Flexible Analysis of Distributed Protocol Implementations using Virtual Time

Elias Weingärtner*, Marko Ritter†, Raimondas Sasnauskas*, and Klaus Wehrle*
†Communication and Distributed Systems (COMSYS) , RWTH Aachen University, Germany
†Plusserver AG, Germany
E-Mail:*{lastname}@comsys.rwth-aachen.de, †m.ritter@plusserver.de

Abstract: **Analyzing the run-time behavior of network protocol implementations in a distributed setting is difficult. However, such analyses are both vital for assuring their functionality and for performance optimizations. Common debuggers typically do not facilitate the inspection of the global state of a protocol implementation that is distributed among several hosts. In this paper we present a virtual machine (VM) based approach for the analysis of distributed protocol implementations. From each of the VMs we extract local state information of choice. By consolidating a global soft-state based on this information and by providing the virtual machines with a virtual and logical progression of time, we facilitate the flexible analysis of x86-based implementations in a distributed fashion.**

1. INTRODUCTION

A major challenge is the development of adequate tools enabling the detailed analysis of distributed network protocol implementations. In general, implementing a network protocol is a difficult task. First, one has to make sure that a *protocol implementation* (PI) conforms to the protocol specification, for example, an RFC. Second, such implementations have to be robust against errors, such as sudden connection losses or the reception of bogus packets. For this purpose, it is necessary to observe their run-time behavior in order to ensure a valid execution for all possible interaction patterns. For the purpose of carrying out such investigations, the ability to closely monitor the distributed execution of a PI is vital. However, this is usually anything but trivial, as the global state of a protocol implementation is distributed among all communication peers.

Standard debuggers like gdb [2] are inadequate for the run-time analysis of protocol stacks. First, they are restricted to user-space applications, which prevents observations of the close interplay of network stacks with other *operating system* (OS) components and network device drivers. While this issue may be overcome by using kernel-level debuggers such as kgdb, debuggers in general are limited to one machine and thus can only deliver information about an implementation's local state. A second major problem with the use of classic debuggers consists in the missing synchronization of the execution with the other communication peers. For example, it might happen that the implementation hits a breakpoint, and thus, the execution is suspended while the execution at the remote communication peers continues. In many cases this may lead to a faulty behavior, for instance due to the expiration of retransmission timers or due to of unwanted connection time-outs.

Full system simulators, for example Simics [9], enable the investigation of an OS' run-time behavior with global state information at hand. Instead of executing the operating system natively, they simulate the entire system hardware and potentially a network of such in software. As the hardware of every communication peer in the network is entirely simulated, one benefits from an unsurpassed degree of control and detail. However, a major disadvantage of full-system simulators is their limited performance and their restricted scalability which directly results from the simulation complexity. In addition the meticulous level of detail is also not needed for most evaluation purposes.

In this paper, we focus on a different approach for the analysis of protocol implementations (PIs). We use *virtual machines* (VM) encapsulating an operating system that hosts a PI of choice. Our framework provides global soft-state information at any point in time during the analysis. We leverage the fact that the VM is fully controlled by a privileged control context that is capable of extracting local state information from the network stack. The local state information is consolidated at a central point in the network in order to form a global view on the execution of all communication peers. Our framework facilitates both interactive and fully automated evaluations by providing the developer a convenient access to the global soft-states using a straightforward scripting interface. We further delineate the concept behind our approach in Section 2 before introducing our Xen-based implementation in Section 3. We also give an example how it can be applied to distributed protocol monitoring. We evaluate our system in Section 4 before we relate our work to other approaches in Section 5. Section 6 concludes this paper with final remarks.

2. A DISTRIBUTED MONITORING FRAMEWORK

2.1 Challenges and Solutions

Scrutinizing the behavior of PIs in a distributed setting is challenging for a couple of reasons. Ideally, such a respective analysis framework delivers consistent state information for all communication peers at any point of time. We need to distinguish between the *local state* of a PI and the *global state* that is constituted by the local state of all peers at the same point in time.

The extraction of local state information turns out to be intricate for the case of PIs. Most importantly, it is vital that the investigation of the run-time behavior is non-intrusive, and hence does not change the way the implementation

executes. Otherwise, odd side effects like the occurrence of heisenbugs [4], that only occur during the analysis run, could be direct consequences. Moreover, the collection of local state information is further aggravated as protocol stacks are typically integrated tightly into the operating systems kernel. We address these challenges by completely abstaining from monitoring the local state on the system that executes the network stack. Instead, we virtualize the entire system and carry out all monitoring operations from an external context governing the virtual machine (VM). This way, the entire extraction of local state information may be performed in a completely transparent fashion.

A major problem with analyzing the run-time behavior of PIs is that local state changes are not only dependent on the causal sequence of the network packets being exchanged but also on internal mechanisms, most notably protocol timers. For example, such protocol timers are commonly used to detect packet loss or connection time-outs. In order to maintain global consistency, it is essential to suspend the execution at all communication peers once one peer's execution is paused for the purpose of closer inspection, say when it reaches a break point. We tackle this problem by virtualizing the entire progression of time at all communication peers. Thus, if the execution of one peer is interrupted, all other systems are paused as well. Since we execute all systems using a virtual and logical continuous time, no communication peer notices such gaps in his execution flow.

In order to obtain a global view on a PI's distributed state, we consolidate local state information from the communication peers. The local state of such a peer is quite bulky, as it encompasses the entire memory as well as all other system resources allocated to the VM. Aiming at an on-line analysis of the distributed execution, the size of the state space prevents a frequent collection of the entire local state from the communication peers. However, only a small fraction of the VM state is usually of interest if one is up to analyzing a PI. For this reason, we only extract selected local state information in order to limit the amount of data retrieved from the communication peers.

A second challenge is the fusion of local state information into a globally consistent view. A well-established method in this context is the use of logical clocks [7]. By associating logical time stamps with every local state, a consistent global state can be consolidated. One example where this concept has been applied to the analysis of distributed applications is D^3S [8]. However, the utilization of logical clocks requires every state change to be propagated. Instead we propose a different approach that utilizes the virtual progression of time at the communication peers for the construction of so-called global soft-states: We assign tiny slices of virtual time to all communication peers. All peers are suspended after they have completed their time slice. The next time slice is assigned after all systems have reported the completion of the time slice. As all communication peers synchronously progress through this series of discrete time slices, we obtain an implicit sequence of global soft-states whose accuracy is determined by the size of the chosen time slices.

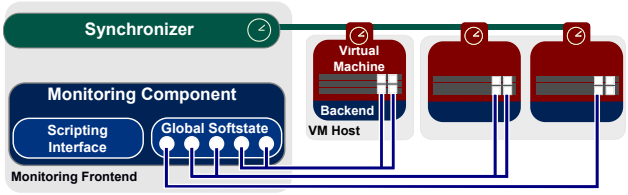

Figure 1: Conceptual Architecture of the monitoring framework

2.2 Architecture

Figure 1 shows the framework that puts our concepts into action. It consists of two main building blocks, the Monitoring Front-end that controls the entire investigation process and the progression of time. The execution of the PIs takes place at multiple VMs, from which we collect local state information using the Monitoring Back-end attached to the VM.

1) **Monitoring Front-end:** This building block encompasses the synchronizer as well as the monitoring component. All analysis tasks are carried out at the monitoring component. It retrieves the local state from all back-ends attached to a VM using remote procedure calls and consolidates a global soft-state. A scripting interface provides a flexible interface to the global soft-state information. This facilitates both automated as well as interactive explorations of the collected state information. In addition, the scripting interface allows the set of investigated soft-states to be modified at run-time. For example, further breakpoints and inspectors may be added in a conditional way, as in case of a particular behavior observed on one of the VMs. Moreover, the scripting interface not only facilitates the passive inspection of state information, but it also allows one to change state descriptors actively. For instance, this allows tuning protocol parameters with "global knowledge" or the simulation of software faults.

The synchronizer assigns small time slices of virtual run-time to the VMs in order to align their local clocks. The next time slice is assigned when all VMs have completed the current one. Thus, the end of every time slice forms a synchronization point for all VMs. We have previously employed this approach for the synchronization of event-driven simulation with VMs. Two aspects are of importance when applying this concept for the purpose of synchronizing the execution of distributed implementations. First, the time drift and hence the elasticity of the global soft-state is bound to the duration of the time-slice. As we aim at a high level of consistency regarding the global soft-state, the duration of the time slices has to be adequately small. Typically, we use time slices between 70 and 200 microseconds. Second, the periodic interruption of all VMs and the synchronization messaging introduces a certain amount of overhead in terms of execution time, yielding to an increased amount of overall real-world run-time for smaller time slices. This is a direct

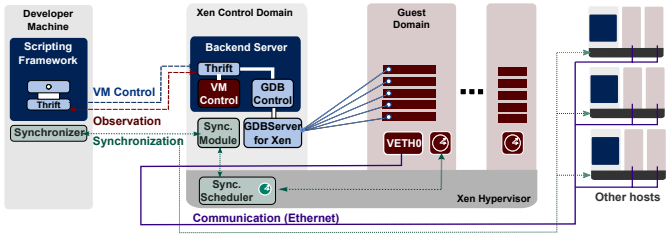

Figure 2: Our framework is implemented on top of the Xen hypervisor and facilitates the analysis and the debugging of multiple VMs using the Python scripting language. The synchronizer, together with a custom Xen scheduler, provides a virtual flow of time to the attached VMs.

consequence of the chosen synchronization approach. It is further elaborated in [12] where we apply it to the synchronization of event-driven network simulations with virtual machines.

2) **Monitoring Back-end:** The monitoring back-end implements all required primitives to support the operations provided to the developer through the scripting interface. The primitives are exposed to the front-end using a standard remote procedure call (RPC) library.

The most important primitives implement the access to local state descriptors on a virtual machine. In order to locate the state information within the memory region allocated to the VM, we access the symbol table of the operating system executed in the VM. This way, we're able to directly access the corresponding state descriptors while avoiding a potential external reimplementation of operating system memory management functions such as the traversal of page tables. Besides the bare access to the VM's memory, the back-end parses the respective memory content in order to provide the developer with a more descriptive representation of the inspected state property.

The monitoring back-end provides further control primitives regarding the VM execution. Basic commands such as PAUSE and UNPAUSE perform the corresponding actions. More sophisticated primitives such as SNAPSHOT allow for storing an entire VM state, which also enables the scripting interface to initiate global distributed snapshots upon the observation of certain behavioral patterns.

Besides the provision of the required access and control primitives, the back-end together with the VM environment needs to support the time-slice based execution required by the synchronization scheme. For this purpose, the VM environment executes a virtual machine for the exact duration of the time slice. In addition the VM environment has to provide the VMs with a virtual progression of time that is aligned to the provisioned time slices.

3. IMPLEMENTATION

Figure 2 depicts our implementation corresponding to the proposed architecture. We distinguish between the developer machine that exposes the global soft-state within a scripting environment and VM hosts executing multiple VM guests on top of a customized Xen [1] hypervisor. The VM hosts and

```
class hostA_thread(threading.Thread)):
 def run(self):
  while True:
   hostA.wait()
   pp.pprint("host A: *skb:" + hostA.getVariable("*skb",))

class hostB_thread(threading.Thread)):
 def run(self):
  while True:
   hostB.wait()
   pp.pprint("host B: *skb:" + hostB.getVariable("*skb",))

bpA = hostA.setBreakpoint("icmp_rcv",)
bpB = hostB.setBreakpoint("icmp_rcv",)
```

Listing 1: Monitoring the exchange of ICMP frames

the developer machine are interconnected using three different flows of communication. First, a combined observation and control flow based on Apache Thrift [11] delivers state information to the developer machine and facilitates controlling the VM execution behavior using the scripting framework. Second, the provision of time slices takes place using a separate synchronization flow. It delivers time slice information to the VM hosts using a lightweight UDP messaging scheme, minimizing the complexity due to the potentially high amount of synchronization messages. A custom Xen scheduler then executes the VM guest for the specified time slice duration. In our previous work [12] we elaborate how Xen was modified for the purpose of synchronization in greater detail. A third communication flow comprises an Ethernet tunnel in order to enable the communication among the protocol stacks hosted on multiple VMs.

3.1 Scripting Environment

The Python-based scripting framework drives the entire inspection process. At any point in the logical flow of time, the individual local states of all communication peers are accessed using a stub that allows for setting breakpoints or for reading and modifying state descriptors using the common gdb syntax. Listing 1 illustrates how our framework can be applied to monitoring the reception of ICMP packets at two Linux hosts. Due to the lack of space we omit the initialization block that establishes the observation and control flow to the sender and to the receiver. The example uses the setBreakpoint primitive to interrupt the execution at both hosts upon the reception of an ICMP packet, e.g. an echo request. It employs two threads in order to cope with the parallel execution of the sender and the receiver. Using the wait() command, each thread waits until one system hits a break-point. In this case, the script outputs the content of the socket buffer. Other commands not used in this example allow for snapshotting a VM and facilitate a manual control of the VM execution.

We further emphasize that the entire process of synchronization is transparent to the scripting environment. The provision of small time slices automatically establishes a series of global soft-states, formed by the individual local states for a particular time slice. As a system is not able to signal the completion of its time slice upon hitting a break-

point, the execution at all other communication peers is also suspended at the end of the current time slice.

3.2 Back-end Server

The core task of this component is to deliver local state information to the scripting front-end. For the purpose of accessing local state information from the Xen domains, we rely on Gdbserver-xen [6]. Gdbserver-xen provides a gdb interface to the kernel being executed inside the VM. It relies on the available symbol table information in order to locate symbols, e.g. protocol state descriptors, in the memory range of the executed kernel. Gdbserver-xen then internally translates the pseudo-physical addresses in the symbol table to the corresponding addresses in the physical host's address range. For the purpose of providing a convenient and efficient access to kernel-level state descriptors, gdbserver-xen maps these memory ranges to the address space of the privileged control domain (domain 0).

Besides the exposure of state information, the Back-end Server also implements a set of control primitives. For this purpose, our implementation executes so-called hypercalls to directly control the state and the scheduling behavior of the VMs. In addition, few operations such as snapshotting are implemented by invoking the corresponding calls to the Xen Management daemon.

4. EVALUATION

We now evaluate our system regarding the valid provision of virtual time to the VMs. We also investigate the capability of our framework to externally monitor state changes of a Linux TCP implementation at the packet granularity level. All experiments were carried out using two Dell Optiplex 960 PCs, each equipped with a 3GHz Intel Core2 Quad CPU and 8 GB of RAM. Each machine executed our Xen-based implementation of the monitoring framework. One machine in addition hosted the monitoring framework and the synchronizer. For all experiments the synchronization accuracy was set to 100µs, at which the run-time overhead amounts to 42% compared to real-time for one Xen domain.

4.1 Provision of virtual time

In order to validate the provision of virtual time to the VMs, we used a straightforward monitoring script to suspend the execution of a VM for a specified amount of time upon the reception of an ICMP frame. In order to trigger this behavior, ICMP echo replies were sent to the local host at a constant rate. On the VM, we measured the average RTT of the Echo replies. Figure 3 displays the result: As expected, for non-synchronized VMs the measured RTT increases for longer execution pauses, as ICMP internally utilizes timestamps to conduct round-trip measurements. By contrast, if we use our implementation to supply a VM with a virtual progression of time, the measured round trip times show the desired invariance to external interruptions of the VM execution.

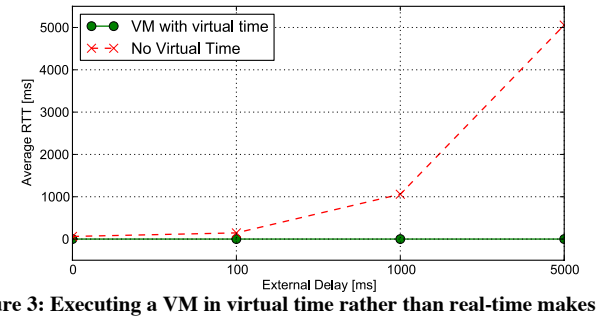

Figure 3: Executing a VM in virtual time rather than real-time makes RTTs on the local host invariant to externally imposed execution gaps.

4.2 Monitoring Accuracy

As the state of a PI may change with every packet sent or received, it is vital to accurately capture state changes at the granularity level of one packet. For the purpose of evaluating our implementation against this requirement, we applied our framework to the TCP implementation of Linux 2.6. One of the TCP protocol states that are subject to change with every packet is the TCP receive window. We are aware that our VM based monitoring approach is not required for tracing the receive window of a TCP connection, as the receive window is also part of the TCP header and thus may also be observed using a packet capturing tool like Wireshark [14]. However, the presence of this information within the TCP packet header allows to use it as a reference value and thus allows for investigating the accuracy of the state information gathered with our framework.

Consequently we used our framework to monitor the receive window on two VMs exchanging data over a bi-directional TCP connection. Figure 4 compares the window sizes extracted from the VMs using our framework with the reference window sizes as reported by Wireshark. The window sizes extracted from the VMs well match the reference values. From this we conclude that our approach enables observations of protocol descriptor state changes at the granularity of one packet.

5. RELATED WORK

The related efforts and challenges in the area of distributed protocol monitoring fall into two main categories: protocol state exposure and global snapshot creation. While the access of protocol state information in operating systems is widely supported by a number of well-established debugging tools such as gdb and Liblog [3], the transparent creation of consistent global snapshots remains to be a challenge. In the remainder of this section, we discuss the existing approaches for deployed protocol monitoring and relate our approach.

In [8] the authors present D^3S, a checker which logs distributed protocol states using binary code instrumentation in user-space. The defined watch points of interest redirect the active program execution to protocol-dependent state exposure routines that subsequently transmit the state information over the network to a global verifier. Although

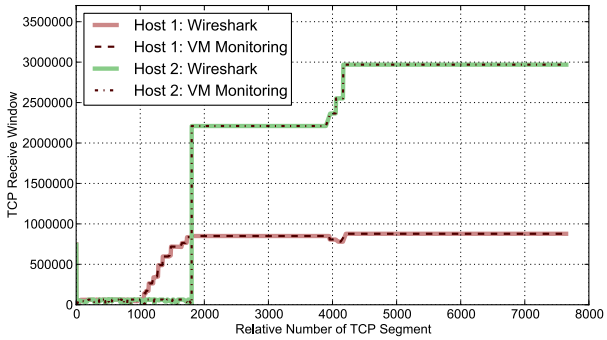

Figure 4: Our framework accurately captures state changes of kernel-level PIs at packet granularity. The state information extracted from the VM well matches the reference values obtained from packet traces.

the performance overhead of less than 8% is reasonable, the intrusive binary instrumentation is bound to one specific operating system. Moreover, the correctness of the global snapshot creation mechanism strictly depends on the messages containing virtual timestamps. D^3S is a post-mortem debugger. Once a distributed assertion is violated, it is not possible to suspend the distributed system execution for further analysis of neither the protocol nor its environment. On the contrary, our monitoring framework offers full control of VMs along with running protocols making distributed event ordering redundant.

PDB [5] employs Xen to debug both user-space and kernel-space software of guest operating systems running on a single physical machine. PDB provides an extensive interface for debugging information access and visualization. In contrast to PDB, our approach spans over a number of physical machines using synchronized virtual time. By suspending and resuming individual VMs we not only synchronize their distributed execution but also preserve the timers that steer the core of PIs. Moreover, a developer is supported with an intuitive script language to formulate and check high-level protocol predicates.

One alternative to using gdbserver-xen is XenAccess [10], which we used in our preliminary work [13] to inspect protocol state descriptors. XenAccess also provides external access to symbol information. However, major drawback of XenAccess is its fixed reimplementation of the memory management of the guest OS to resolve the symbols. This makes it rather inflexible to apply to different operating systems or kernels varying in the way they organize the system memory.

6. CONCLUSION

In this paper we provided an in-depth discussion of our framework allowing for the analysis of distributed network protocol implementations. It incorporates the concept of VM monitoring with the provision of a consistent and virtual progression of time to a set of virtualized communication peers. This relieves the investigation of the distributed run-time behavior from any real-time constraint. Hence, it becomes possible to transparently carry out extensive analysis tasks like the deep exploration of a network protocol's state space or complex operations such as snapshotting an entire system.

We conclude that our framework eases the analysis of the distributed execution of network PIs. First, our framework automatically delivers the developer with a series of global soft-states, in which any aspect of a PI may be investigated using a flexible scripting environment. Second, we emphasize that the application of our framework for the purpose of monitoring a kernel-level application is straightforward. Hence, we regard it to be a very supportive tool for both researchers and developers that interested in investigating the execution of distributed protocol implementations.

ACKNOWLEDGEMENTS

This work is partly funded by the German Research Foundation (DFG). The authors thank Hendrik vom Lehn and Suraj Prabhakaran for their helpful comments.

REFERENCES

[1] 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, Bolton Landing, NY, USA, Oct. 2003. ACM.

[2] Free Software Foundation. GDB documentation: (the GNU source-level debugger). http://www.gnu.org/software/gdb/documentation/ (accessed 08/16/2010).

[3] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In ATEC '06: Proceedings of the annual USENIX ' Annual Technical Conference, pages 27–27, Berkeley, CA, USA, 2006.

[4] J. Gray. Why do computers stop and what can be done about it? In Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems (SRDS'86), Los Angeles, CA, USA, 1986

[5] A. Ho, S. Hand, and T. Harris. Pdb: Pervasive debugging with Xen. In GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, pages 260–265, Washington, DC, USA, 2004.

[6] N. Kamble, J. Nakajima, and A. Mallick. Evolution in Kernel Debugging using Hardware Virtualization With Xen. In Proceedings of the Ottawa Linux Symposium, 2006.

[7] L. Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7):558–565, July 1978.

[8] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3S: Debugging deployed distributed systems, Proceedings of the 5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, San Francisco, CA, USA, 2008

[9] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. IEEE Computer, 35:50–58, 2002.

[10] B. D. Payne and W. Lee. Secure and flexible monitoring of virtual machines. In ACSAC, pages 385–397. IEEE Computer Society, 2007.

[11] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical Report, Facebook Inc., 2007.

[12] E. Weinga¨rtner, F. Schmidt, T. Heer, and K. Wehrle. Synchronized network emuqlation: Matching prototypes with complex simulations. In 1st Workshop on Hot Topics in Measurement & Modeling of Computer Systems (HotMetrics'08), Annapolis, MD, June 2008.

[13] E. Weinga¨rtner, C. Terwelp, and K. Wehrle. PromoX: A protocol stack monitoring framework. In Proceedings of the GI/ITG KIVS Workshop on Overlay and Network Virtualization 2009, Kassel, Germany, 2009.

[14] Wireshark. A network protocol analyzer. http://www.wireshark.org/, (accessed 08/16/2010).