Replicated Virtual Machines

John R. Douceur and Jon Howell

MSR TR-2005-119

TR release date: September 20, 2005

Patent filing date: December 22, 2004

1 Abstract

The Replicated State Machines (RSM) is a powerful, simple abstraction for providing fault tolerance to arbitrary computational tasks. Unfortunately, RSMs require the computational task to be specified as a deterministic state machine, a model that is theoretically convenient, but in practice, often difficult to achieve given legacy libraries and development environments.

We observe that the Virtual Machine (VM) interface is an interesting level at which to constrain the behavior of a computation to be a deterministic state machine. Because it is a narrow interface, it is possible to eliminate nondeterminism in the execution of the VM; this paper describes the techniques necessary to achieve that goal. And because the VM is a very low interface, it enables the reuse of almost the entire software stack, including the operating system, all libraries, and existing applications, even entirely unmodified.

2 Origin of this report

This technical report describes an invention upon which a patent has been filed. One of the purposes of the patent system is to encourage the public disclosure of new inventions; unfortunately, patent applications are written in “Patentese,” a degenerate descendent of English notorious for its inscrutability.

Filed patent applications in Patentese are often derived from reasonably scrutible original documents written in English, and that is the case for the present invention. The authors have no immediate further plans for this invention, and hence do not expect to write a pedagogically-improved description any time soon. Therefore, we are releasing this English document as a technical report in the hopes of better communicating the invention to the technical public.

Therefore, this report has been produced with a minimum of effort beyond that originally required to begin the Patentese disclosure. In particular, it does not evaluate an implementation, and is decidedly incomplete about academic scholarship. *Caveat lector*.

Section 3 provides background on the key technologies used in the invention. Section 4 describes the invention itself.

3 Background

This section provides high-level background about replicated state machines (RSMs) and virtual machines (VMs).

3.1 Replicated state machines

Many software applications have a client-server architecture. The client component of the application resides on a client machine, and the server component of the application typically resides on a separate server machine. Typically, multiple clients interact with a server. For example, multiple electronic-mail clients might interact with a mail server, or multiple database clients might interact with a database server.

Fig. 1 illustrates a typical client-server computer system. The dashed line indicates that the client application communicates with the server application. The actual path for this communication involves the operating system, network interface card (NIC) driver, and network interface card on each computer.

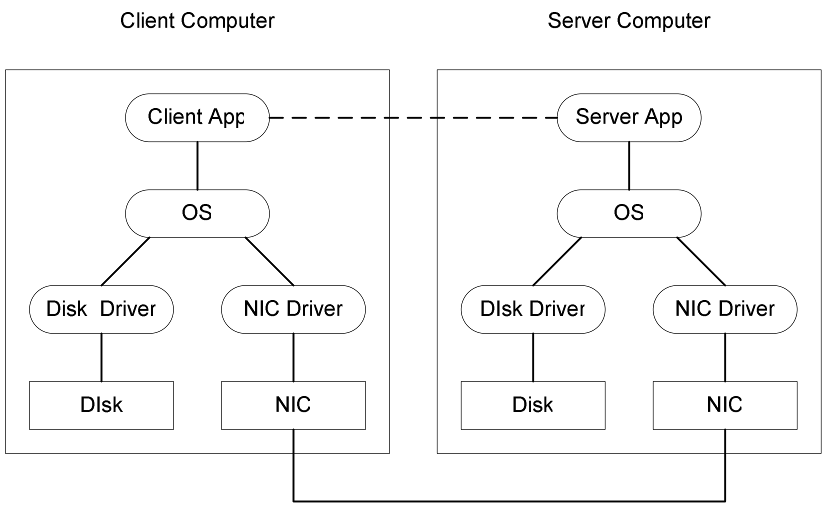

Figure 1. A typical client-server computer system.

In client-server systems, it is advantageous for the service to be reliable. In particular, this means that the service should be resilient to server-machine faults. One common mechanism for improving service reliability is replication. By running a service on several machines concurrently, the failure of some machines may be masked by other machines.

A known prior-art technique for supporting service replication is the “replicated state machine” (RSM) strategy. In the conventional RSM approach, the service is written as a deterministic state machine; this state machine is replicated on several machines, and an RSM substrate coordinates the behavior of the separate state machines so that their executions proceed consistently.

Fig. 2 illustrates a typical RSM-based client-server computer system. The client component of the application communicates with the client portion of the RSM substrate, and the server component of the application communicates with the server portion of the RSM substrate.

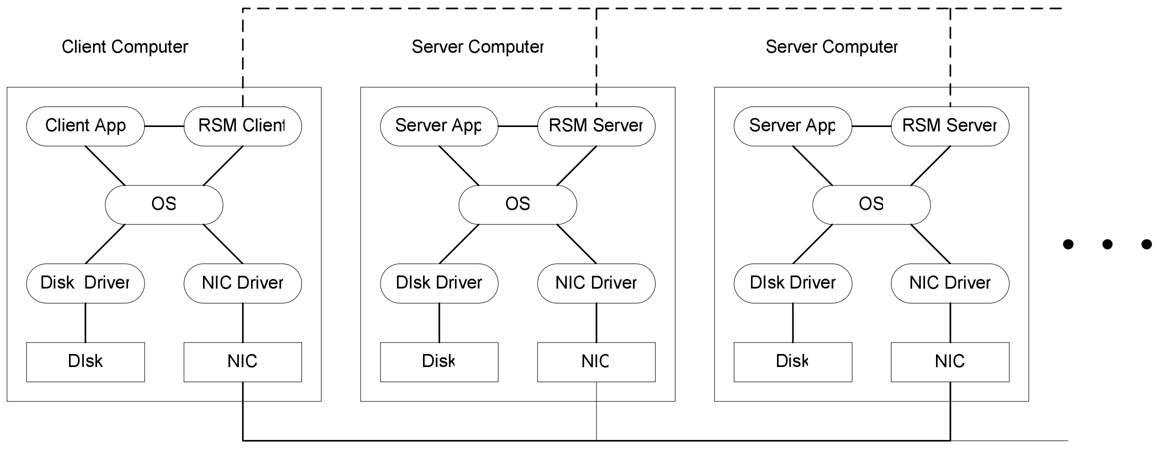

Figure 2. A typical RSM-based client-server computer system.

The client portion of the RSM substrate ensures that the client application's message is received by the replicated server. It does this by sending the message to all server replicas. However, as an optimization, it may first send the message to only one server, and if the server group does not reply correctly, it may then send the message to all servers. The client portion of the RSM substrate also collects replies from the replicated servers, and it passes a single aggregated reply to the client application.

The RSM substrate coordinates all the various instances of the server application so they appear to act as a single server, even if some of the server computers fail.

A key task of the RSM substrate is to establish a task ordering for the server's operation. Fig. 3 illustrates an example timing diagram. The downward-pointing arrows indicate requests for operations; each of these may be a request from a client, or it may be a request triggered by a server-based timer. The RSM substrate performs a protocol to determine an agreed order for the requests, and then each server replica executes the request. For example, when request Foo is received, this receipt triggers the RSM substrate to run its agreement protocol, which decides to that Foo should be the next request to execute. Following the agreement, each of the server replicas executes operation Foo. A similar sequence of events happens when request Bar is received.

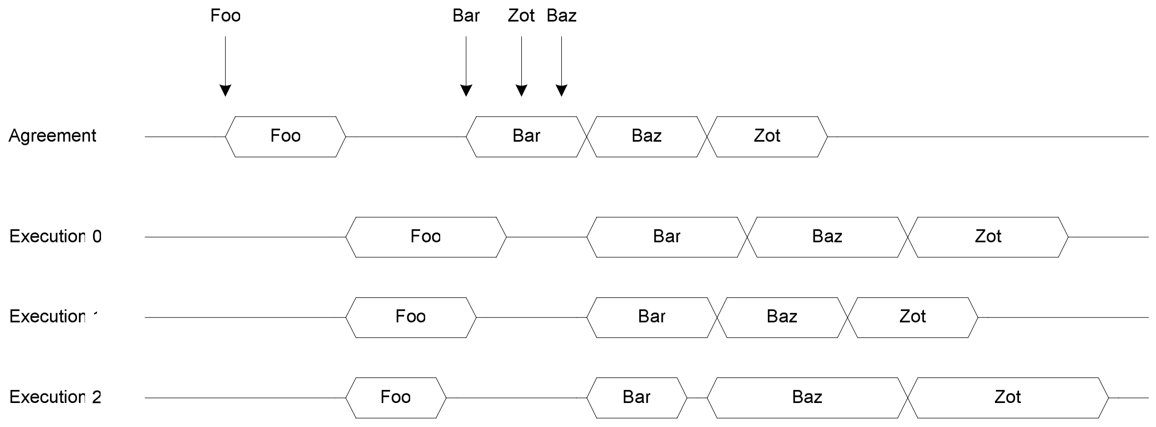

Figure 3. An example timing diagram.

Requests Zot and Baz are received while the agreement protocol is still deciding on request Bar. Once the agreement for request Bar is complete, the RSM substrate then decides whether Zot or Baz should be processed next. In the example, the substrate chooses Baz, and in the subsequent agreement step, the substrate chooses Zot.

Note that replicas 0 and 1 execute operations slower than the agreement protocol makes decisions. This seems to suggest that the agreement can get arbitrarily far ahead of the execution; however, the RSM substrate prevents agreements from concluding if they get more than a given operation count ahead of the execution.

As Fig. 3 shows, the server replicas may execute operations at differing rates. This may be because different server computers have different processor speeds, or it may be because they have varying workloads other than the workload of running the replicated service. For example, replica 2 executes operations Foo and Bar relatively quickly, but then it executes operations Baz and Zot slowly, perhaps because another process began competing for resources.

Fig. 4 illustrates a typical interface presented by an RSM substrate. The substrate uses an *execute* call to tell the application to update its state. This call includes the client message that triggered the update. The application uses the *reply* call to indicate a message to send to the client.

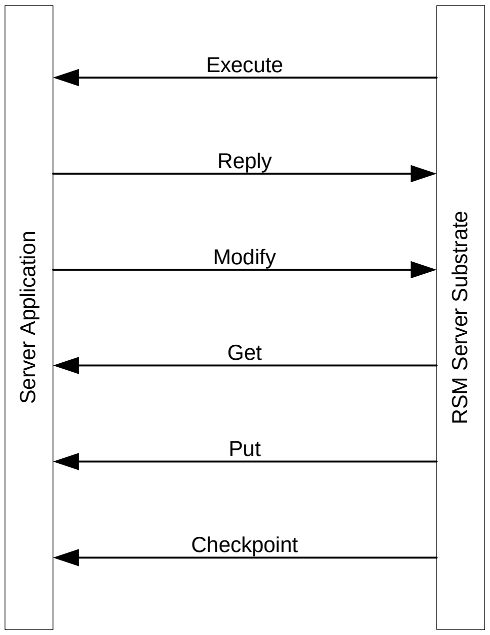

Figure 4. A typical interface presented by an RSM substrate.

The RSM substrate needs to track the state of the replicated application. Before the application modifies any part of its state, it uses the *modify* call to warn the substrate about the part of its state it is about to change. The substrate uses the *get* call to retrieve the value of any part of the application's state, and it uses the *put* call to change the value of a part of the application's state.

The RSM substrate uses the *checkpoint* call to tell the application to save a checkpoint of its state. This means that if the application were to crash and restart, then the state that it should restart with should be the state of its most recent checkpoint. Checkpoints must be saved atomically, and they must be coordinated with the RSM substrate's saving of its own internal state.

A major disadvantage of this prior-art technique is that it requires the application to interact with the RSM substrate in a rigidly defined manner, such as that described above. The server application must be architected as a state machine that updates its state only in response to messages received via the substrate from clients (or from a server-based timer). In addition, messages to clients must be sent via the RSM substrate, rather than directly. Furthermore, the server application must be able to export its state to the substrate; it must be able to import its state from the substrate; and it must ensure that all of its actions are deterministic. Further still, the server application must be able to checkpoint its state in a manner that is both atomic and coordinated with the saving of the RSM substrate's state.

These requirements may be very difficult to satisfy for an existing application that was not originally written as a state machine. They may be extremely difficult to satisfy if the application was written with multiple threads of control. Even writing a new program as a deterministic state machine is not simple, because this style of programming is unfamiliar to many programmers and because it precludes the use of nondeterministic abstractions, such as threads.

These difficulties have been a major factor in limiting the widespread adoption of RSM technology, despite this technology's promise of a general mechanism for replication-based fault tolerance.

3.2 Virtual machine monitors

Virtual machine monitors (VMMs) are well known in the prior art. Fig. 5 illustrates a computer running a typical VMM. The VMM runs as an application in the computer's host operating system. The VMM uses software to emulate computer hardware, thereby presenting a virtual machine (VM) environment. A guest operating system runs inside this VM, and the guest OS is unaware that it is not running directly on actual hardware.

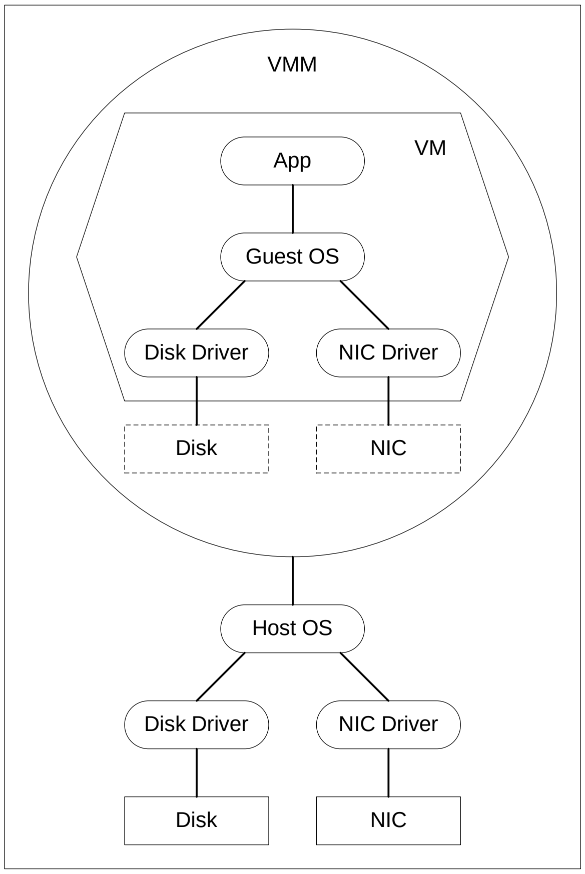

Figure 5. A computer running a typical VMM.

The VMM presents virtualized resources to the VM. In particular, it presents virtualized disk, virtualized physical memory, virtualized network interface, and so forth. Note: Virtualized physical memory should not be confused with virtual memory. Virtualized physical memory appears to the guest OS as physical memory, and the guest OS implements virtual memory on top of this virtualized physical memory. The VMM uses the host OS's virtual memory to implement its virtualized physical memory.

The VMM implements virtualized storage resources using the real storage resources it accesses through the host operating system, and it implements virtualized communication resources using the real communication resources it accesses through the host operating system. For example, the VMM presents a virtual disk to the VM, and it uses the physical disk as a backing store for this virtual disk. Similarly, the VMM presents a virtual network card to the VM, and it uses the physical network card to send and receive packets on behalf of the virtual network card.

The trickiest resource for a VMM to virtualize is the processor. For performance reasons, it is best to allow the guest operating system and guest applications to execute their instructions directly on the real processor. However, the VMM must trap certain instructions in order to simulate the behavior of privileged instructions and to redirect I/O operations to the virtualized resources. If a particular processor architecture has instructions that cannot be trapped but whose behavior must be augmented for virtualization, dynamic binary rewriting is used to replace instances of these instructions with explicit trap instructions.

Virtual machines have several customary uses. By running a newer version of a guest operating system on top of an older version of a host operating system, virtual machines can allow testing of new software without endangering the host system. By running an older version of a guest operating system on top of a newer version of a host operating system, virtual machines can support legacy applications that cannot run on a newer operating system. By running different operating systems as host and guest (for example, running a guest Unix operating system on a host Windows operating system), virtual machines can support operating-system heterogeneity.

4 The invention

The invention is a mechanism that enables any client-server application to be run as a replicated state machine, without requiring the application to be modified in any way. The invention uses an RSM substrate to coordinate the execution of multiple VMMs, each of which runs an identical copy of an operating system and server application.

Fig. 6 illustrates the preferred embodiment of the invention. Network messages to and from the unmodified client application are intercepted by an RSM client driver, which performs sent-message replication and received-message aggregation as described above. The RSM server substrate expects the VMM to act as a deterministic state machine following an interface such as that described above. The present invention includes mechanisms that enable the VMM to act as a state machine and to support such an interface.

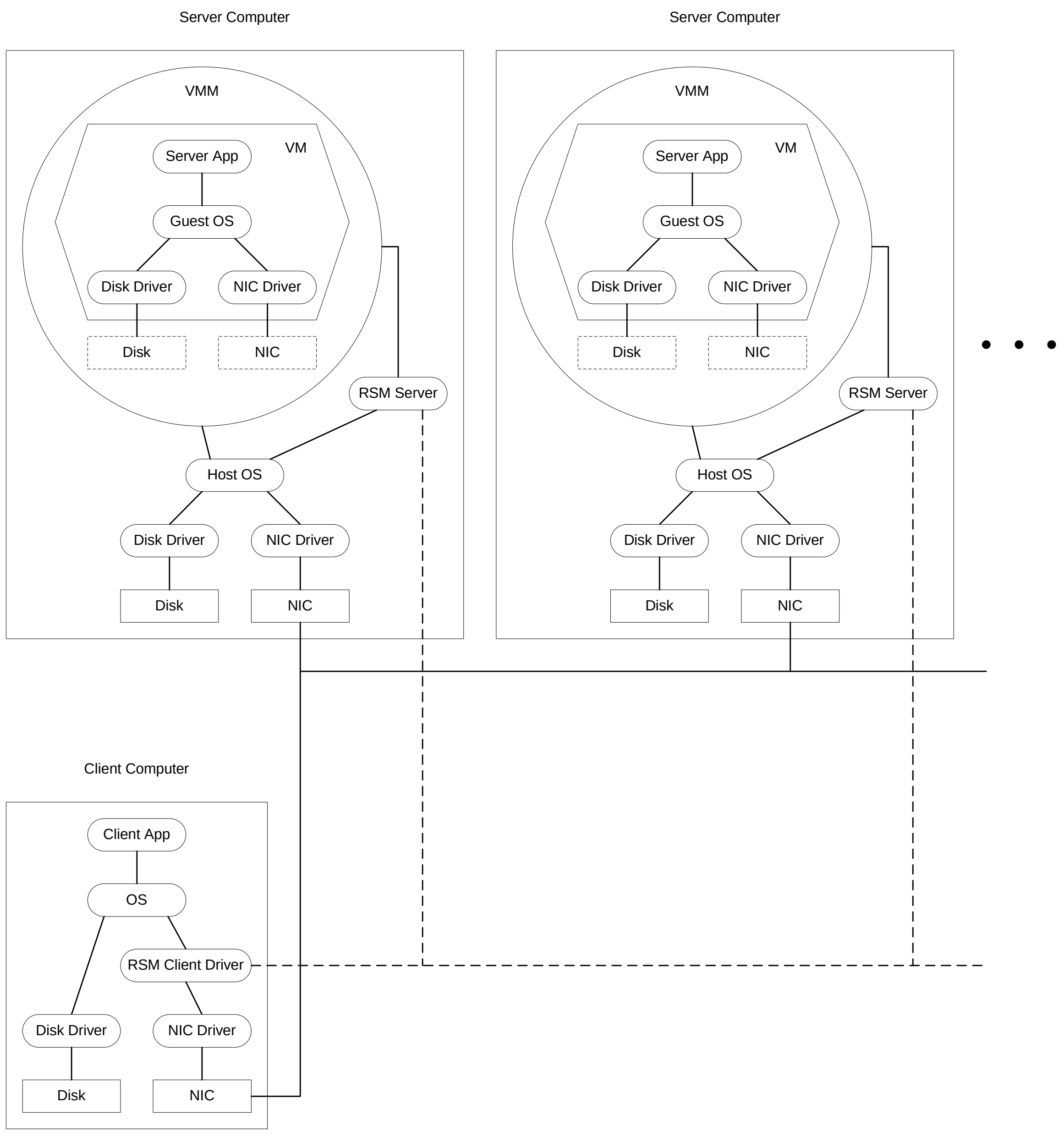

Figure 6. The preferred embodiment of the invention.

Fig. 7 illustrates an alternative embodiment of the invention. This embodiment introduces a redirector computer that acts as a liaison between the client and servers. The client sends network messages to the redirector, which replicates the messages and sends them to the server computers. The redirector also collects multiple messages from the servers, which it aggregates into single messages and sends to the client. This redirector may be replicated so it does not constitute a single point of failure.

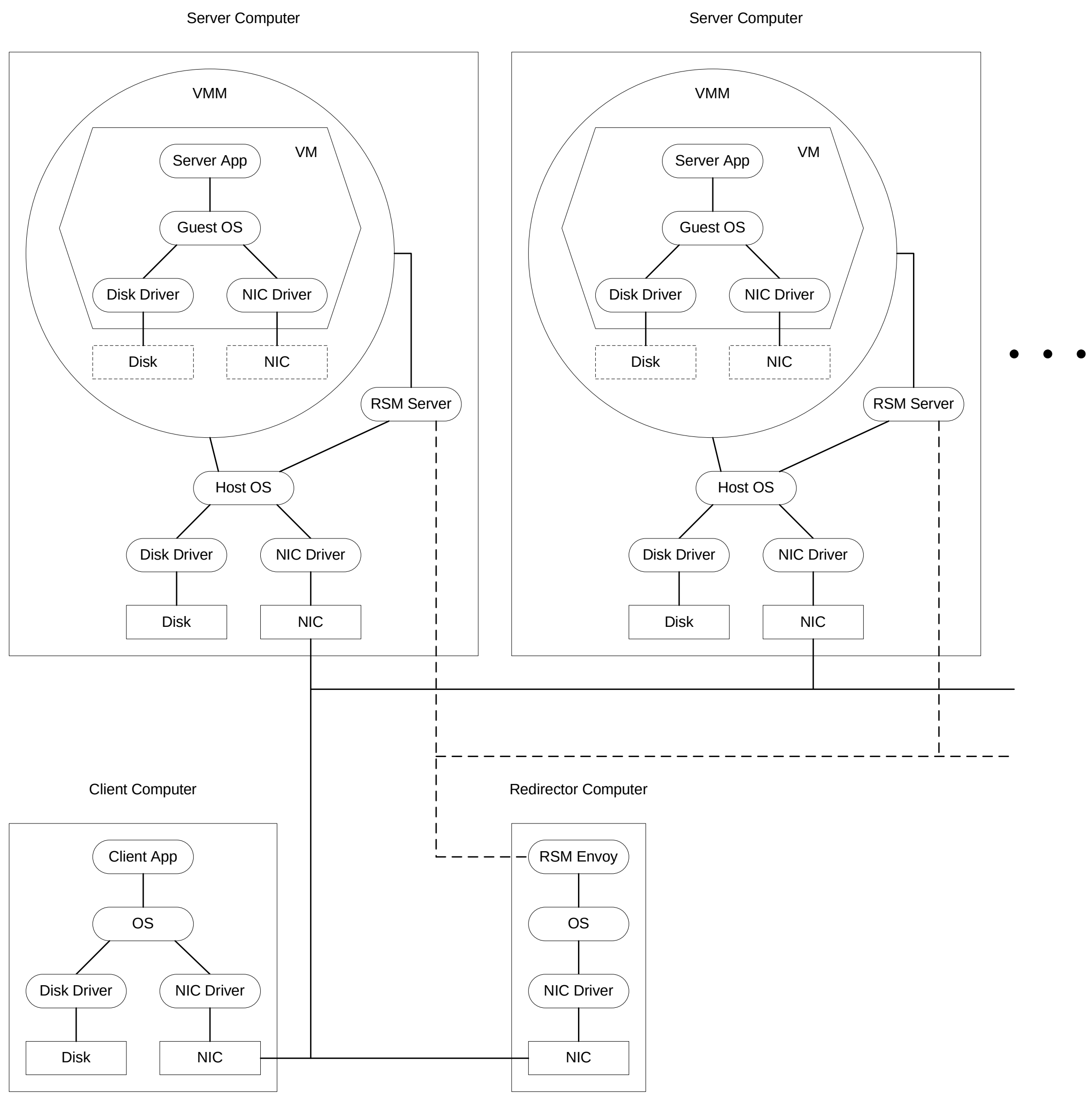

Figure 7. An alternative embodiment of the invention.

The RSM client driver of Fig. 6 and the RSM envoy of Fig. 7 are straightforward adaptations of the prior-art RSM client. The main challenge for the present invention is on the server.

4.1 Turning a VM into a state machine

Since server applications (and the guest OS) are not generally written as state machines, it would be difficult to employ the agreement/execution pattern shown in Fig. 3. Instead, the invention approximates the hardware behavior of a computer: The server app and guest OS execute with apparent continuity, and messages (and other events) arrive in an apparently asynchronous fashion.

To achieve this effect, the agreement protocol of the RSM substrate is used in a slightly different manner from the prior-art usage. Fig. 8 shows an example timing diagram. The downward-pointing arrows indicate requests from clients. The invention partitions time into a sequence of discrete intervals, and within each interval, the agreement protocol determines whether any messages are to be processed and, if there are any, the order in which to process them.

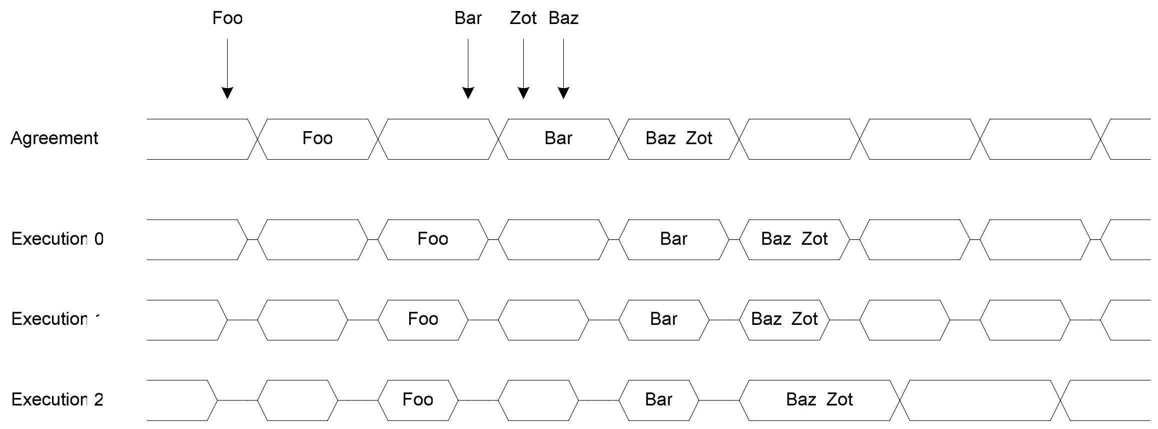

Figure 8. A slightly different timing diagram.

For example, during the agreement interval that begins after message Foo arrives, the substrate decides that the next execution will include message Foo. Since no message arrives during the next interval, the substrate decides that the following execution will include no messages. During that interval, message Bar arrives, and so during the following interval, the substrate decides that the next execution will include message Bar. During that interval, messages Zot and Baz arrive, and so during the following interval, the substrate decides that the next execution will include messages Zot and Baz, and it decides that the order of these messages will be Baz followed by Zot.

Once the agreement protocol completes its decision, the virtual machine is allowed to execute for a determinate length of execution. Typically, this will be measured as a count of processor instructions, but it can be any measure that produces a deterministic result. (In contrast, we cannot use real time for this purpose, because virtual machines on different servers might execute to different points in their code, since the timing of clock cycles and instructions is approximate.)

The specific mechanism by which the VM is allowed to run for a determinate length of execution is determined in part by the processor architecture. If a processor has an interrupt that can be triggered after a certain count of retired instructions, this may be used. If a processor has no direct mechanism for running for a determinate length of execution, then this can be achieved by the following two-phase process: First, allow the VM to run for a length of time that is guaranteed to perform no more execution than the target amount. Second, single-step the VM to the target execution point by setting the processor's trap flag. The first phase is not required for correctness; it is a performance optimization. As a further optimization, the first phase may be repeated, with progressively smaller lengths of time, before switching to the second phase.

If an execution interval includes no incoming messages, then the VMM begins the execution interval by resuming the VM at the execution point from which it was interrupted. On the other hand, if an execution interval includes one or more incoming messages (as determined by the agreement protocol), the VMM first delivers the messages to the VM. It does this by vectoring to the VM’s handling routing for interrupts from the virtual NIC. After the VM completes handling the interrupts for all messages in the current execution interval, the normal interrupt return causes the VM to resume at the execution point from which it was interrupted.

The previous paragraph described how the present invention delivers network interrupts to the VM. However, there are other types of interrupts that also must be dealt with. In particular, there are interrupts from local virtual devices, such as a disk, and also interrupts from the virtual real-time clock. These are described below.

4.2 A deterministic virtual disk

Interrupts from devices are handled as follows (wherein we use a disk as a concrete example). Fig. 9 illustrates a more detailed view of the virtual and physical disk subsystems of the server computer in Fig. 6. When the disk driver in the VM wants to read data from the virtual disk, it programs the virtual DMA with the read request, and it expects to be interrupted after the DMA has transferred the indicated data from the virtual disk into the driver’s memory. The VMM implements this behavior by performing a corresponding read operation to the physical disk, using the physical disk DMA and the physical disk driver, accessed through the host OS.

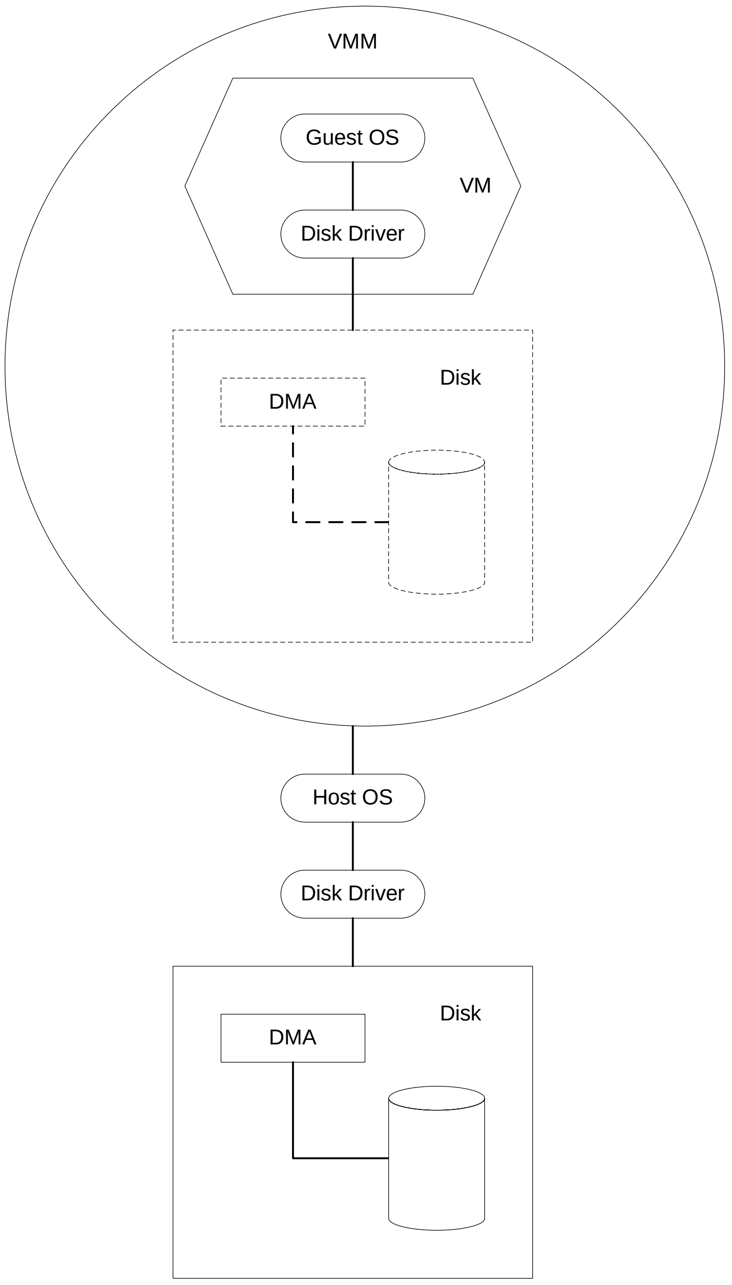

Figure 9. A slightly different timing diagram.

In a conventional VMM, when the physical read operation completes, the VMM interrupts the VM to indicate the completion of the virtual disk read. However, the physical disk will take an indeterminate amount of time to perform the read operation, but the replicated process must exhibit deterministic behavior to satisfy the requirements of an RSM.

Therefore, the present invention handles the DMA interrupt as follows. When the virtual DMA is programmed to perform the operation, the VMM deterministically estimates the length of VM execution that will elapse while the DMA operation is performed. This estimate may be as crude as a constant (e.g., every operation is estimated to take 500,000 processor instructions) or it may be computed based on the size of the data; however, the estimate must be deterministic. Then, using a technique as described in the previous subsection, the VMM interrupts the VM after the indicated length of execution.

If, when the VM is interrupted, the physical read operation has already completed (because the estimate was high), then the VMM delivers the virtual DMA interrupt to the VM at this point. On the other hand, if, when the VM is interrupted, the physical read operation has not yet completed (because the estimate was low), then the VMM pauses the VM and does not resume it until the physical read operation completes, at which point it delivers the virtual DMA interrupt to the VM.

The above technique will work correctly even if the estimate is very high or very low, but the system operates more efficiently with increasing accuracy of the estimate. A high estimate will reduce the disk's data-transfer rate to the VM. A low estimate will reduce the VM's computation rate.

4.3 A deterministic virtual clock

Physical computers typically provide a real-time clock (RTC) register that may be read by the operating system. Physical computers also typically provide a periodic clock interrupt, which is used, among other things, to timeshare the processor among several processes. Within the context of an RSM, all replicas must read identical clock values, and all replicas must be interrupted at the same execution point.

The present invention provides a periodic virtual clock interrupt that is deterministic with respect to the VM's execution. The interrupt is simply triggered after a fixed length of VM execution, using a technique such as that described above. For example, if the VM expects to be interrupted once per millisecond, and if the processor executes roughly 100 million instructions per second, then a clock interrupt is delivered to the VM every 100 thousand instructions. This approach guarantees determinate execution, and it provides interrupts at the required frequency for effective timesharing. However, the interrupts may occur at intervals that are irregular with respect to real time; we discuss this issue further below.

The present invention provides a virtual real-time clock that is deterministic with respect to the VM's execution. The virtual RTC value is simply the value of the VM's execution counter, which may a retired-instruction counter or whatever execution counter is available on the particular processor architecture. Thus, in the case of a retired-instruction counter, if the one-billionth instruction that the VM executes is a read of the RTC, then the value returned will be one billion. If the processor architecture has an execution counter with a small number of bits, such that it risks wrapping, this counter may be extended in software using a well-known technique: The VMM maintains a count of the number of times the VM has been delivered a clock interrupt, and it also records the value of the execution counter at the most recent interrupt. The value of the virtual RTC is then equal to:

IntCount * IntInterval + (ExecCtr – LastExecCtr + ExecCtrLimit) mod ExecCtrLimit

where **IntCount** is the count of the times the VM has been delivered a clock interrupt, **IntInterval** is the length of execution between interrupts, **ExecCtr** is the VM execution counter, **LastExecCtr** is the value of the execution counter at the most recent interrupt, and **ExecCtrLimit** is the limit on the size of the execution counter.

This clock will not track real time very well. If the application requires a better real-time clock, the guest OS in the VM can participate in any standard clock-synchronization protocol, such as NTP, with a computer that has a more accurate real-time clock. The computer that provides the time-synchronization information can either contain an RSM client driver or interact with redirector computer.

4.4 Sending replies to the client

So far, we have discussed only how the present invention implements the *execute* call of the RSM substrate. In the present and following subsections, we discuss the remaining calls.

In a conventional RSM, communication between the client and server has a remote-procedure-call (RPC) structure: The client makes a request; this request is ordered consistently along with requests from other clients; the server executes the request; and the server replies to the client. Thus, the *reply* call is typically invoked once per state update, to send the requesting client a reply to the request that initiated the state update.

The present invention supports arbitrary applications, which may not have been written with an RPC communication structure. The server application might send messages to clients in a manner that bears no obvious relationship to the requests it has received from clients. The present invention handles messages from the server in a straightforward manner: They are sent to the client or redirector immediately. Whenever the RSM client driver or the RSM envoy receives a sufficient count of each message from the servers, it passes the message on to the client application. This technique requires only that message ordering is preserved by the network layer, which it is when using a reliable transport layer, such as TCP.

4.5 Tracking and transmitting state

The RSM substrate needs to track the state of the replicated application. In the context of the present invention, this state includes the state of both the VMM and the VM. Since the VMM is part of the invention's substrate, its state can be handled in the same manner as any RSM's state is handled: The code for this portion of the system must use the *modify* call before it changes any of its state; and it must appropriately implement the *get* and *put* call interfaces. Lastly, the VMM must persistently and atomically record its state in response to a *checkpoint* call. There are well-known prior-art techniques for all of these operations, and they are standard in the world of RSMs.

To track changes to the VM's memory, the VMM sets the protection bits on all of the VM's memory to non-writable. Thus, when the VM executes a write instruction, this execution causes a trap to the VMM. The VMM then uses the *modify* call to inform the RSM substrate that the indicated memory page is being modified. The VMM implements the *get* and *put* call interfaces to the VM's memory by simply reading or writing the indicated page. Lastly, the VMM checkpoints the VM’s memory by recording the values of all VM pages that have been modified.

5 Related Work

The closest related idea, Hypervisor-Based Fault Tolerance [1], differs in that it uses a primary-backup fault-tolerance scheme, and it resolves nondeterminism by determining a specific outcome on the primary machine, and constraining the backup to conform to the same outcome. In the present invention, we arrange the execution of the virtual machine to be deterministic *a priori*, enabling the use of a symmetrical RSM configuration. Our scheme has two benefits: low latency and resistance to Byzantine faults.

Resolving nondeterminism *a priori* permits a lower-latency implementation. In [1], the VM execution must occur first in order to produce the nondeterminism-resolving record, and then the primary and backup must agree on that record, before a result can be safely returned to the client. With the *a priori* nondeterminism resolution of the present invention, the agreement may be overlapped with speculative execution of the VM, and the answer returned once both operations complete.

Using an RSM substrate lets us exploit a number of RSM-based schemes. The obvious fault-tolerance scheme is Paxos [2], but Paxos has many useful relatives which would work equally well in this context. Castro’s Byzantine RSM [3] provides resistance to Byzantine (adversarial) faults; it has been optimized for minimum latency [4]. Fast Paxos [5] is a minimum-latency Paxos implementation, and Generalized Paxos [6] can exploit implicit workload concurrency to improve latency and throughput.

The ReVirt project [7], like the Hypervisor project, measures and records the nondeterministic behaviors of a virtual machine execution, and uses the information to cause a deterministic reproduction of the same behavior in a later re-execution of the same environment. The ReVirt application is distinct: rather than fault tolerance, wherein the replay happens immediately, ReVirt replays the behavior much later, at the request of an intrusion investigator.

6 Conclusion

We describe virtual machine monitor techniques that make a virtual machine behave as a deterministic process, and therefore make it amenable to replication for fault tolerance using replicated state machine techniques. The use of a virtual machine removes the burden of determinism from the programmer, making all existing applications, libraries, and operating system software available for use in the context of replicated state machines. The use of replicated state machines as the fault-tolerance substrate provides the lowest latency, and enables the use of Byzantine-fault-tolerant RSMs and other RSMs with specialized performance characteristics.

7 References

1. Thomas C. Bressoud and Fred B. Schneider. "Hypervisor-Based Fault Tolerance." In *Proceedings of the 15th Symposium on Operating Systems Principles*, p. 1--11. December 1995.

2. L. Lamport. Paxos made simple. ACM SIGACT News (Distributed Computing Column) 32(4):51-58, 2001.

3. M. Castro and B. Liskov. Practical Byzantine fault tolerance. In Proceedings of the 3rd OSDI}, pp. 173-186, 1999.

4. J-P. Martin and L. Alvisi. Fast Byzantine Paxos (extended technical report) The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-04-07. February 2004.

5. Leslie Lamport. Fast Paxos. Microsoft Research Technical Report MSR-TR-2005-112, July 2005.

6. Leslie Lamport. Generalized Consensus and Paxos. Microsoft Research Technical Report MSR-TR-2005-33, March 2005.

7. George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay." In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation*, p. 211--224. December 2002.