition C9.

We will not insist anymore on conditions C8 and C9, because they are very similar with C3. We only note that in case the buffer is almost full, the next evaluation moment will not be as usual after  $\tau$  ms, but after the time needed to transfer the saved VM state with the maximum size.

Condition C7 is very similar with C5, the difference consisting in a new factor taken into account: the time the algorithm has to wait for enough space in state buffer to save the current VM state, in case the current phase is terminated, before starting a new replication phase. It can be written like:

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

where  $t_T^{add}$  is the transfer time for the  $\Delta^{add}$  pages,  $t_T^{com}$  is the transfer time for the  $\Delta^{com}$  pages, and  $t_{wait}$  is the time the algorithm must wait for sufficient space in state buffer to save the current VM state.

#### IV. TESTS AND RESULTS

The following tests were made on two computers having the same configuration Intel Core 2 Duo 2.7GHZ processor, 2GB of RAM, 80GB of hard disk space and a 100Mbit Ethernet interface. Xen 3.3.1 was installed on each machine, with the Linux kernel 2.6.26-1-xen-686, running Ubuntu 9.04 as primary operating system for Dom0. In DomU was installed Linux Debian Lenny.

```

task VM_replication;
function degraded_functionality () : boolean
   $t_{chk} \leftarrow \tau$ ;
  if “there is available bandwidth” then
    | return TRUE;
  else
    if “state buffer is full” then
      if “it is more efficient extending the current
      phase than waiting for saving space in state
      buffer and starting the next phase” then
         $t_{wait} \leftarrow \frac{\max(0, \text{size}(\Delta_i) + \text{size}(\text{state\_buffer}) - \text{MAX\_BUF})}{B_{crt}}$ ;
         $t_{chk} \leftarrow \min(t_{wait}, \tau)$ ;
        return FALSE;
      else
        | return TRUE;
      end
    else
      if “it is more efficient extending the current
      phase than starting the next one” then
        | return FALSE;
      else
        if “not enough space in state buffer” then
          if “state buffer is almost full” then
             $t_{chk} \leftarrow \max(t_{T_k} | \Delta_k \in \text{state\_buffer}) - t_S$ ;
          end
        end
        return TRUE;
      end
    end
  end

```

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

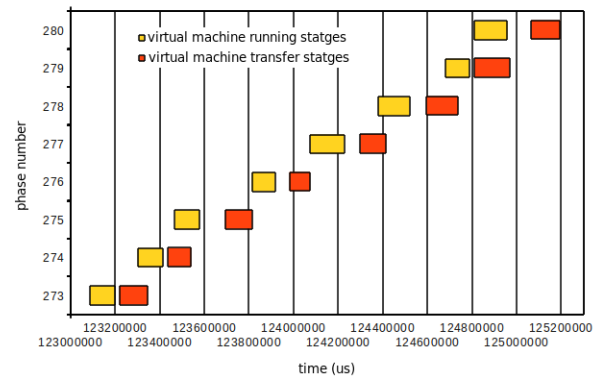


Fig. 3. Stages superimposing under usual stress



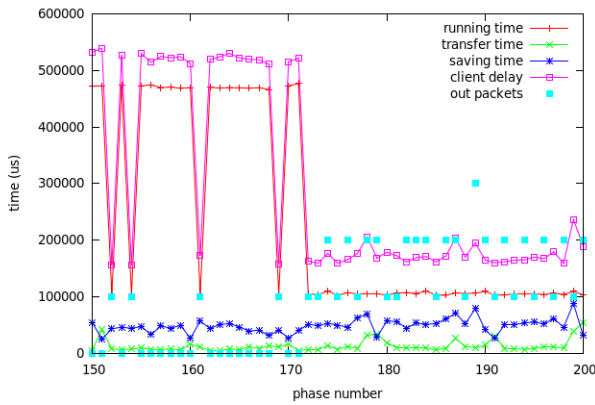


Fig. 4. Client delay reducing on increasing output packets number

In Figure 3 we illustrated a Gantt chart to show how some of the consecutive phases are superimposing in practice and reduce overall client delay. In Figure 2 we see how the phases are superimposing based on theoretical formulas. Our practical results are thus similar. If the replication of the system would be made using only the asynchronous strategy without the adaptive algorithm, the length of the running stages would be of fixed length as they were configured by the user or the system administrator. In the Figure 3 on the  $x$  axis is represented the running time in micro seconds. Time 0 corresponds to the starting of the replication process. The first time represented is the second 123 and the distance between the vertical grids is  $200ms$ . In 4 phases out of 8 there is an overlapping between the transmission stage of the previous phase and the running stage of the next phase. The process which is running in background is modifying around 350 pages per each iteration. If the replication strategy would use an synchronous algorithm the total running time would be larger because the phases would succeed one after another.

In Figure 4, starting with phase 174 a large number of inputs (client requests) was simulated using the Linux command `ping target -i 0.1`. This command makes the virtual machine to generate around 2 or 3 output packets in each phase and simulates the fact that 9 clients are making a request each second. As seen from the figure, when there are output packets, our algorithm instantly reduces the running time in order to create a very small delay to the clients, around  $200ms$ , when the replication algorithm is configured to run the virtual machine for at least  $100ms$  in order to ensure the required CPU efficiency. In this graphic we can also see that if there is a small number of dirtied pages, the saving time of the state to a local buffer in memory is greater than the transfer time of the state to the network. This is because dirtied pages are generated in a process and sent to another process using shared memory and the Linux scheduler decides when to switch between processes.

In Figures 5 and 6 are represented client delays in two different situations: (1) when there are 9 requests per second, simulated with the command `ping -i 0.1`, and (2) in the

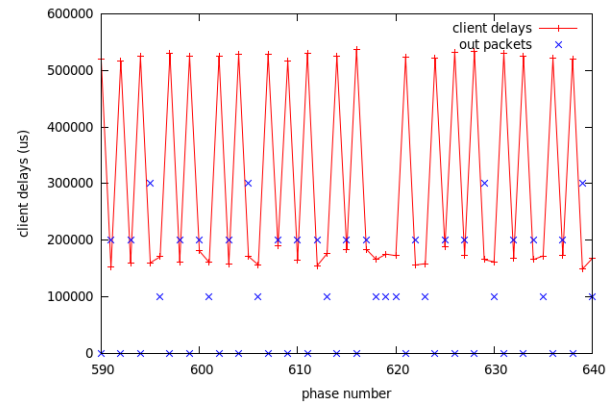


Fig. 5. Delays in context of 9 client requests per second

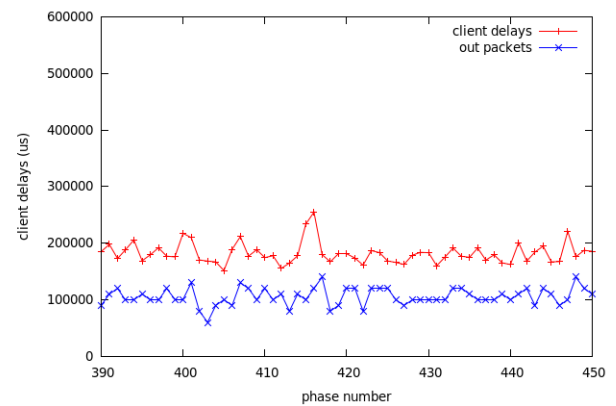


Fig. 6. Delays in context of 100 client requests per second

situation where 100 client requests are generated per second, sent by clients simulated with the command `ping -i 0.01`. Output network packets are not plotted against the  $y$  axis, which represents the time, but they are plotted against a secondary fictional  $y$  axis show there number multiplied with  $10^4$ , just to see them easily. As seen from the Figure 5, there is a quick response in how the the client delays are reduced when there are output packets. Otherwise, the client delays are not reduced. In Figure 6 the delays are less than  $200ms$  for all 100 clients, which are making one request per second.

The test in the Figure 7 shows how the system behaves when the number the output packets remains constant and the number of modified pages is increased. The number of modified pages is plotted against a fictional  $y$  axis showing their number. In the first phases, from 40 to 90, the number of modified pages is constant and around 120. From the phase 90 to 130, the number of modified pages will increase up to 4000 modified pages per phase. The page number is increased running in the VM a test program, which is modifying many pages. In this case, the system will not be able to generate a very good response time because it will run in degraded functionality, but the modified pages will have a increased locality of memory modifications, and for this

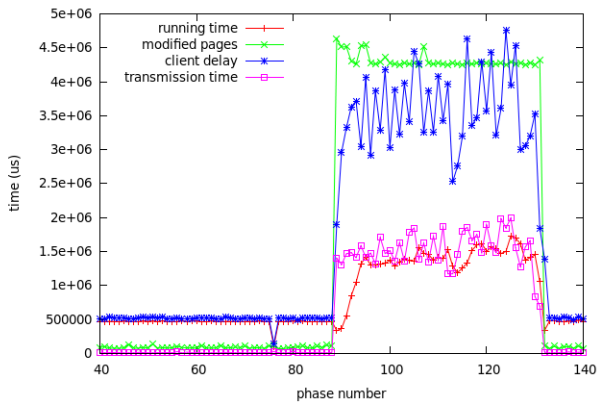


Fig. 7. Dirtied page variation, no clients connected

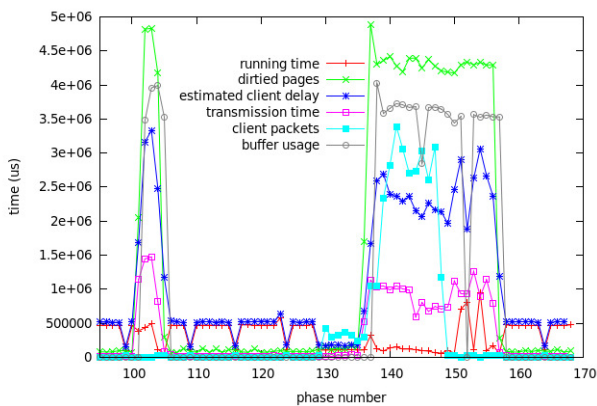


Fig. 8. Variation of running time in a complex scenario

reason we can increase the running time without generating too many additional dirtied pages. The major benefit of increased running time in this case is a better CPU usage efficiency. We can observe an increase in CPU efficiency and also while the CPU is running a phase, the previous phase is being transferred by the network interface at the same time. This is also confirmed by the fact that the transmission stage is equal to the running stage.

In Figure 8 is represented an all case scenario for our system. In this test, a program which is generating around 4800 pages per iteration is being run in two random moments. Also, while the program was running, clients connections were simulated as represented in the figure. We have to mention that the network bandwidth during the idle periods, when no program is running, is around  $1.3MiB/s$  and during the degraded functionality is around  $18MiB/s$ . When the adaptive algorithm was deactivated, the idle network traffic was around  $7MiB/s$ . This means that the adaptive algorithm reduces network traffic during idle periods by up to 5.3 times, compared to a simple asynchronous replication protocol.

Another test we made was to copy a large archive of  $125MB$  to the protected VM and then extract the data and compare the results in cases our protection system was

activated and not activated respectively. The copying speed over the network of the archive was around  $404KB/sec$ . After the file was copied the contents was extracted. The extraction period was around 50 seconds, when there were no other clients connected and around 58 seconds, when there were simulated 100 requests per second. We unzipped the same file on the system without the replication being activated and we measured 10 seconds. We conclude from this test that our replication system is 5 up to 5.8 times slower than the system with protection deactivated.

## V. RELATED WORK

Virtual machine replication based on Xen has been explored in [15], [16], Remus [14] and Kemari [17]. The main improvement our system brings over other solutions is the adaptation property of the replication algorithm. We are not aware of any other similar strategy.

The system in [16] is the most similar with our system. Actually, we developed further it. The adaptive replication algorithm is overall more efficiently than its non-adaptive variant, reducing in case of a degraded network link between primary and backup the client response time.

Remus is also very similar with our system. Actually it was introduced as a high-availability mechanism in newer versions of Xen. They reported in [14] a possible improvement consisting in dynamically modification of the rate at which the protected VM operates, in order to reduce the number of modified pages per replication phase, which would result in a reduced transfer time and, consequently, a reduced response time. Our strategy is better in cases of VM manifests high locality of memory changes, since it let the VM run at normal rate, by enlarging the replication phases. The proposed Remus optimization can be combined with ours in case the locality of memory changes and network bandwidth are very poor. What our system also makes better than Remus is that it reduces the frequency of replication phases, when there are few or no output packets, just to get a better CPU usage efficiency. Although, they have a very efficient saving method of VM states, which automatically lead to better CPU usage, but actually this is complementary to our strategy and can be integrated with it in order to get an even better efficiency.

## VI. CONCLUSION

This paper proves the fact that the asynchronous adaptive replication algorithm can improve the performance, client response time and reduce network bandwidth of high availability systems in situations where the environment changes are very often.

A drawback of our system is the buffering time. In present, the buffering time is too large and for a small number of modified pages it can be greater even than the network transfers. Some improvements can be made by implementing a better buffering technique.

## 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. Dancy, 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: ACM, 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*. IEEE Computer Society, 2006, pp. 301–310.
- [7] —, "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.
- [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] 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.
- [11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: enabling intrusion analysis through virtual-machine logging and replay," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 211–224, 2002.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] A. Coleșa and B. Marincăș, "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.
- [16] 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)*. IEEE Computer Society, 2010, pp. 442–448.
- [17] Y. Tamura, "Kemari: Virtual machine synchronization for fault tolerance using domt," June 2008.