System Support for Elasticity and High Availability

by

Shriram Rajagopalan

M. S., Computer Science, University of California, Santa Barbara, 2009
B. Tech., Computer Science, National Institute of Technology, Trichy, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University Of British Columbia
(Vancouver)

March 2014

Abstract

Elasticity and high availability (HA) are key requirements among modern Internet applications. Elasticity enables applications to dynamically allocate/release physical resources in proportion to request load. High availability enables applications to mask failures in the system from end users. In current practice, every application implements these features as part of its own application logic, resulting in unnecessary design complexity. This thesis argues that facilities for elasticity and HA should be exposed as system-level primitives, in the same way abstractions for files and networks became operating system-level primitives three decades ago. Unfortunately, providing these higher-level services efficiently may require knowledge of application data structures, consistency requirements, and workloads. This thesis describes initial instantiations of such interfaces for two broad (and different) classes of applications: network middleboxes (e.g., load balancers, intrusion prevention systems, etc) and database systems.

Elasticity is achieved typically through dynamic partitioning of state and inputs into independent subsets, while HA is achieved through state replication. Guided by this principle, this thesis presents a system-level runtime that partitions the middlebox state along flow boundaries and provides abstractions for elasticity and HA using live migration and replication of flows respectively. For database systems, this thesis presents a hypervisor-level HA system that performs database-aware virtual machine replication, eliminating the need for complex application-level HA mechanisms.

This thesis concludes that while there may not be a one-size-fits-all solution to application elasticity and HA, it is still feasible and beneficial to provide system-level primitives that are applicable across one or more application domains.

Preface

This dissertation draws on my research associated with the following publications:

P1. SHRIRAM RAJAGOPALAN, DAN WILLIAMS, HANI JAMJOOM, AND ANDREW WARFIELD. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

P2. SHRIRAM RAJAGOPALAN, DAN WILLIAMS, AND HANI JAMJOOM. Pico Replication: A High Availability Framework for Middleboxes. In *Proc. of ACM Symposium on Cloud Computing (SoCC)*, 2013.

P3. UMAR FAROOQ MINHAS, SHRIRAM RAJAGOPALAN, BRENDAN CULLY, ASHRAF ABOULNAGA, KENNETH SALEM, AND ANDREW WARFIELD. RemusDB: Transparent High Availability for Database Systems. *The VLDB Journal*, 22(1), 2013.

P4. SHRIRAM RAJAGOPALAN, DAN WILLIAMS, HANI JAMJOOM, AND ANDREW WARFIELD. Escape Capsule: Explicit State is Robust and Scalable. In *Proc. of USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2013.

A version of Chapters 3, 4 & 6.2 has been published in P1, P2 & P4 respectively. I was the lead investigator responsible for all aspects of the research, while Dan Williams and Hani Jamjoom from IBM Research, and my adviser Andrew Warfield acted as supervisory authors, contributing to system design and manuscript composition.

A version of Chapter 5 has been published in P3. Umar Farooq Minhas (University of Waterloo) and I were co-investigators on this project while Ashraf Aboulnaga, Kenneth Salem from University of Waterloo, and Brendan Cully, Andrew Warfield from UBC acted as supervisory authors, contributing to system design and manuscript composition.

I have obtained permission from relevant sources to reuse content in this dissertation.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to extend my sincere gratitude to my supervisor Andrew Warfield for his unwavering support and technical guidance. I am grateful to Andy for giving me complete freedom to pursue my research interests, and for providing me with candid, in-depth feedback in a constructive manner that has always left me feeling highly motivated and happy, even during difficult times of my research. I have come a long way in my research career over the past four years and a lion's share of the credit goes to Andy.

I would like to thank Dan Williams and Hani Jamjoom from IBM T.J. Watson Research for their phenomenal mentoring during my year long stint at IBM Research. They helped me improve my writing, presentation and project management skills. I would also like to thank Norman C. Hutchinson and Michael J. Feeley for serving in my supervisory committee, and their valuable feedback on my dissertation proposal and initial drafts of this thesis. Mahdi Tayarani Najaran also provided helpful comments on initial drafts of this thesis. I would like to acknowledge my fellow grad students in the NSS lab for making my stay at UBC memorable.

I would like to thank my family, especially my mom, dad, and my cousin for their love, encouragement, and moral support during times good and bad. Last but not the least, to my loving partner Divya for teaching me the art of maintaining a work-life balance, for believing in me and standing by my side at all times, thank you.

Chapter 1

Introduction

Successful Internet applications are elastic and highly available. These applications scale-out and scale-in dynamically according to the request load, while maintaining an optimal overall system utilization. They mask failures in the underlying infrastructure from users, while maintaining connectivity and consistency of data. In order to achieve these capabilities, an application needs to manage resources and runtime state in a dynamic fashion. While cloud platforms have made it easy to provision resources dynamically, the onus of managing runtime state during events such as scale-out or failover, is still on the application. For example, cloud platforms provide facilities to automatically add new resources, such as virtual machines, to the application cluster in response to increase in user request load [60, 139, 148]. However, it is still the responsibility of the application to take advantage of those new virtual machines to handle the load in an efficient manner [105, 110, 161].

Elasticity is the ability of an application to maintain scalability through dynamic resource allocation instead of static over provisioning. **High availability (HA)** is the ability of an application to survive failures in the underlying cloud infrastructure, without disrupting the service being provided to end users. Together, these properties enable a cloud application to become **adaptive** to changes in load and failures, while maintaining performance and resource utilization.

In practice, building "elastic and highly available" software has proven to be very challenging, partially due to the complexity of developing applications that

may have to scale across varying resources, and even out across large numbers of computers. Select companies such as Google, Facebook, Twitter, Amazon, etc., possessing both money and expertise, have demonstrated that it is indeed possible to achieve these two properties at Internet scale. However, enterprise applications and commodity software are still struggling to achieve the same at the application layer [40,119,122,123]. Applications are constantly forced to "reinvent the wheel" in an attempt to become adaptive. Instead, if these facilities are exposed as system-level primitives, application designs can be greatly simplified, enabling a larger set of commodity applications to function successfully at scale. As an analogy, consider the introduction of abstractions for network communications into operating systems three decades ago. Rather than forcing every application to develop its own network stack, the OS abstracted the common functionality in a manner that dramatically simplified the design of networked applications. In a similar vein, it is the position of this thesis that similar to other systems-level facilities, like the implementation of protocol stacks, abstractions for elasticity and high availability are universal requirements for a broad class of applications; they should be implemented in a supported and trustworthy manner by a central source.

The unfortunate reality is that, at least from an interface perspective, there is no silver bullet to generalized elasticity and availability; no single set of abstractions can satisfy the needs of the diverse set of applications that exist in the cloud today. This poses a logistical hurdle, since it is untenable to customize the system to each application's need. However, there is a silver lining: workloads in the same domain exhibit similar characteristics in terms of their high level design, scalability, and consistency requirements, and there are a finite set of application domains to consider. This thesis takes concrete steps in the general direction of domain-specific approach to adaptive software by developing abstractions for two broad classes of applications: network middleboxes and database management systems.

Middleboxes [120, 125–130] comprise a critical piece of data center infrastructure. Recent efforts from both the industry and academia have focused on re-architecting middleboxes such that they are more service oriented [41, 132, 139, 146], composable [92] and extensible [8]. In this new vision, services like protocol acceleration, load balancing, and intrusion detection/prevention can be easily customized and managed to match the needs of the flows in the data center.

Despite the renewed interest in middleboxes, dynamic scalability (elasticity) and high availability support is limited and requires that each middlebox application implements its own mechanisms and polices to scale and tolerate failures. This work re-examines system support for adaptive services in middleboxes. Based on the observation that the flows in a middlebox are independent of one another, a flow-centric system framework for developing middlebox applications is presented. This platform has two components: *Split/Merge* (Chapter 3) and *Pico Replication* (Chapter 4). Split/Merge enables middleboxes to scale elastically in a load balanced fashion through flow migration. Pico Replication provides low overhead, and customizable fault tolerance through flow granular replication and a set of abstractions to build application specific replication policies.

A large class of web applications, from large scale social networking sites like Facebook to e-commerce sites, rely on traditional database management systems (DBMS) for managing their persistent data. In a cloud environment, a DBMS running in a virtual machine can take advantage of different services and capabilities provided by the virtualization infrastructure such as live migration [25], elastic scale-out, and better sharing of physical resources. These services and capabilities expand the set of features that a DBMS can offer to its users while at the same time simplifying the implementation of these features. However, due to limited system support for HA, most deployments resort to complex application level logging and replication techniques [24, 159, 166]. This work presents system level techniques that capitalize on the memory access patterns of database servers and the transactional properties of a DBMS to provide low overhead and cost effective high availability to commodity database systems.

To summarize, based on the premise that system abstractions for adaptability simplify application code, it is the thesis of this work that, while universal interfaces may be infeasible, applications in the same domain exhibit similar characteristics that can be exploited to provide domain-specific abstractions. Since the number of application domains is finite, this work argues that a domain-specific approach to providing system-level adaptability services is a feasible and sensible alternative to universal interfaces. Towards this goal, this work presents domain-specific abstractions for databases and network middlebox applications. The rest of this dissertation is organized as follows: Chapter 2 explores in detail, the need

for system-level primitives for elasticity and high availability in modern applications. Chapters 3 & 4 describe the design and implementation of these system-level abstractions for network middleboxes, while Chapter 5 describes a system-level solution for database systems. Chapter 6 revisits the problem under the influence of developing such abstractions and explores their applicability to systems other than middleboxes and databases. Finally, Chapter 7 concludes the dissertation.

Chapter 2

Background & Motivation

In this chapter, we characterize the state of the art approaches to elasticity and HA (Section 2.1). Later, we question the possibility of developing a unified set of system primitives for these facilities. The challenges involved are discussed in Section 2.2. Continuing down this path, in Section 2.3, two different applications are chosen for further exploring the idea of system level abstractions for elasticity and HA.

Elasticity and high availability (HA) are key properties that applications should have in order to operate successfully in a cloud computing environment. Since cloud platforms charge applications only for resources they actually use, applications that are elastic can now scale on demand while reducing overall operating costs. At the same time, the commodity nature of the compute infrastructure coupled with the sheer scale of modern cloud environments has resulted in frequent infrastructure failures [67, 133–135, 141, 147]. This low-cost unreliable compute environment necessitates that applications should be "designed for failure" as opposed to considering failures as rare events. Applications that do not possess these abilities may incur higher costs due to poor resource utilization or suffer prolonged downtimes during outages in the infrastructure.

2.1 Characterizing Adaptive Applications

We use the term "*adaptability*" to collectively denote the two desirable properties of cloud applications: elasticity and HA. In this section, we analyze each of these properties in detail and characterize existing solutions accordingly. We find that applications need to carefully manage both their resources (e.g., VMs) and their runtime state in order to become adaptive.

2.1.1 Elasticity

Elasticity, as defined by Herbst et al. [51], is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at any point in time the available resources match the current demand as closely as possible.[1] Based on this definition, the following are the main goals than an application needs to achieve in order to become elastic:

1. **Scale-out.** In a cloud environment, the process of scaling out involves addition of resources, namely VMs, to an application cluster when the load on the application starts to exceed its current processing capacity. In order to effectively utilize the newly acquired VMs, the application has to carefully distribute incoming request load over its scaled out deployment while ensuring that each VM in the cluster has access to the required runtime state needed to process a given request.

2. **Scale-in.** The process of scaling in involves release of excess processing capacity back to the cloud when the load on the application begins to fall below its current processing capacity. Merely releasing resources such as VMs is not enough as they may still be processing requests. To scale-in quickly and statefully, applications need the ability to consolidate runtime state into fewer VMs and distribute incoming request load accordingly. Stateful scaling implies that on-going requests should continue to be processed while the application consolidates its load.

[1] N.B. Elasticity is not the same as scalability. Scalability is the ability of an application to sustain increasing workloads without performance degradation, assuming that additional resources are added. Scalability is a prerequisite to elasticity.

3. **Rebalance load transparently.** In addition to being able to scale-out and scale-in, an application needs to be able to dynamically rebalance its load across the deployment to achieve load balanced elasticity. For example, when one of the nodes in a cluster becomes overloaded, the application needs to eliminate the hotspot by "shedding load" from the overloaded node onto other nodes. The load rebalancing event should be transparent to clients. In internet facing applications that are generally structured as a composition of client sessions, the rebalancing process involves migrating live sessions from one node to another. During this migration, all state associated with a user's session (e.g., session data, TCP connections, etc.) need to be shipped to the target node without interrupting or terminating the connectivity to the user.[2]

In addition to the above goals, solutions that provide system-level elasticity can be characterized using the following properties:

4. **Horizontal vs vertical scaling.** There two ways of adding resources to an application for scalability: horizontal and vertical scaling. The cloud deployment model typically emphasizes horizontal scaling where applications scale-out in a distributed fashion through the addition of VMs to their deployment. In the vertical scaling model, a single application node is scaled-up by adding more resources such as CPU, memory, etc., to the VM, thereby increasing the processing capacity of the application.

5. **System managed state.** From an application designer's perspective, a desirable property of a system that provides elasticity abstractions is that it should not require application-level logic to (re)distribute input load and runtime state as the underlying infrastructure footprint changes dynamically in response to load. The system should dynamically scale the underlying resources as well as rearrange the application's inputs and runtime state during elasticity related operations.

[2]Short lived sessions on the order of a few seconds eliminate the need for such migration facilities. However, the duration of a session is highly application-specific varying from few seconds to hours even for popular web applications [36, 62].

	Horizontal scaling				
	Scale-out	Scale-in	Rebalance load transparently	Vertical scaling	System managed state
Process migration [11, 14, 35, 63, 68, 74, 82, 98]				✓	✓
VM migration [15, 25, 50, 52, 57, 101, 104, 107, 117]				✓	✓
VM auto scaling (IaaS) [17, 60, 148, 149]	✓	✓			
PaaS Frameworks [138, 143, 144]	✓	✓			
Special purpose applications [2, 23, 31, 34]	✓	✓	✓		

Table 2.1: Characterizing the state-of-the-art solutions for elasticity

Table 2.1 classifies existing solutions based on the above criteria. Process and VM migration based systems take the vertical scaling approach (i.e., scale-up or scale-down) to achieve on-demand scalability. The typical strategy is to oversubscribe physical resources such that several processes (or VMs) are multiplexed over a minimal set of physical resources. When one or more VMs experience high load, they are live migrated [25] to a dedicated host with more physical resources (e.g., additional CPU cores, memory, etc). Vertical scaling strategy is typically not a preferred option since the maximum scalability of an application is limited by the processing capacity of any one single physical machine inside the data center.

Cloud environments prefer the horizontal scaling approach where an application's scalability is limited by the combined processing capacity of the entire data center. For example, in Infrastructure as a Service (IaaS) cloud offering from Amazon, Microsoft, etc., system primitives such as Auto Scaling [148, 149], and

SnowFlock [60]'s *VM fork* abstraction enables a distributed application to add/remove VMs in a dynamic and application agnostic fashion. When coupled with a load balancer such as Amazon's Elastic Load Balancer [139], this technique enables applications to balance incoming load across the cluster. Platform as a Service (PaaS) frameworks [138, 143, 144] scale applications similar to auto scaling techniques. For example, the Heroku PaaS platform encapsulates application processes inside isolated virtualized OS containers [145] akin to FreeBSD's jails. New application instances can be launched to handle increased load. However, neither IaaS or PaaS scaling approaches provide the capability to rebalance existing load, i.e., transparently migrate live sessions, from one machine to another in order to shed load or consolidate load across several lightly loaded machines.

Enterprise applications have traditionally used a combination of application clustering, middleware platforms [21] and application layer logic to maintain a load balanced cluster. When rebalancing load on the cluster, client connections are typically dropped from the overloaded node. When clients reconnect, they are assigned to a different node that has access to the application layer session data through a distributed cache [136, 142, 172]. Static application clustering, unlike modern cloud application design, is a complex process: it requires careful capacity planning, complex integration with program logic, and ongoing maintenance as the application code evolves over time. Recent research has also focused on designing data center scale systems such as Dynamo [34], BigTable [23], etc., where consistency properties are sometimes relaxed to support the massive scale. In these special purpose applications, adaptability features are tightly coupled with the application logic, resulting in a highly customized software stack design that is unsuitable for commodity applications. Often times, these applications interact with custom client programs that cooperate with the application during the rebalancing process.

2.1.2 High Availability

High availability (HA) characterizes the ability of an application to avoid loss of service to end users by reducing or managing failures in a transparent fashion. This section describes the failure model assumed in this thesis, the associated challenges

in providing HA to withstand such failures, and a characterization of related work in this area.

Failure model. This thesis focuses on providing **protection from fail-stop failures**, a typical cause of outages in the data center. These include provider hardware failures, power outages, destructive events such as fires, etc., that result in service downtime from an end user perspective. Network failures can also be promoted to be fail-stop through the use of a watchdog that automatically removes the active host from service in the case of network disconnection. This approach helps maintain availability when a data center is experiencing site-wide connectivity problems by transparently migrating to a better-connected backup during the network outage. The failure model assumed in this thesis does not attempt to survive "wounded" applications experiencing partial failures as a result of overload, software bugs, internal data corruption caused by hardware such as memory bit flips, etc.

Based on the fault model described above, the following are the main goals that an application needs to achieve in order to become highly available:

1. **Client transparent failover.** A highly available application should mask failures (and recovery) completely from end users. Thus, when any node in the application cluster fails, any further requests to the failed node should be redirected to the remaining nodes in the cluster. With the help of a load balancer, it is relatively easy to remove unhealthy nodes from the cluster and redistribute new incoming connections/sessions to the remaining nodes.

2. **Connection failover.** A subtle but harder problem that is often overlooked with regard to failover is recovering sessions that were being serviced by the failed node when the failure occurred. Since a client application cannot distinguish between node failure and link failure, it would continue trying to send packets to the failed node until its protocol stack (usually TCP) decides to terminate the connection. Ideally, recovery should be such that ongoing TCP sessions with clients should continue without disruption. To achieve this transparent failure recovery, the application needs to maintain a hot-standby copy of the relevant pieces of its runtime state at a backup server, such that when a primary node fails, the backup can immediately take charge

without exposing the failure to the client.

3. **Performance overhead.** Failures, while costly are also relatively rare. Adding HA to any application introduces some performance overhead, due to the need to backup the application's state in a consistent manner. While the performance overhead of providing HA cannot be avoided, it should be kept minimal in order to maintain overall scalability of the system. The overhead of HA typically manifests in the form of increased latency between the client and the server. When client-server interaction is latency sensitive (e.g., transactional workloads), the latency overhead also impacts the throughput of the system.

In addition to the above goals, a system-level HA abstraction should possess the following additional property:

4. **Application transparent recovery.** When providing HA as a system abstraction, the application's involvement during failure recovery should be minimal or none. Ideally, the system should maintain necessary backup copies of the runtime state and reincarnate it during failure recovery in such a way that the application merely resumes execution with minimal interruption.

Table 2.2 characterizes existing HA solutions based on some of the criteria described above. Performance overhead of various techniques depends on the workload. They are noted, where applicable, in the text that follows.

The typical approach for failure recovery in many environments today involves the asynchronous replication of storage [77,86,116]. In the event of failure, storage-based recovery systems (a) lose some amount of data and (b) require that the hosts reboot successfully from a crash-consistent image. Clients have to re-establish connectivity to the newly launched application. In addition to OS level recovery (e.g., file system checks), the application may also need to perform additional recovery (e.g., replaying transaction logs).

There are several open-source and commercial solutions [111, 113, 176] that provide automatic failure detection, recovery and cluster resource management services. Failure recovery involves rebooting the VM from a crash consistent snapshot

	Transparent recovery			
	Client	Application	Connection failover	Notes
Asynchronous storage replication [77, 86, 116]				Longer downtime
HA clustering [111–113, 139, 160, 176, 180, 182]	✓			Recovers only resources, not state
VM checkpoint replication [27–29, 79]	✓	✓	✓	Unsuitable for latency bound workloads
VM execution record and replay [5,16,37,42,61,76,89]	✓	✓	✓	Unsuitable for shared memory multiprocessor workloads
Special purpose applications [23, 34, 73, 136]	✓			

Table 2.2: Characterizing the state-of-the-art solutions for high availability

present in a shared storage medium. Ultimately, it is the responsibility of the application to complete the recovery. Stateful application clusters deployed using middleware solutions [21, 161, 167] need special application level logic for recovery. In case of stateless applications that are scaled elastically (e.g., using Auto Scaling [148, 149]), there is little to no recovery task required at the application layer. A load balancer, such as Amazon's Elastic Load Balancer [139], monitors the health of application nodes. When a node fails, it is removed from the load balancer's pool; incoming requests are redistributed among the healthy nodes. However, clients connected to the failed node lose connectivity.

In VM checkpointing systems like Remus [28], periodic incremental checkpoints of the state of an entire VM (memory, disk, CPU context, etc) are replicated to a backup host. In order to preserve consistency of VM state during failure recovery, the checkpoint replication process uses the output commit [100] principle. Output commit delays the release of outputs from a VM (e.g., writes to disk, responses to clients, etc.), until the checkpoint associated with the outputs is com-

mitted at the backup. In the event of failure, no data is lost, and reboots, with associated file system checks, are not required. Failover is transparent to the client, with no loss of end-to-end connectivity. Unfortunately, the use of output commit also results in a high performance overhead for latency sensitive applications, as network responses from a VM to its clients are delayed until the end of a checkpoint period. Process based checkpoint-restart systems incur equally high overhead during normal operations [59].

Execution replay based systems take the state machine replication approach [90] conceptually using two VMs. Both primary and backup VMs execute in lock-step manner, processing exactly the same set of inputs and events in the same order. This technique requires the system to manage all sources of non-determinism [16, 42, 89] and expects both the primary and backup to have an identical hardware configuration. Non-deterministic events occurring at the primary VM are captured and replayed at the backup, in order to keep their execution synchronized. In VMs with a single CPU, execution replay based HA incurs much lower latency overhead since network outputs can be released as soon as any non-deterministic events are propagated to the backup. However, when a VM has multiple CPUs, the system has to record the order of accesses to shared memory from various processors and replay the same at the backup. Recording and replaying shared memory accesses is a very expensive operation, and has been shown to cause very high overhead when executing multi-threaded applications [5, 37, 131].

Special purpose applications such as Dynamo [34], BigTable [23], Cassandra [136], RAMCloud [73] are built from ground-up with HA in mind, as stated earlier. Their customized software stack and relaxed consistency models lead to very low performance overhead. Most of these systems also have control over their clients, eliminating the need to provide completely transparent failover. The client code is typically designed to automatically reconnect and retry the operation as quickly as possible.

2.2 Towards Unified Primitives for Adaptability

Consider some popular classes of applications that are being deployed in the cloud today [137], such as e-commerce, business analytics, collaborative systems, cus-

tomer relationship management, media servers, content management systems, etc. Recall from Sections 2.1.1 and 2.1.2 that, in each of these applications, both the underlying infrastructure and the runtime state in the application must be managed carefully in order to become adaptive. Applications require the ability to dynamically scale their deployments and rebalance load efficiently. They need to ensure that the failures in the underlying infrastructure do not disrupt the service provided to the user. Given these requirements, we find that existing solutions are too generic, specialized or lack reusable interfaces. Consequently, commodity applications roll out their own solutions for adaptability. Recent events, though, have demonstrated that applications are still struggling to scale [119, 122–124], and are unable to gracefully recover from infrastructure failures [133, 134, 147].

The position of this thesis is that implementing adaptability properties at the application level is both hard and expensive. Given that they are universally desired by all applications, there is a strong incentive to provide adaptability services from the system-level such that they can be readily leveraged by applications. Application layer solutions can vary widely from one domain to another, depending on the application's execution semantics, consistency requirements, and the structure of runtime state. However, their fundamental design principles remain the same. Elasticity is achieved through dynamic partitioning of state and associated inputs across a cluster of application instances [21, 40, 105, 110, 161]. High availability, in the face of fail-stop failures, is achieved through replication of state [58, 77, 118, 159]. This observation leads us to the following question: *Is it possible to abstract these operations into a unified set of system primitives that could be used across different application domains?* With the aim of answering this question, this thesis explores the design space of system support for elasticity and high availability.

Let us begin with the high level operations involved in elasticity and HA, and then zero in on the core facilities that need to be in place to implement these operations. In order to dynamically scale an application, we need to automatically partition its runtime state and inputs. To protect an application from failures, we need to automatically replicate the state inside a partition from one node (primary) in the application cluster to another (backup). On failure of the primary, operations on the partition can be resumed by activating the backup's copy of the partition, followed by rerouting the inputs pertaining to the partition. There are three challenges

to achieving these properties:

- **Identifying partitions.** The first challenge is to identify the partitioning boundary at which the application's state and inputs can be broken into smaller pieces. While the structure of runtime state varies from one application to another, applications in the same domain tend to have a similar representation of state. For example, the runtime state of network middleboxes is a composition of flow-level states which are independent of one another. The state of a web application is a composition of per-client sessions.

- **Moving partitions and inputs.** The second challenge is to enable the partitioned state to become mobile. For elasticity, we need the ability to migrate partitions and inputs across a dynamically scaling cluster, while for HA we need the ability to replicate (continuously migrate) state to other nodes and re-route inputs on failure. In order to maintain consistency of state during the move/replicate operation, access to the partitioned state must be made exclusive, between the application layer and the underlying system providing elasticity and HA. When the application is accessing the partitioned state, the system should not move or replicate the state to another machine. Conversely, when the system is performing a move/replicate operation, the application should not be allowed to access the partitioned state.

- **Triggers.** The final challenge is to identify when the system should adapt (scale out/in or failover) and provide a policy framework that allows users to determine how the system reacts to these reconfiguration events. For elastic scaling, we need to define metrics to evaluate load on an application node, the conditions under which an application node can be considered to be overloaded or under-loaded so that a scale-out or scale-in operation can be orchestrated, respectively. For HA, we need to be able to detect failures quickly and reliably taking into account corner cases such as network partitions.

Since the placement of a partition and its inputs in an application cluster are now controlled by the system, management of the partitions must be delegated to

the underlying system. In order to communicate the partition boundaries to the underlying system, we need a set of APIs that allow a developer to demarcate partition boundaries, and access the state in each partition in a transactional fashion.[3] Each partition has to be a self contained entity with no dependencies on other partitions and the rest of the system. When such dependencies exist, the runtime state has to be re-partitioned such that the resulting partition has no external dependencies. Consequently, the granularity of a partition (for e.g., few sessions per partition or the entire state of a VM) determines the efficiency and performance that can be achieved when performing elasticity and availability related operations.

To tackle the issues related to state management, this thesis borrows key principles from prior work on distributed shared memory (DSM) systems [6,13,20,32,38, 39, 55], such as object migration, virtual memory range partitioning, type-specific coherency models for shared data, etc. This thesis combines these principles with recent advances in networking to simplify the development of adaptive internet applications through system-level support. At the same time, in order to make the underlying system practical to implement and maintain, this thesis makes a key assumption regarding the runtime state: memory references cannot cross machine boundaries. Unlike DSM systems, the system will not attempt to page-in missing data across the network. References to state residing on a remote replica will result in an incorrect output or application crash.

2.3 Starting with Domain-Specific Primitives for Adaptability

How should we go about designing adaptability interfaces that cater to all applications? Rather than starting with the design of a new set of system interfaces, this thesis adopts a bottom-up refactoring style approach to the problem. By developing system interfaces for different applications, we can glean common design ideas that can later be transformed into a set of universally applicable system interfaces for adaptability. Since there is a diverse set of applications that exist in the cloud, it is infeasible to design such interfaces for every application. Fortunately, appli-

[3] In the absence of such APIs, the system can fallback to coarse-grained partitioning (e.g., at VM level).

cations in the same domain tend to be structured in a similar manner. This thesis thus presents case studies describing the design and implementation of system interfaces for adaptability for different types of applications. In Chapter 6, we review the individual interface designs and attempt to derive unified system interfaces.

Which application domains should we investigate? From the challenges listed in Section 2.2, we know that the first step towards designing unified adaptivity interfaces is to identify partitions – the granularity at which the application would be scaled. In the best case each partition would correspond to a single user's session/flow, while in the worst case an entire application node with all its user sessions would be considered as one big partition. We pick two popular application domains that represent each of these cases, namely network middleboxes and database systems representing the best and worst case partitioning scenarios respectively.

Middleboxes tend to achieve a clean separation between a small amount of per-flow network state and a large amount of complex application logic. The runtime state in a middlebox can be partitioned at flow boundaries, such that each flow is an independent, isolated, execution context, with very little state being shared across the system. *Split/Merge* [85] (Chapter 3) takes advantage of this property to provide load balanced elasticity, while *Pico Replication* [84] (Chapter 4) provides customizable high availability.

Database systems contain partitionable session state that depends heavily on the shared state in the application (e.g., locks, buffer pools, persistent data, etc). This shared state cannot be made distributed in a scalable fashion. As a result, the entire application node becomes the minimal unit of state for partitioning. From an interface design perspective it is not possible to provide effective elasticity primitives as the system cannot manage any part of the application's state. On the other hand, system-level primitives for HA can be provided by encapsulating each application node inside a VM and replicating it as one unit. *RemusDB* [69] (Chapter 5) describes a hypervisor-level abstraction that provides HA with minimal performance overhead through database-aware VM replication.

Summary. This chapter looked at the state-of-the-art design techniques for adaptive applications and the existing system-level support for developing such applica-

tions (Section 2.1). We find that they are insufficient for a wide variety of commonplace applications. We raised the possibility of a unified set of primitives across all applications and identified the main challenges in developing such interfaces (Section 2.2). In search of an answer, this thesis starts with two different application domains: databases and middleboxes.

Chapter 3

Transparent Elasticity for Middleboxes

This chapter presents a system-level abstraction that provides load balanced elasticity to network middlebox applications. As virtualization pervades all aspects of the data center, the ability to dynamically scale—known as elasticity—must be exhibited not only by servers, but also network middleboxes that perform crucial tasks such as packet filtering, load balancing and intrusion detection. A recent survey of 57 enterprise networks of various sizes found that scalability was indeed critical for middleboxes [93].

Unlike general applications, the state held by a middlebox application exhibits a unique property: it is mostly composed of a collection of independent sub-states per network flow. To exploit this property for efficient, balanced elasticity, we present a state-centric, systems-level abstraction for elastic middleboxes called *Split/Merge*. A virtual middlebox that has appropriately classified its state (e.g., per-flow state) can be dynamically scaled out (or in) by a Split/Merge system, but remains ignorant of the number of replicas in the system. Per-flow state may be transparently split between many replicas or merged back into one, while the network ensures flows are routed to the correct replica. As a result, Split/Merge enables load-balanced elasticity. We have implemented a Split/Merge system, called *FreeFlow*, and ported Bro [78], an open-source intrusion detection system, to run on it. In controlled experiments, FreeFlow enables a 25% reduction in maximum

latency while eliminating hotspots during scale-out and a 50% quicker scale-in than standard approaches.

3.1 Split/Merge

In this section, we describe the common structure in which middlebox state is organized. Motivated by this common structure, we define the three types of states exposed by the Split/Merge abstraction. We then describe how robust elasticity is achieved by tagging state and transparently partitioning network input across virtual middlebox replicas. We conclude the section with design challenges.

3.1.1 Anatomy of a Virtual Middlebox

A middlebox is defined as "any intermediary device performing functions other than the normal, standard functions of an IP router on the datagram path between a source host and destination host" [19]. Middleboxes can vary drastically in their function, performing such diverse tasks as network address translation, intrusion detection, packet filtering, protocol acceleration, and acting as a network proxy. However, middleboxes typically process packets and share the same basic structure [48, 54, 94, 108].

Figure 3.1 shows the basic structure of a middlebox. State held by a middlebox can be characterized as policy and configuration data or as run-time responses to network flows [48, 96, 108, 109]. The former is provisioned, and can include, for example, firewall rules or intrusion detection rules. The latter, called *flow state* is created on-the-fly when packets of a new flow are received for the first time or through an explicit request. Flow state can vary in size. For example, on seeing a packet from a new flow, a middlebox may generate some small state like a firewall pinhole or a NAT translation entry, or it may begin to maintain a buffer to reconstruct a TCP stream.

Flow state is stored in a *flow table* data structure and accessed using flow identifiers (packet headers) as keys (Figure 3.1). Models of middleboxes have been developed that represent state as a key-value database indexed by addresses (e.g., a standard IP 5-tuple) [54]. A middlebox may have multiple flow tables (e.g., per network interface). It may also contain timers that refer to flow state, for example,

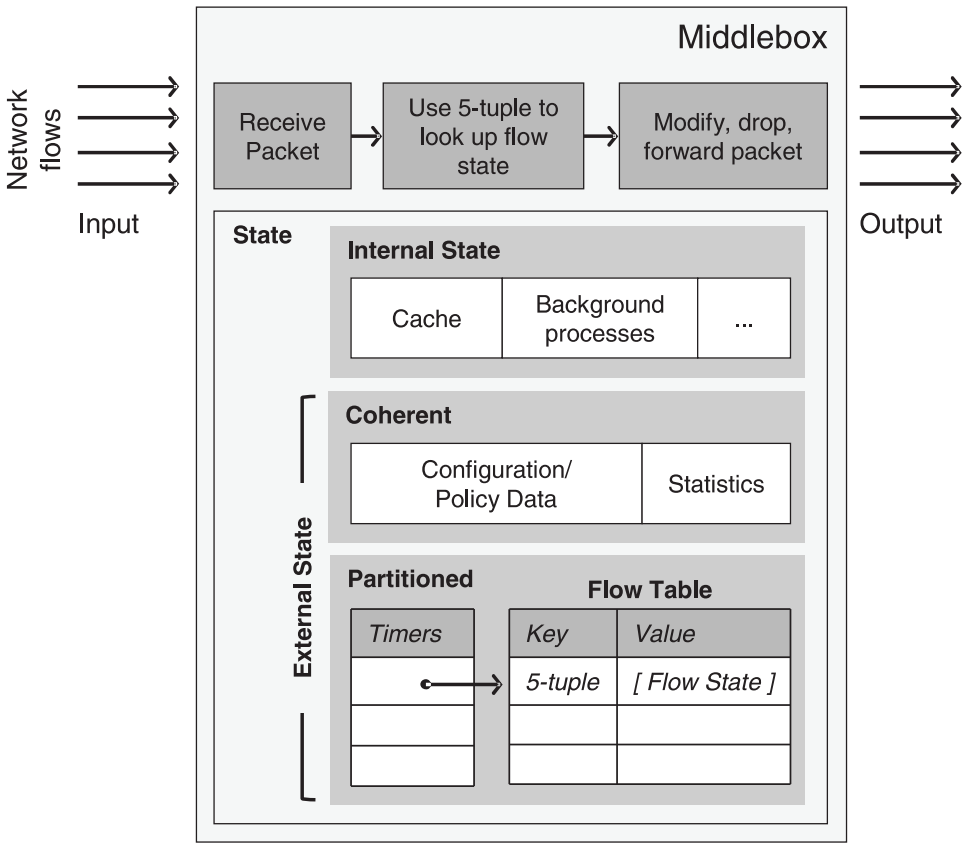


Figure 3.1: Typical structure of a middlebox

to clean up stale flows.

We have performed a detailed analysis of the source code or specifications of several middleboxes to confirm that they fit into this model. We discuss three of them below:

Bro. Bro [78] is a highly stateful intrusion detection system. It maintains a flow table in the form of a dictionary of `Connection` objects, indexed by the standard IP 5-tuple without the protocol field. Inside the `Connection` objects, flow-related state varies depending on the protocol analyzers that are being used. Analyzer objects contain state machine data for a given protocol (e.g., HTTP) and reassembly buffers to reconstruct a request/response payload, leading to tens of

kilobytes per flow in the common case. A dictionary of timers is maintained for each `Connection` object. Bro also contains statistics and configuration settings.

Application Delivery Controller (ADC). An ADC [121, 126, 158] is a packet-modifying load balancer that ensures the addresses of servers behind it are not visible to clients. It contains a flow table that is indexed by the source IP address and port. Flow-specific data includes the internal address of the target server and a timestamp, resulting in only tens of bytes per flow. An ADC also maintains timers for each flow, which it uses to clean up flow entries.

Stateful NAT64. Stateful NAT64 [65] translates IPv6 packets to IPv4 and vice-versa. NAT64 maintains three flow tables, which it calls session tables: for UDP, TCP, and ICMP query sessions, respectively. Session tables are indexed using a 5-tuple. Flow state, called session table entries (STEs), consists of a source and destination IPv6 address and a source and destination IPv4 address, so is therefore tens of bytes in size. Timers, called STE lifetimes, are also maintained.

3.1.2 The Split/Merge Abstraction

The Split/Merge abstraction enables transparent and balanced elasticity for virtual middlebox applications. Using Split/Merge, middlebox applications can continue to be written and configured oblivious to the number of replicas that may be instantiated. Each replica perceives an identical VM abstraction, down to the details of the MAC address on the virtual network interface card.

As depicted Figure 3.2, using Split/Merge, the output of a middlebox application remains *consistent*, regardless of the number of replicas that have been instantiated or destroyed throughout its operation. Slightly more formally:

Definition. *Let a VM be represented by a state machine that accepts input from the network, reads or writes some internal state, and produces output back to the network. A Split/Merge-aware VM is abstractly defined as a set of identical state machine replicas; the aggregate output of which—modulo some reordering—is identical to that of a single machine, despite the partitioning of the input between the replicas. Consistency is achieved by ensuring that each replicated state machine*

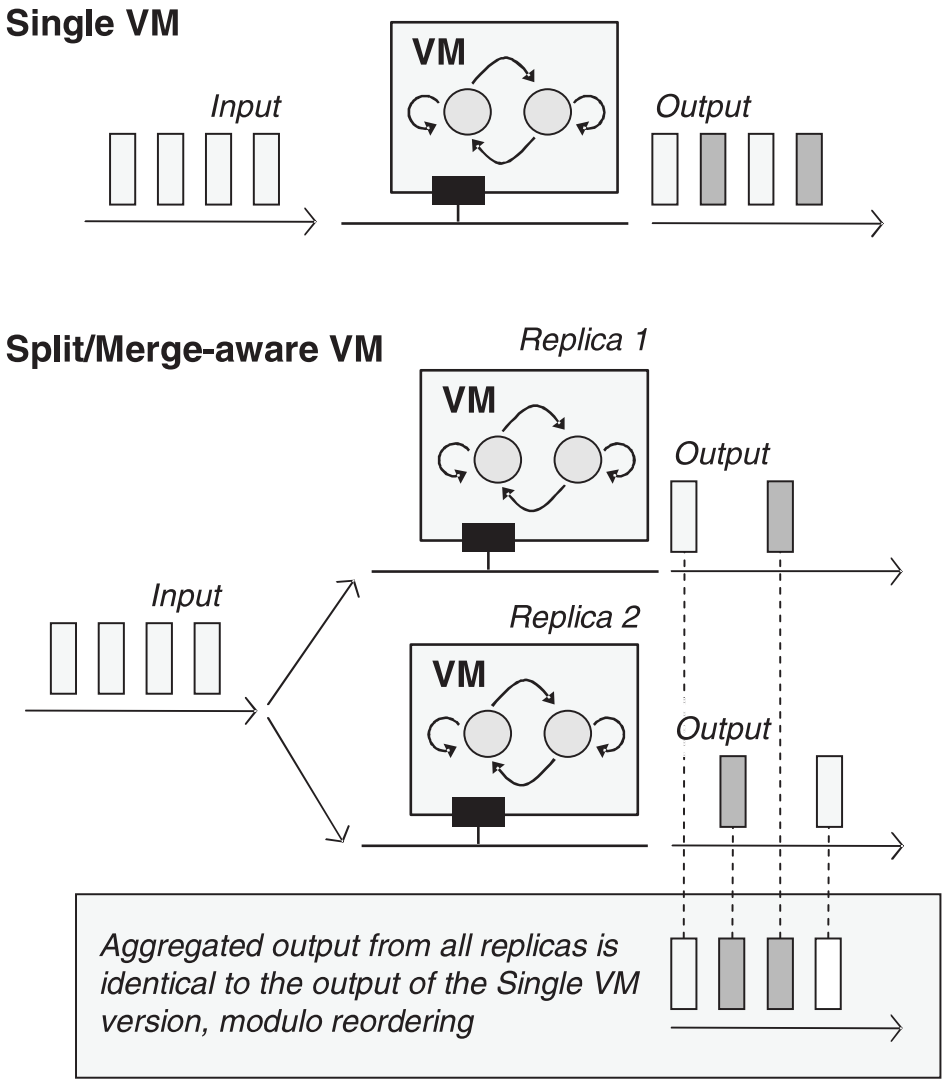


Figure 3.2: Split/Merge retains output consistency irrespective of the number of replicas

can access the state required to produce the appropriate output in response to its share of the input.

There are two types of state in a Split/Merge-aware VM (Figure 3.1): *internal* and *external* state. Internal state is relevant only to a single replica. It can also be thought of as "ephemeral" [22]; its contents can deviate between replicas of the state machine without affecting the consistency of the output. Examples of internal state include background operating system processes, cache contents, and temporary side effects. External state, on the other hand, transcends a single replica. If accessed by *any* replica, external state cannot deviate from what it would have been in a single, non-replicated state machine without affecting output consistency. For example, a NAT may look up the port translation for a particular flow. Any deviation in the value of this state would cause the middlebox to malfunction, violating consistency.

As depicted in Figure 3.1, external state can take two forms: *partitioned* or *coherent*. Partitioned state is made up of a collection of sub-states, each of which are intrinsically tied to a subset of the input and therefore only need to be accessed by the state machine replica that is handling that input. The NAT port translation state is an example of partitioned state, because only the replica handling the network flow in question must access the state. Coherent state, on the other hand, is accessed by multiple state machine replicas, regardless of how the input is partitioned. In Figure 3.1, the flow table and timers reside in partitioned state, while configuration information and statistics reside in coherent state.

3.1.3 Using Split/Merge for Elasticity

Figure 3.3 depicts how the state of a middlebox VM is split and merged when elastically scaling out and in. On scale-out, internal state is replicated with the VM, but begins to diverge as each replica runs independently. Coherent state is also replicated with the VM, but remains consistent (or eventually consistent) because access to coherent state from each replica is transparently coordinated and controlled. Partitioned state is split among the VM replicas, allowing each replica to work in parallel with its own sub-state. At the same time, the input to the VM is partitioned, such that each replica receives only the input pertinent to its partitioned

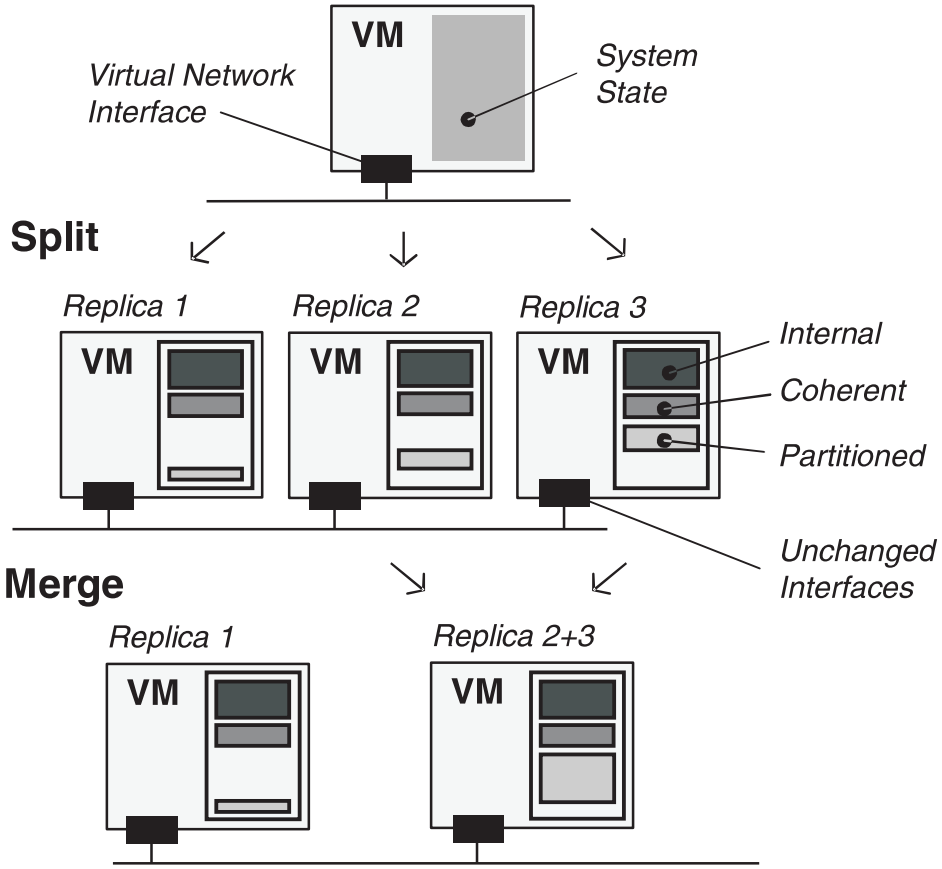


Figure 3.3: A Split/Merge-aware VM uses the different types of state to achieve transparent elasticity.

sub-state.

On scale-in, one of the replicas is selected to be destroyed. Internal state residing at the replica can be safely discarded, since it is not needed for consistent output. Coherent state may be discarded when any outstanding updates are pushed to other replicas. The sub-states of the partitioned state residing at the dying replica are merged into a surviving replica. At the same time, the input that was destined for the dying replica is also redirected to the surviving replica now containing the partitioned sub-state.

3.1.4 Challenges

To implement a system that supports Split/Merge for virtual middleboxes, several challenges need to be met.

C1. **VM state must be classified.** For virtual middlebox applications to take advantage of Split/Merge, each application must identify which parts of its VM state are internal vs. external. Fortunately, the structure of middleboxes (Figure 3.1) is naturally well-suited to this task. The flow table of middleboxes already associates partitioned state with a subset of the input, namely network flows.

C2. **Transactional boundaries must be respected.** In some cases, a middlebox application may need to convey that it finished processing relevant input before partitioned state can be moved from one VM to another. For example, an IDS may continuously record information about a connection's state; such write operations must complete before the state can be moved. Other cases, such as a NAT looking up a port translation, do not have such transactional constraints.

C3. **Partitioned state must be able to move between replicas.** Merging partitioned state from multiple replicas requires at the most primitive level the ability to move the responsibility for a flow from one replica to another. In addition to moving the flow state, the replica receiving the flow must update its flow table data structures and timer structures so that it can readily access the state.

C4. **Traffic must be routed to the correct replica.** As partitioned state—associated with network flows—is split between VM replicas, the network must ensure that the appropriate flows arrive at the replica holding the state associated with those flows. Routing is complicated by the fact that partitioned state may move between replicas and each replica shares the same IP and MAC address.

The Split/Merge abstraction can be thought of in two parts: splitting and merging *VM state* between replicas (Figure 3.3), and splitting and merging *network*

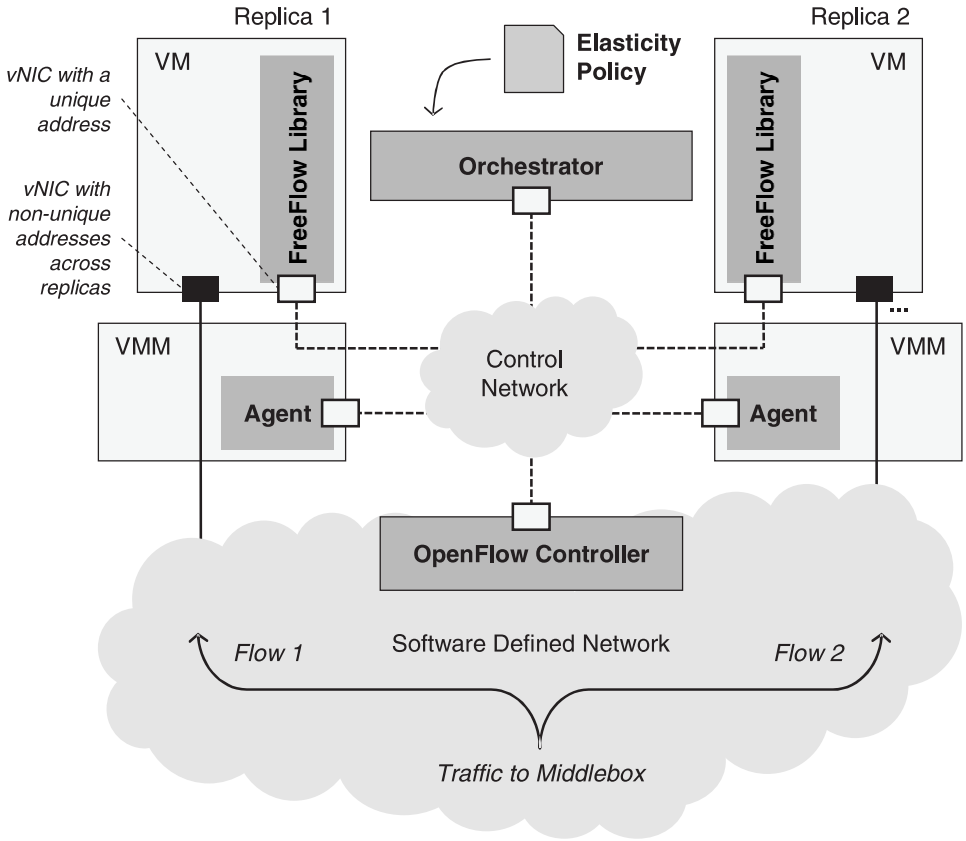


Figure 3.4: FreeFlow architecture

input between replicas (Figure 3.2). As such, the challenges can also be classified into those that deal with state management (C1, C2, C3) and those that deal with network management (C4).

3.2 FreeFlow

FreeFlow implements the Split/Merge abstraction to enable virtual middleboxes to achieve transparent, balanced elasticity. The design of FreeFlow is shown in Figure 3.4. It consists of four components. First, the state aspects of the Split/Merge abstraction are implemented via the application-level *FreeFlow library*, which addresses the state-related challenges (C1, C2,C3). In particular, through the interface

```
// MIDDLEBOX−SPECIFIC PARTITIONED STATE HANDLING

create_flow(flow_key, size);     // alloc flow state
delete_flow(flow_key);           // free flow state

flow_state get_flow(flow_key); // increment refcnt
put_flow(flow_key);              // decrement refcnt

flow_timer(flow_key, timeout, callback);


// COHERENT STATE HANDLING

create_shared(key, size, cb); // if cb is null, then use
delete_shared(key);           // strong consistency

state get_shared(key, flags); // synch | pull | local
put_shared(key, flags);       // synch | push | local
```

Figure 3.5: Interface to the FreeFlow library

to the library, a middlebox application classifies its state as internal or external and communicates its transactional requirements. Additionally, the library manages all aspects of external state, including the migration of partitioned sub-states. Second, the network aspects of the Split/Merge abstraction are implemented in FreeFlow's *Split/Merge-aware software defined network* (SDN). The SDN addresses the final challenge (C4) and ensures that the correct network flows are routed to the replica maintaining the corresponding partitioned sub-state. Third, the *orchestrator* implements an elasticity policy: it decides when to create or destroy VM replicas and when to migrate flows between them. Finally, *VMM agents* perform the actual creation and destruction of replicas. The four components communicate with each other over a control network, distinct from the Split/Merge-aware SDN.

We have implemented a prototype of FreeFlow, including all of its components shown in Figure 3.4. Each physical machine runs Xen [12] as a VMM and Open vSwitch [80] as an OpenFlow-compatible software switch. In all components, flows are identified using the IP 5-tuple.

3.2.1 Guest Library: State Management

Middlebox applications interact with the FreeFlow library in order to classify state as external and identify transaction boundaries on such state. The interface to the library is shown in Figure 3.5. Behind the scenes, the library interfaces with the rest of the FreeFlow system to split and merge partitioned state between replicas and control access to coherent state.

To fulfill the task of identifying external state, the library acts a memory allocator, and is therefore the only mechanism the middlebox application can use to obtain partitioned or coherent sub-state. Partitioned state in middlebox applications generally consists of a flow table and a list of timers related to flow state; therefore, the library manages both. The library provides an interface, `create_flow`, to allocate a new entry in the flow table against a *flow key*, which is usually an IP 5-tuple. A new timer (and its callback) can be allocated against a flow key using `flow_timer`. Coherent sub-state is allocated against a key by invoking `create_shared`, but the key is not necessarily associated with a network flow.

Transaction boundaries are inferred by maintaining reference counts for external sub-states. Using `get_flow` or `get_shared`, the middlebox application accesses external sub-state from the library, at which point a reference counter is incremented. When the application finishes with a transaction on the sub-state, it informs the library with `put_flow` or `put_shared`, which decrements the reference counter. The application must avoid dangling references to partitioned state. If it fails to inform the library that a transaction is complete, the state will be pinned to the current replica.

The library may copy partitioned sub-state across the control network to another replica in response to a notification from the orchestrator (Section 3.2.3). When instructed to migrate a flow—identified with a flow key and a unique network address for a target replica on the control network—the library waits for the reference counter on the state to become zero, then copies the flow table entry and any timers across the control network. The flow table at the source is updated to record the fact that the particular flow state has migrated. Upon future `get_flow` calls, the library returns an error code indicating that the flow has migrated and the packet should be dropped. Similarly, when the target library receives flow data—

and the flow key for it to be associated with—during a flow migration, the flow table and timer list are updated and the orchestrator is notified. At any one time, only one library instance maintains an active copy of the flow data for a particular flow.

The library also manages the consistency of coherent state across replicas. In most cases, strong consistency is not required. For example, the application can read and write counters or statistics locally most of the time (using the `LOCAL` flag on `get_shared`). Periodically, the application may require a consistent view of a counter. For example, an IDS may need to check that an attack threshold value has not been exceeded. For periodic merging of coherent state between replicas, FreeFlow supports *combiners* [33, 66]. On `create_shared`, an application can specify a callback function, which takes a list of coherent state elements as an argument and combines them in an application specific way. In most cases, this function simply adds the values of the counters in the coherent state. The combiner will be invoked automatically by the library when a replica is about to be destroyed. It can also be invoked explicitly by the application either before a reference to the coherent state is obtained (using the `PULL` flag on `get_shared`) or after a transaction is complete (using the `PUSH` flag on `put_shared`). The combiner never runs in the middle of a transaction; `get_shared` using `PULL` may block until other replicas finish their transaction and the state can be safely read. In the rare case that strong consistency is required, the application does not specify a combiner, and library instead interacts with a distributed locking service [18, 53]. On `get_shared` (with the `SYNCH` flag), the library obtains the lock associated with the specified key and ensures that it has the most recent copy of the coherent data. The library releases the lock on `put_shared` and the system registers the local copy of the coherent data as the most recent version.

We have implemented the FreeFlow library as a C library. In doing so, we addressed the implementation challenge of allowing flow state to include self-referential pointers to other parts of the flow state. To support unmodified pointers, the library must ensure that the flow state resides at same virtual address range regardless of which replica it is in. To accomplish this, the library allocates a large virtual address space *before* notifying the VMM agent to compute the initial snapshot. Within the virtual address range, the orchestrator provides each replica with

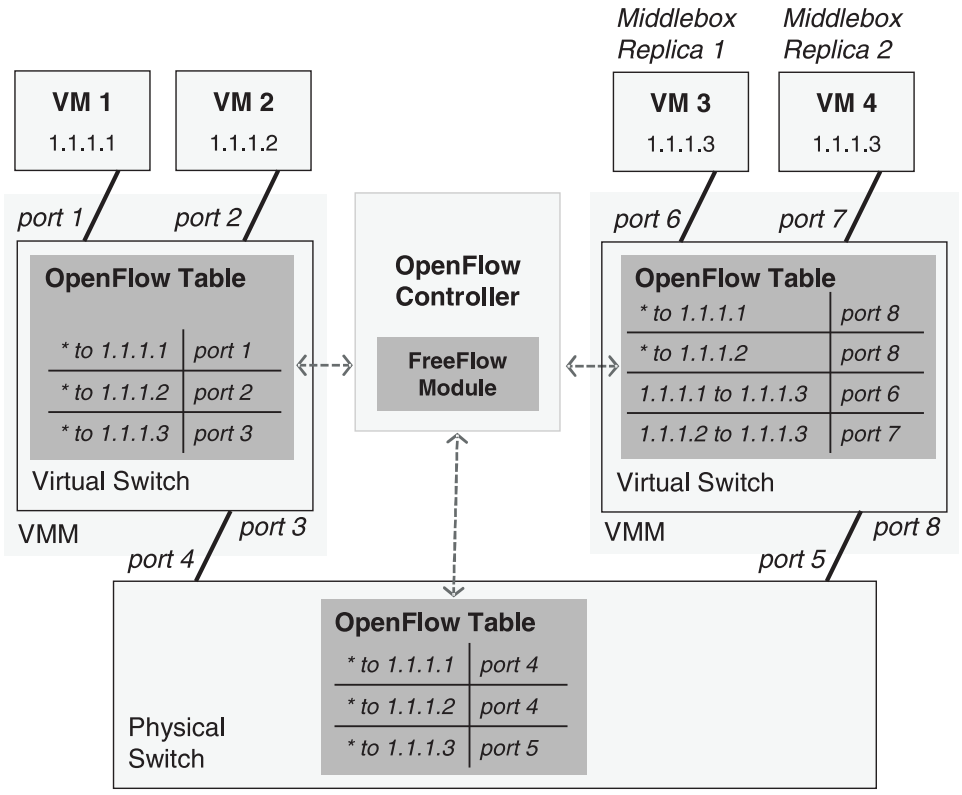


Figure 3.6: The SDN splits network input to replica VMs based on flow rules. The SDN ensures that traffic from VM 1 arrives at VM 3 and traffic from VM 2 arrives at VM 4. For clarity, we have omitted the flow rules for routing middlebox output.

a non-overlapping region to service new flow related memory allocations obtained with `create_flow`.

3.2.2 Split/Merge-Aware SDN: Network Management

The Split/Merge-aware SDN implements the networking part of the Split/Merge abstraction. Each replica VM contains an identical virtual network interface. In particular, every replica has the same MAC and IP address. Maintaining consistent addresses in the replicas avoids breaking OS or application level address dependencies in the internal state within a VM.

As depicted in Figure 3.6, FreeFlow leverages OpenFlow-enabled [165] network elements (e.g., switches [80] and routers) to enforce routing to various replicas. As packets flow through the OpenFlow network, each network element searches a local forwarding table for rules that match the headers of the packet, indicating they belong to a particular flow. If an entry is found, the network element forwards the packets along the appropriate interface on the fast path. If no entry exists, the packet (or just its header) is forwarded to an *OpenFlow controller*. The OpenFlow controller has a global view of the network and can make a routing decision for the new flow. The controller then pushes a new rule to one or more network elements so that future packets belonging to the flow can be forwarded without consulting the controller.

The Split/Merge-aware SDN must ensure that packets arrive at the appropriate replica even as partitioned flow state migrates between replicas. To do this, FreeFlow contains a customized OpenFlow controller that communicates with the orchestrator (Section 3.2.3). When a flow is migrated between replicas, the orchestrator interfaces with the OpenFlow controller to communicate the new forwarding rules for the flow. Packets belonging to new flows are forwarded to the OpenFlow controller by default. The OpenFlow controller picks a replica toward which the new flow should be routed and notifies the orchestrator.

When a flow migration notification is received from the orchestrator, rules to route the flow are deleted from all network elements in the current path traversed by the flow. The flow is then considered *suspended*. Packets arriving from the switches are temporarily buffered at the OpenFlow controller until the flow is *resumed* by the controller at the new replica. The flow is not resumed until partitioned sub-state has arrived at its new destination. The controller resumes a flow by calculating a new path for the flow that traverses the new replica, installing forwarding rules in the switches on the path, and injecting any buffered packets directly into the virtual switch connected to the new replica.[1]

We implemented the SDN in a module on top of POX [169], a python version of the popular NOX [49] OpenFlow controller. The controller provides a simple web API that allows it to receive notifications from the orchestrator about events

[1] Alternately, buffering could occur at the destination hypervisor and the controller could update the path immediately upon suspend, thereby reducing its load.

like middlebox creation and deletion, or instructions to migrate one or more flows from one replica to another. We addressed three implementation challenges. First, the controller cannot use MAC learning techniques for middleboxes because every replica shares a MAC address. Instead, when replicas are created, the VMM agent registers a replica interface on a virtual switch port with the controller. Second, ARP broadcast requests may cause multiple replicas to respond or other unexpected behavior, since all the replicas share a MAC address. To avoid this, the controller intercepts and replies to ARP requests that refer to the middlebox IP. Finally, the controller must ensure that bi-directional flows are assigned to the same replica. This is achieved by maintaining a table that maps each flow to its replica. Before assigning a new flow, this table is checked to see if a mapping exists for the flow in the reverse direction and if so, the respective replica is chosen to handle the new flow.

3.2.3 Orchestrator: Splitting and Merging

The orchestrator implements the most fundamental primitive for enabling the splitting and merging of partitioned state between replicas: flow migration. Figure 3.7 shows the migration of a flow $\langle b \rangle$ between two replicas. The orchestrator interacts with other parts of the system as follows. It:

- instructs the SDN to suspend the flow $\langle b \rangle$ such that no traffic of the flow will reach either replica.
- instructs the guest library in Replica 2 to transfer the partitioned state associated with $\langle b \rangle$ to Replica 1.
- instructs the SDN to resume the flow by modifying the routing of flow $\langle b \rangle$ such that any new traffic belonging to the flow will arrive at Replica 1.

It is possible, although rare in practice, that some packets will arrive at Replica 2 after the flow state has been migrated to Replica 1. For example, packets may be buffered in the networking stack in the kernel of Replica 2 and not yet have reached the application. In case the application receives a packet after the flow state is migrated, it should drop the packet.[2]

[2] In this case, the library returns an error code when flow-specific state is accessed (Section 3.2.1).

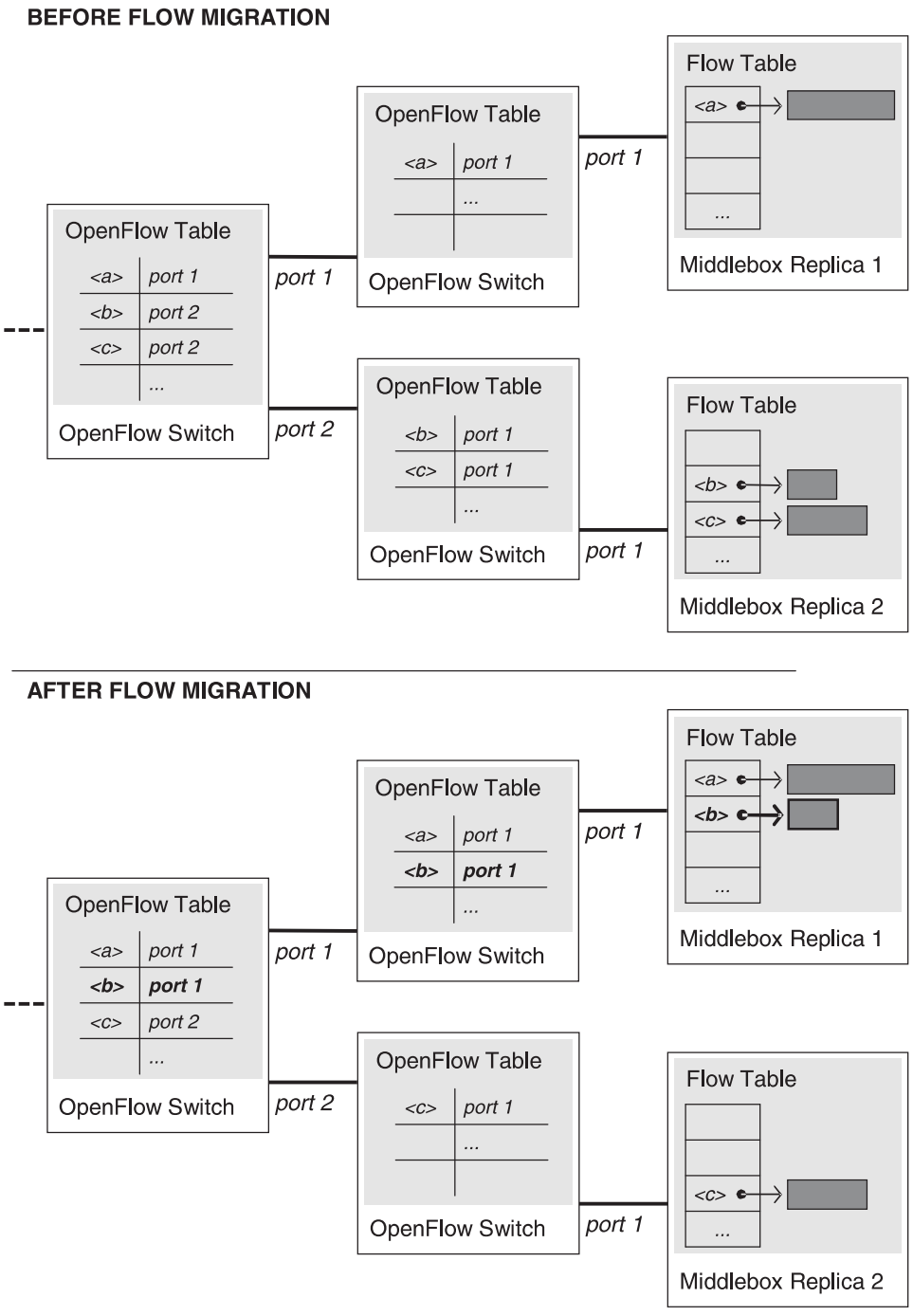


Figure 3.7: Migrating flow $\langle b \rangle$ from Replica 2 to Replica 1

The orchestrator triggers the creation or destruction of replicas by the VMM Agent (Section 3.2.4) in order to scale in or out as part of an elasticity policy. It can trigger these operations automatically based on utilization (resembling the Auto Scaling functionality in Amazon EC2 [148]) or explicitly in response to user input.

3.2.4 VMM Agent: Scaling In and Out

The VMM agent creates or destroys replica VMs in response to instructions from the orchestrator. Replica VMs are instantiated from a point-in-time snapshot of the first instantiation of the middlebox VM before it began processing packets. During library initialization, after variables in the internal state are initialized but before the VM has allocated any external state, the FreeFlow library instructs the VMM agent to compute the snapshot. By definition, the internal state in the replica VM can diverge from the snapshot.

We have implemented the VMM agent in Xen's control domain (Domain 0). Communication with the library is implemented using Xenstore, a standard, non-network based virtual communication channel used between guest VMs and Domain 0 in Xen. Checkpoints are computed with the `xm save` command, and replicas are instantiated with the `xm restore` command.[3] The time to create a new fully operational replica is on the order of a few seconds; it may be possible to reduce this delay with rapid VM cloning techniques [60].

3.2.5 Limitations

Virtual middleboxes cannot use FreeFlow (or the Split/Merge paradigm in general) unless their structure roughly matches that described in Figure 3.1. While most middleboxes we have examined do fit this architecture, it should be noted that some middleboxes are more difficult to adapt to FreeFlow than others. The main cause of difficulty is how the middleboxes deal with partitioning granularity and coherent state.

Middleboxes can be composed of numerous layers and modules, each of which may refer to flows using a different granularity. For example, in an IDS, like Bro,

[3] In our prototype, the distribution of VM disk images to physical hosts is performed manually.

one module may store coarse-grained state (e.g., concerning all traffic in an IP subnet), while another may store fine-grained state (e.g., individual connection state). There are two approaches to adapting such a middlebox to FreeFlow. First, the notion of a flow could be expanded to the largest granularity of all modules. In the preceding example, this would mean using the same flow key for all data related to all flows in an IP subnet, fundamentally limiting FreeFlow's ability to balance load. Second, a fine-grained flow key could be used to identify partitioned state, causing the coarse-grained state to be classified as coherent. If strong consistency is required for the coarse-grained state or a combiner cannot be specified, this approach may cause high overhead due to state synchronization.

3.3 Experience Building Split/Merge Capable Middleboxes

To validate that the Split/Merge abstraction is well suited to virtual middleboxes, we have ported Bro, an open-source intrusion detection system, to run on FreeFlow. To evaluate a wider range of middleboxes, we have also implemented two synthetic FreeFlow middleboxes.

3.3.1 Bro

Bro is composed of two key components: an Event Engine and a Policy Script Interpreter. Packets captured from the network are processed by the Event Engine. The Event Engine runs a protocol analysis, then generates one or more predefined events (e.g., connection establishment, HTTP request) as input to the Policy Script Interpreter. The Policy Script Interpreter executes code written in the Bro scripting language to handle events. As explained in Section 3.1.1, the Event Engine maintains a flow table with each table entry corresponding to an individual connection. Each event handler executed by the Policy Script Interpreter also maintains state that is related to one or more flows.

Our porting effort focused on Bro's Event Engine and one event handler.[4] The event handler scans for potential SQL injection strings in HTTP requests to a web-

[4]For ease of implementation, we ported the event handler to C++ instead of using the Bro scripting language.

server. The handler tracks—on a per-flow basis—the number of HTTP requests that contain a SQL injection exploit. When this number exceeds a predefined threshold Bro issues an alert.

Porting Bro to FreeFlow. Porting Bro to FreeFlow involved the straightforward classification of external state and interfacing with the FreeFlow library to manage it. First, we identified all points of memory allocation in the code. If the memory allocation was for flow-specific data, we modified the allocation to use FreeFlow-provided memory instead of the heap. In certain cases, we had to provide custom implementations of standard C++ constructs like `std::List`, to avoid leaking references to FreeFlow-managed memory.

After ensuring partitioned state was allocated in FreeFlow-managed memory, we checked for external references to it. The only two references were from the global dictionary of `Connection` objects and the global dictionary of timers. Since FreeFlow manages access to flow-related objects and timers, we could replace these two global collections. We found that Bro always accesses flow-related state in the context of processing a single packet, and therefore has well-defined transactional boundaries. References from FreeFlow-managed classes to external memory occur only to read static configuration data (internal state).

As expected, there was very little data that we classified as coherent state. We used FreeFlow's support for combiners for non-critical global statistics counters. The combiners were configured to only be invoked by the system (i.e., on replica VM destruction). We did not find any variables that required strong consistency or real-time synchronization across replicas.

Verification. To validate the correctness of the modified system, we used a setup consisting of a client and a webserver, separated by two middlebox replicas running the modified version of Bro. At a high level, we used the client to issue a single flow of HTTP requests containing SQL injection exploits while FreeFlow migrated the flow between the two replicas multiple times. We check for the integrity of state and execution by ensuring (a) Bro generates an alert, (b) the number of exploits detected exactly matches those sent by the client (c) both replicas remain operational after each flow migration. Assuming Bro sees all packets on the flow, the first two conditions cannot be satisfied if the state becomes corrupted during migration. Ad-

ditionally, the system would crash on flow migration when objects inside FreeFlow memory refer to external memory that does not exist on the local replica.

3.3.2 Synthetic Middlebox Applications

We built two synthetic FreeFlow-based middlebox applications that capture the essence of commonly used real world middlebox applications. The first application is compute bound. It performs a set of computations on each packet of a flow, resembling the compute intensive behavior of middlebox applications like an Intrusion Prevention System (IPS) or WAN optimizer. The second application modifies packets in a flow in both directions, using a particular application-level (layer 7) protocol, resembling a NAT or Application Layer Gateway. Both middleboxes were built in user space using the Linux netfilter [163] framework to interpose on packets arriving at the VM. The user space applications inspect and/or modify packets before forwarding them to the target.

3.4 Evaluation

FreeFlow enables balanced elasticity by leveraging the Split/Merge abstraction to distribute—and migrate—flows between replicas. In this section, we evaluate FreeFlow with the following goals:

- demonstrate FreeFlow's ability to provide dynamic and stateful elasticity to complex real world middleboxes (Section 3.4.1),
- demonstrate FreeFlow's ability to alleviate hotspots created by a highly skewed load distribution across replicas (Section 3.4.2),
- measure the gain in resource utilization when scaling in a deployment using FreeFlow (Section 3.4.3), and
- quantify the performance overhead of migrating a single flow under different application loads (Section 3.4.4).

In our experimental setup, a set of client and server VMs are placed on different subnets. Traffic—TCP or UDP—is routed between the VMs via a middlebox.

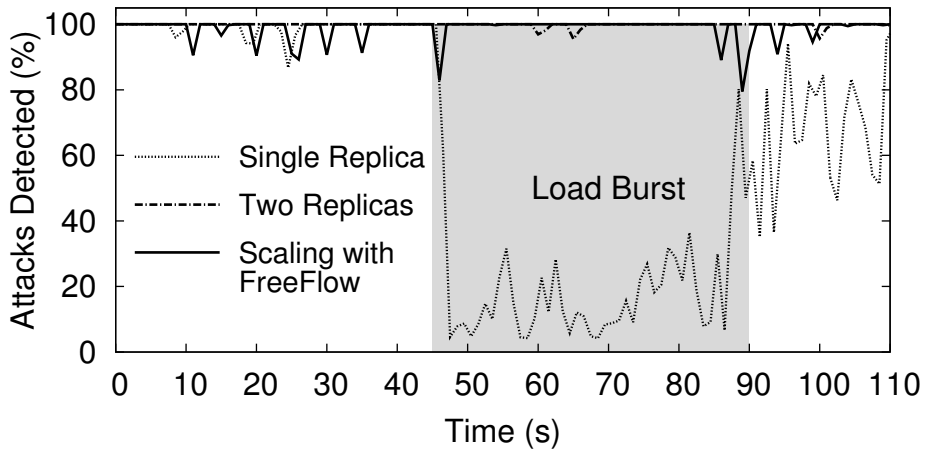


Figure 3.8: Splitting/Merging Bro for stateful elasticity

We evaluate FreeFlow using Bro or one of the synthetic middleboxes described in Section 3.3.2.

3.4.1 Stateful Elasticity with Split/Merge

Figure 3.8 shows FreeFlow's ability to dynamically scale Bro out and in during a load burst, splitting and merging partitioned state. In this experiment, the generated load contains SQL injection exploits; we measure the percentage of attacks detected by Bro to determine Bro's ability to scale to handle the load burst.

Load is generated by a configurable number of cURL-based [151] HTTP clients in the form of a continuous sequence of POST requests to a webserver. The requests contain SQL injection exploits; an attack comprises 31 consecutive requests. Each client is configured to generate 50 requests/second. Throughout the experiment (for 120 seconds), 30 clients generate a base load. We inject a load burst 45 seconds into the experiment by introducing an additional 30 clients and 10 UDP flows (1 Mbps each) that do not contain attacks. The load burst lasts 45 seconds, after which the additional client and UDP traffic ceases.

We compare three scenarios: a single Bro instance that handles the entire load burst, a pair of Bro replicas that share load (flows are assigned to replicas in a

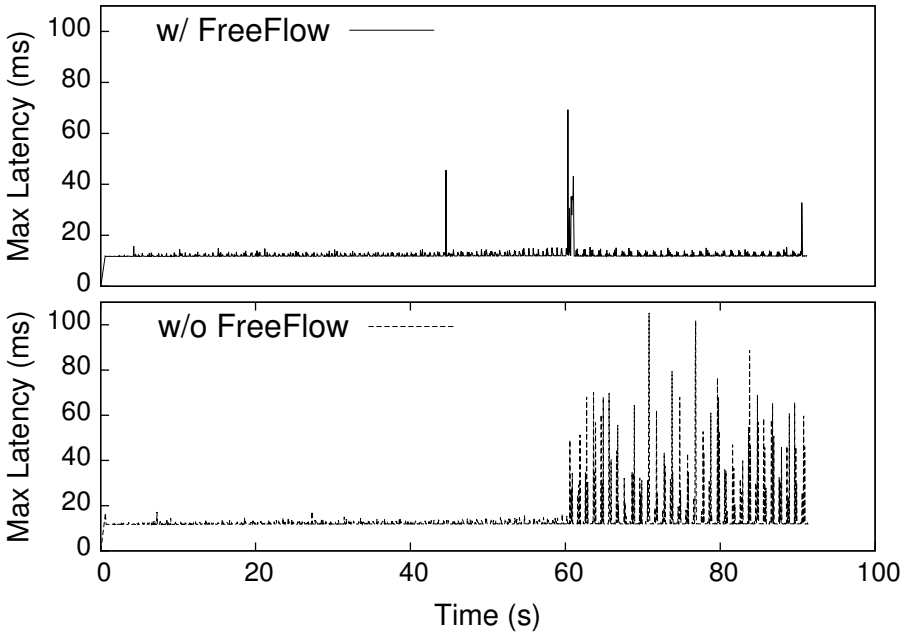


Figure 3.9: Eliminating hotspots with FreeFlow

round-robin fashion), and Bro running with FreeFlow. The FreeFlow scenario begins with a single replica and FreeFlow is configured to create a new replica and split flows and state between them when the number of flows handled by the replica exceeds 60. Similarly, it is configured to merge flows and state and destroy a replica when the number of flows handled by a replica drops below 40.

As shown in Figure 3.8, until the load burst at $t = 45$s, all three configurations have a 100% detection rate. During the load burst, the performance of the single replica reduces drastically because packets are dropped and attacks are missed. The two-replica cluster does not experience any degradation as it has enough capacity and the load is well balanced between the two replicas.

The FreeFlow version of Bro behaves in the same manner as a single replica, until the load burst is detected around $t = 45$s. While partitioned state is being split to a new replica, packets are dropped and attacks are missed. However, the detection rate quickly rises because the two replicas have enough capacity for the load burst. After the load burst ($t = 85$s), FreeFlow detects a drop in load, so

merges partitioned state and destroys one of the replicas. The FreeFlow version of Bro continues to detect attacks at the base load with a single replica. FreeFlow therefore enables Bro to handle the load burst without wasting resources by running two replicas throughout the entire experiment.

3.4.2 Hotspot Elimination

In this experiment, we demonstrate FreeFlow's ability to eliminate hotspots that arise when the load distribution across middleboxes becomes skewed. For the purpose of this discussion, we define a hotspot as the degradation in network performance due to high CPU or network bandwidth utilization at the middlebox.

We use the compute-bound middlebox application described in Section 3.3.2 under load from 1 Mbps UDP flows. We define our scale-out policy to create a new replica once the number of flows in a replica reaches 100 (totaling 100 Mbps per replica). Flows are gradually added to the system every 500 ms up to a total of 101 flows. After scaling out, the system has two replicas: one with 100 flows and another with just one flow.

As expected, the replica handling 100 flows experiences much higher load than the other replica. The resulting hotspot is reflected by highly erratic packet latencies experienced by the clients, shown in Figure 3.9 and Figure 3.10. Figure 3.9 shows the maximum latency, while Figure 3.10 shows the fluctuations in the average latency during the last 40s of the experiment. FreeFlow splits the flows evenly among the two replicas thereby re-distributing the load and alleviating the hotspot. Ultimately, FreeFlow achieves a 26% reduction in the average maximum latency during the hotspot, with a 73% lower standard deviation.

Irrespective of flow duration and traffic patterns, without FreeFlow's ability to balance flows, an over-conservative scale-out policy may be used to ensure hotspots do not occur, leading to low utilization and wasted resources. By balancing flows, FreeFlow enables less conservative scale-out policies leading to higher overall utilization.

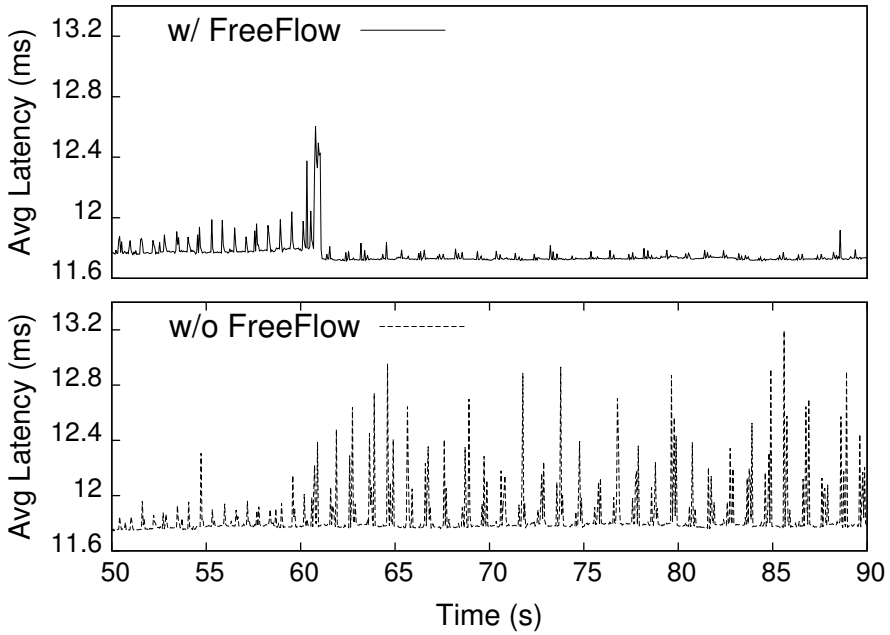


Figure 3.10: Performance impact of FreeFlow's load rebalancing on hotspots

3.4.3 Efficient Consolidation

In this experiment, we show how FreeFlow's ability to statefully merge flows between two or more replicas can be used to consolidate resources during low load and improve overall system utilization. We measure how quickly FreeFlow can scale in compared to a standard *kill-based* technique, in which a replica is killed only when all its flows have expired. We also measure the average system utilization per live replica during scale in, shown in Figure 3.11.

We start with 4 replicas running the compute-bound middlebox application (Section 3.3.2), handling 50 UDP flows of 1 Mbps each. The experiment is configured such that one flow expires every 500 ms since the beginning of the measurement period.

In the best case scenario, the first 50 flows expire from the first replica in the first 25 seconds, enabling the kill-based technique to destroy the replica. The second 50 flows expire from the second replica in the next 25 seconds, enabling the second replica to be destroyed, and so on. In this case, the average system uti-

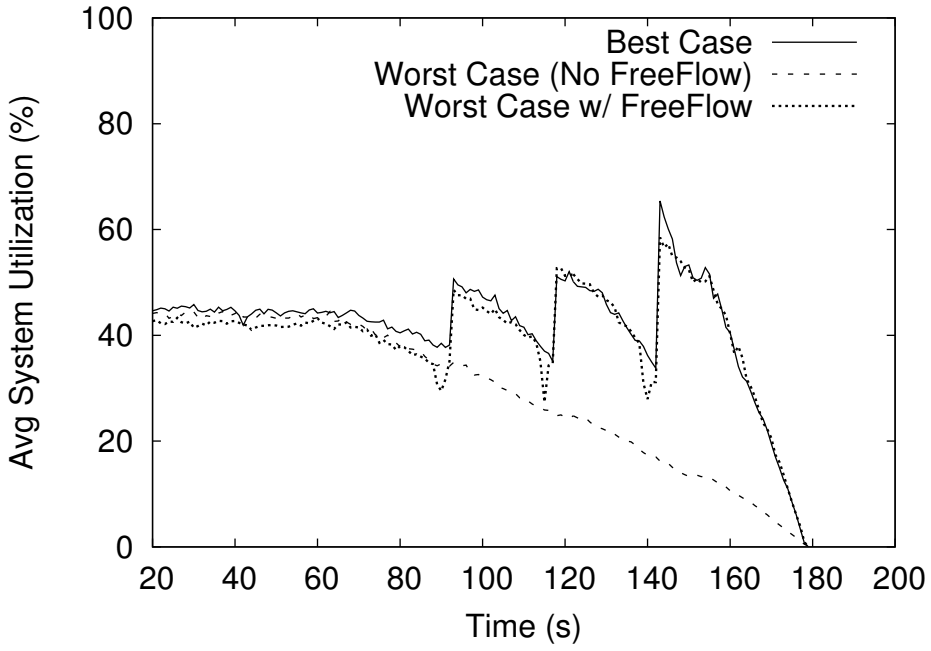


Figure 3.11: Scaling in with FreeFlow

lization remains high throughout the scale-in process, with a sawtooth pattern as shown in Figure 3.11.

In the worst case scenario, flows expire from replicas in a round-robin fashion. When kill-based scale-in strategy is employed all replicas have to be kept operational until the very end of the experiment since each of the 4 replicas contains one or more active flows. This causes the average system utilization to steadily degrading over the duration of the experiment.

On the other hand, even in the worst case, FreeFlow can destroy a replica every 25 seconds. To accomplish this, FreeFlow is configured with a scale-in policy that triggers once the average number of flows per replica falls below 50. When scaling in, FreeFlow kills a replica after merging its state and flows with the remaining replicas. Subsequently, in the worst case, FreeFlow maintains average system utilization close to that of the kill-based strategy in the best case scenario and improves the average system utilization by up to 43% in the worst case scenario. Based on the the time at which the first replica was killed in the worst case

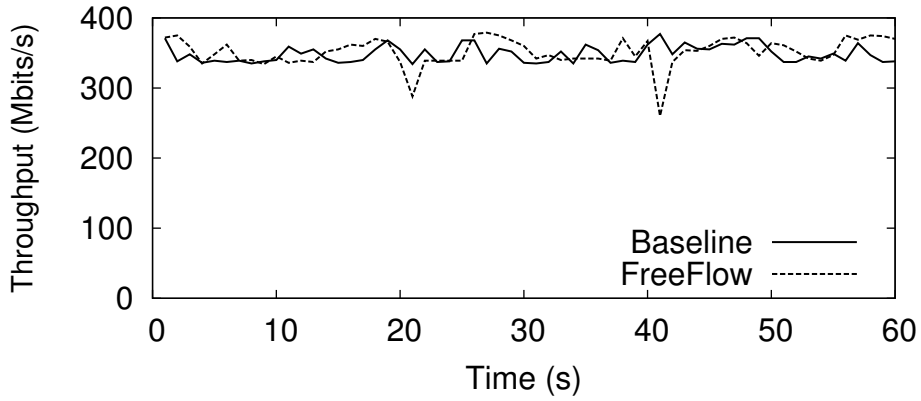


Figure 3.12: Impact of flow migration on TCP throughput (migration at 20s & 40s)

scenario, FreeFlow can scale in 50% faster than the standard kill-based system.

FreeFlow does impact the performance of flows during the experiment; in particular, packet drops are caused by flow migrations that happen when a replica is merged. However, performance impact is low: the average packet drop rate per-flow was 0.9%.

3.4.4 Migrating Application Flow State

Flow state migration is a fundamental unit of operation in FreeFlow, when splitting or merging partitioned state between replicas. Figure 3.12 shows the impact on TCP throughput during flow migration compared to a baseline where no migration is performed. We use the Iperf [156] benchmark to drive traffic on a single TCP stream between the client and the server, through the compute-bound middlebox. We perform two flow migrations: one at the 20th and another at the 40th second, respectively. When sampled at 1 second intervals, we observe a 14 – 31% drop in throughput during the migration period, lasting for a maximum of 1 second.[5]

We further study the overhead of flow migration on a single UDP flow using

[5]Due to Iperf's limitation on the minimum reporting interval, (1 second), we are unable to calculate the exact duration of the performance impact.

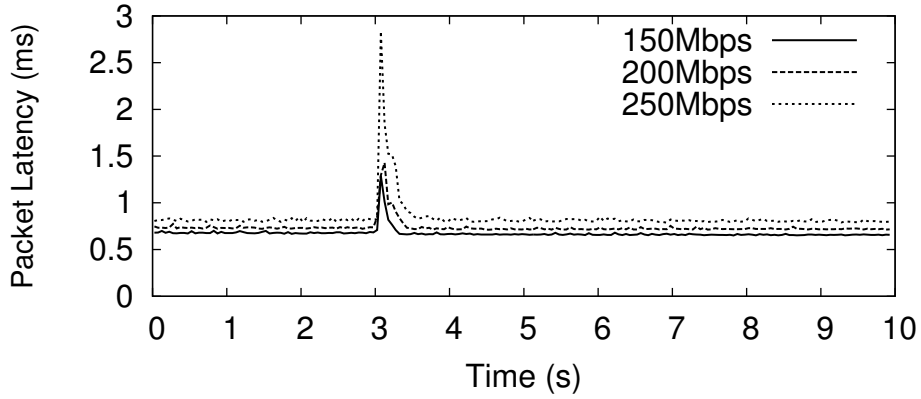


Figure 3.13: Latency overhead of flow migration

the packet modifier middlebox application (Section 3.3.2). For these experiments, the flows are 10 seconds in duration and the migration is initiated three seconds after the start of the flow. The impact of a single flow migration on end-to-end latency for different flow rates is shown in Figure 3.13. We observe a maximum of 1 ms increase in latency during flow migration. The latency fluctuations last for a very short period of time (500 ms). Figure 3.14 shows the overall packet drop rate for the entire duration of the flow. The overall packet drop rate is less than 1% including any disruption caused by the migration. Figure 3.15 shows the impact on throughput as observed by the client when the flow migration occurs. The plotted throughput is based on a 50 ms moving window. As the load on the network increases, there is an increase in throughput loss due to flow migration. However, the drop in throughput occurs only for a brief period of time and quickly ramps up to pre-migration levels.

3.5 Related Work

Split/Merge relies on the ability to identify per-flow state in middleboxes. The behavior and structure of middleboxes has been characterized through the use of models [54]. In other work, state in middleboxes has been identified as global,

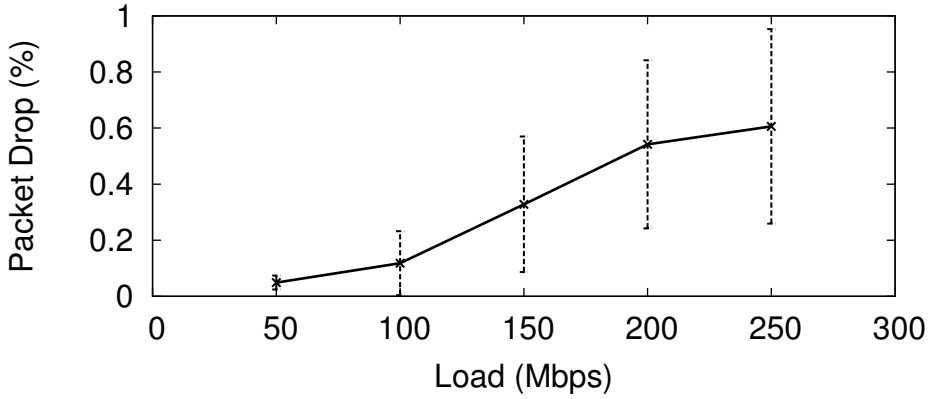


Figure 3.14: Packet drop rate during flow migration

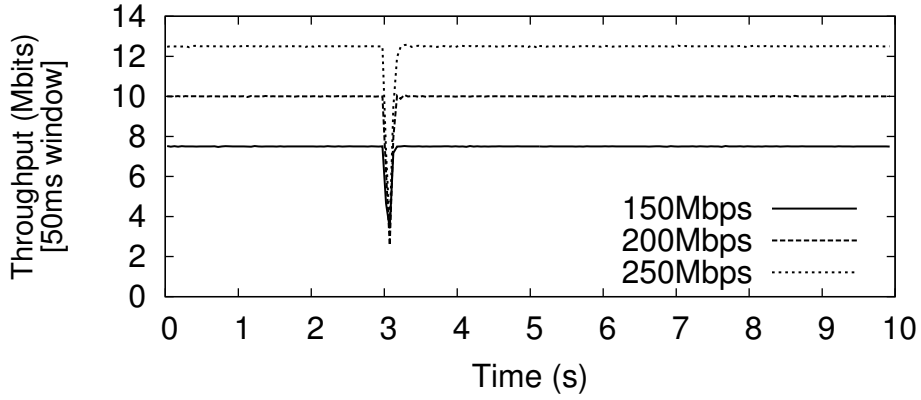


Figure 3.15: Throughput overhead of flow migration (50 ms window)

flow-specific, or ephemeral (per-packet) [94, 108]. On a single machine granularity, Oboe [94], MLP [109], HILTI [96], and multi-threaded Snort [88, 91] all exploit the fact that flow-related processing rarely needs access to data for other flows or synchronization with them. CoMb [92] exploits middlebox structure to consolidate heterogeneous middlebox applications onto commodity hardware, but

does not address the issue of scaling, parallelism, or elasticity.

Clustering techniques have traditionally been used to scale-out middleboxes. The NIDS Cluster [105] is a clustered version of Bro [78] that is capable of performing coordinated analysis of traffic at large scale. By exposing policy-layer state and events as serializable state [97], individual nodes are able to obtain a global view of the system state. The NIDS Cluster cannot scale dynamically and statefully, as it lacks the ability to migrate lower-layer (event-engine) flow state and their associated network flows across replicas.

FreeFlow leverages OpenFlow in its Split/Merge-aware SDN. Load balancing has been implemented in an SDN using OpenFlow with FlowScale [95], and wildcard rules can accomplish load balancing in the network while reducing load on the controller [114]. The Flowstream architecture [47] includes modules—for example, VMs—that handle flows and can be migrated, relying on OpenFlow to redirect network traffic appropriately. However, Flowstream does not characterize external state within an application. Olteanu and Raiciu [72] similarly attempt to migrate per-flow state between VM replicas without application modifications.

There are many ways in which different types of applications are dynamically scaled in the cloud [107]. Knauth and Fetzer [57] describe scaling up general applications using live VM migration [25] and over subscription. Amazon's Auto Scaling [148] automatically creates or destroys VMs when user-defined thresholds are exceeded. SnowFlock [60] provides sub-second scale-out using a *VM fork* abstraction. In either case, applications are typically stateless or written to explicitly communicate local state to their parent before the destruction of any replicas. These approaches do not enable balancing of existing load between instances, potentially resulting in load imbalance [115].

3.6 Summary

This chapter described a new abstraction, Split/Merge (Section 3.1), and a system, FreeFlow (Section 3.2), that enables transparent, balanced elasticity for stateful virtual middleboxes. Using FreeFlow, middleboxes classify their state into ephemeral, flow related (partitioned) and global. Flow state can be split among replicas or merged together into a single replica. At the same time, FreeFlow partitions the

network input source to ensure packets are routed to the appropriate replica. Experiences developing new applications and porting existing applications to FreeFlow were presented in Section 3.3 and validated experimentally in Section 3.4. As networks become increasingly virtualized, FreeFlow addresses a need for elasticity in middleboxes, without introducing the configuration complexity of running a cluster of independent middleboxes. Further, as virtual servers become increasingly mobile, utilizing live VM migration across or even between data centers, the ability to migrate flows—or split and merge them between replicas—will become even more important.

Chapter 4

Transparent High Availability for Middleboxes

This chapter presents a system for providing transparent high availability (HA) to network middleboxes running on the FreeFlow system described in Chapter 3. Middleboxes [120, 125–130] pervade data center networks of various scales. High availability is crucial for middleboxes, as evidenced by recent outages related to middlebox services [134, 135, 141]. Existing high availability (HA) solutions for network middleboxes are application-specific and complex to manage at dynamic scale [93]. In this section, we explore system support for HA in middleboxes. A naive approach is to directly apply HA solutions for virtual machines (VMs), that can protect arbitrary workloads transparently. However, such approaches are heavyweight because (1) the entire VM must be suspended to ensure a consistent checkpoint, (2) all flows—including delay-sensitive flows—are delayed for every checkpoint, and (3) all flows are replicated to the same target, limiting the scope of possible recovery policies.

We propose *Pico Replication (PR)*, a system-level framework for middleboxes that exploits their flow-centric structure to achieve low overhead, fully customizable HA. Unlike generic (virtual machine level) techniques, PR operates at the flow level. Individual flows can be checkpointed at very high frequencies while the middlebox continues to process other flows. Furthermore, each flow can have its own checkpoint frequency, output buffer and target for backup, enabling rich and

Class	State Access Pattern	Examples	HA
Flow Independent	N/A	Stateless Firewalls	No Sync
Flow Dependent	write once, then read	NAT, Vyatta [129, 130]	Sync Once
	read/write for every packet	IPS [171], IDS [78], ADC [120, 125–128]	Sync Continuously

Table 4.1: Taxonomy of Middleboxes for the purposes of HA

diverse policies that balance—per-flow—performance and utilization. PR leverages OpenFlow to provide near instant flow-level failure recovery, by dynamically rerouting a flow's packets to its replication target. We have implemented PR and a flow-based HA policy. In controlled experiments, PR sustains checkpoint frequencies of 1000Hz, an order of magnitude improvement over current VM replication solutions. As a result, PR drastically reduces the overhead on end-to-end latency from 280% to 15.5% and throughput overhead from 99.5% to 3.2%.

4.1 Middlebox HA

In cloud environments, design for failure is critical, including middlebox failure. With short-lived flows that last for a few seconds, such as between two tiers of a N-tier web application, the impact of failure may or may not be pronounced. On the other hand, With even moderately long-lived flows (on the order of few minutes), commonly found on e-commerce websites, session-oriented web applications [62], etc., losing middlebox state results in widespread outages. Previous work [4] has shown that middleboxes can indeed reduce end-to-end availability to 99.9%, compared to the five 9s that today's users typically expect. Apart from disrupting connectivity, loss of a middlebox increases resource usage at the server, as hung TCP connections linger for 50s or more, until a TCP timeout [3]. In this section, we look into which categories of middleboxes can benefit from HA solutions, the requirements for providing effective HA for middleboxes, and the current state of the art.

4.1.1 Middlebox Classification

Middleboxes cover a broad spectrum of applications. Some act as gate keepers (e.g., firewalls), inspecting packet headers and making decisions independent of

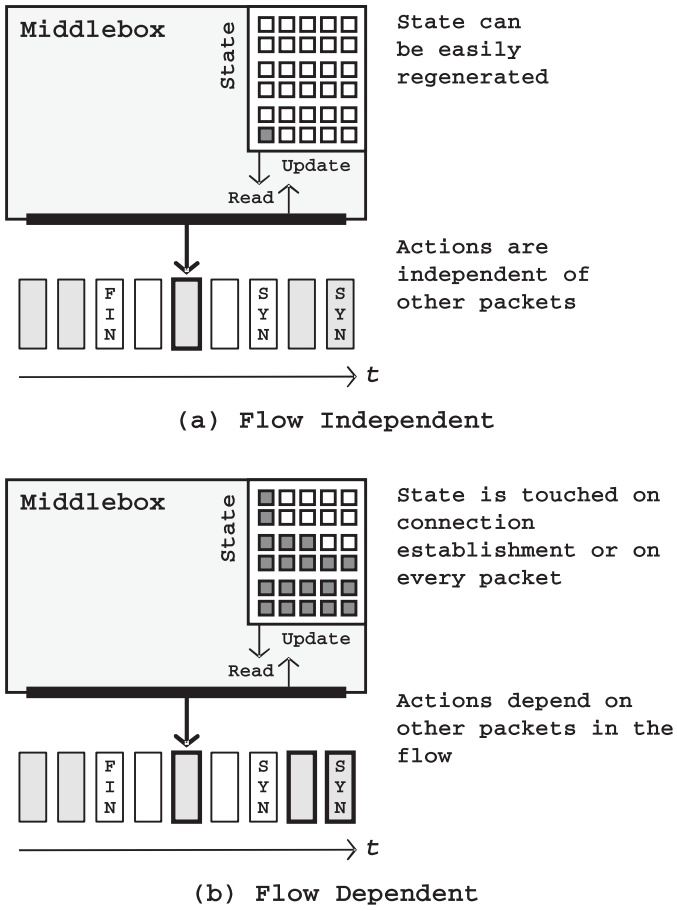


Figure 4.1: Middlebox types

previous or future packets. Others analyze packets at line speed and maintain state for each flow (e.g., intrusion prevention systems). We classify middlebox applications into two types based on their per-flow *statefulness* (Table 4.1).

Flow Independent. *Flow independent* refers to a middlebox that either maintains no state or contains state that depends only on the packet being inspected (independent of past or future packets), or contains state that can be regenerated automatically (Figure 4.1(a)). Examples of applications in this class include stateless firewalls that block all traffic matching a specified set of rules.

From an HA perspective, flow independent middleboxes can be made highly

available by having redundant instances. When one or more instances fail, traffic to the failed instance can be re-routed via the remaining active instances, without any impact to end-to-end connectivity.

Flow Dependent. *Flow dependent* middleboxes generate per-flow state (Figure 4.1(b)). Some do so only during flow establishment, while others update flow state on every packet. In flow dependent middleboxes, the per-flow state is vital to maintaining the end-to-end connectivity.

Middleboxes like NAT, for example, only create state—a port mapping—during flow establishment. Processing of other packets in the same flow involves only reads from the port mapping table. Vyatta's vRouter [129, 130] is one example of a commercial product that generates flow state on flow establishment. On failure, since a new flow state is created only during flow establishment, merely re-routing ongoing flows to standby hosts will not suffice. The standby hosts would drop the packets. For HA, middleboxes like NAT require flow state synchronization *only once* during flow establishment. This approach is commonly found in Netfilter Connection Tracking based HA systems, such as those employed by Vyatta [181].

Flow dependent middleboxes can also manipulate the state for every packet in the flow. For example, an inline Intrusion Prevention System (IPS) such as Suricata [171], intercepts each packet entering or exiting the network, performing deep packet inspection (DPI), tracking the protocol state machine, etc. Other common examples include full proxies, layer-7 firewalls, protocol accelerators, traffic managers, web content optimizers, etc. These are collectively known as application delivery controllers (ADC) [120, 125–128]. To provide HA to this class of middleboxes, the per-flow state has to be continuously replicated and kept up to date, should one wish to ensure a seamless failover. In the case of an inline IPS with no HA support, simply failing over to a redundant instance potentially terminates all existing TCP streams since the backup instance possess no knowledge of prior traffic. Existing HA solutions for flow dependent middlebox are either ad hoc, such as the one employed by F5's BIG-IP [179] or decide to drop flows altogether causing downtime, such as one used by Riverbed's Stingray Traffic Manager [180]. Router redundancy protocols like HSRP [154] and VRRP [177] only address part of the problem: how to re-route flows to a standby appliance in case of a failure. These

protocols do not address the problem of persisting session (flow) state, which is essential to maintaining end-to-end connectivity.

4.1.2 Requirements for Middlebox HA

Our goal in this chapter is to develop a generic, system level HA solution for flow dependent middleboxes. We assume a fail-stop failure model, in which network partitions are not tolerated. We present the following requirements that an ideal middlebox HA solution should satisfy.

R1. **Recovery-transparency.** The middlebox is often an invisible entity that lies along the network path between two end points.Thus, it is insufficient to transparently failover to a redundant middlebox for new flows. State from the failed middlebox must be recovered with consistency so that existing flows can continue with minimal interruption.

R2. **Low Performance Overhead.** Middleboxes process millions of packets per second. The performance overhead of the HA framework (latency and throughput) on individual flows must be minimal.

R3. **Tunable Policies.** When attempting to provide a transparent HA mechanism for a wide array of middlebox applications, the policy developer should be able to easily create HA policies that control where to place the backup for a given set of flows and when (with what frequency) to checkpoint the flows. These two parameters allow the designer to make tradeoffs between end-to-end latency and the overall system utilization in the middlebox cluster.

4.1.3 State of the Art

When considering the applicability of previous work, we focus on systems that can preserve the runtime state of the middlebox application in the failed replica, ensure transparent failover and maintain consistency of state after recovery. We find that none of the existing solutions can be readily applied to virtual middlebox applications in a cloud infrastructure.

Ad Hoc Solutions. Commercial middlebox products use ad hoc HA implementations. For example, one common approach involves a clustered configuration

with DNS load balancing and failover [180]. When a node fails, ongoing client connections are lost. Another typical approach is to use active-standby or active-active configurations with custom state replication techniques [179, 181] to provide transparent failure recovery. However, with such point solutions, the management complexity increases dramatically as the number of middleboxes in the network increases. This is apparent in enterprise networks today where there are as many middleboxes in the network as L3 devices [93], frequently resulting in device sprawl, network downtime due to misconfiguration, etc. Thus, at large scale, it becomes infeasible to configure and manage different HA solutions specific to each middlebox vendor. Our goal is to develop a generic framework at the system level that can be leveraged by a large class of applications, whose inner workings are well understood.

VM Replication. VM level HA techniques [16, 28, 37, 42, 61, 69, 83, 89] can be applied to arbitrary applications, to provide a completely stateful, consistent and transparent recovery (R1). However, these solutions do not satisfy requirement R2, since the coarse-grained protection granularity (the entire VM) degrades the performance of the middlebox application during normal operation. Similarly, at the VM granularity, very few HA policies can be instituted (R3).

The root cause of the performance degradation in replication based solutions arises from output buffering, a critical requirement for achieving consistent and transparent failover. Output buffering and commit involves buffering the VM's output (e.g., network packets) during an epoch and releasing the output only after the checkpoint for that epoch is committed at the backup. Suspending and resuming the entire VM, as is done for every checkpoint in systems like Remus [28, 69], delay the release of the output buffer and limit the frequency of checkpointing.

4.2 Pico Replication

The Pico Replication framework proposes a new approach to preserving state. Instead of suspending and checkpointing an entire middlebox at the VM level, Pico Replication capitalizes on the unique structure of middlebox applications to enable fine-grained flow-level replication. By operating on this fine-grained level, the middlebox can continue to process packets from other flows even as one flow

is suspended for checkpointing. With Pico Replication, the most valuable pieces of data in a middlebox are replicated at very high frequencies (1000 checkpoints per second), with little impact on the application's execution; satisfying all three requirements mentioned in Section 4.1.2.

Figure 4.2 shows the high level components that make up the Pico Replication framework. We assume a deployment model in which a dynamically scaling middlebox cluster comprises a number of replica VMs. Each replica runs three modules. First, the *State Management Module* (SMM) is responsible for managing flow-related state. The SMM manages and controls access to both a set of primary flow states for flows that are currently being processed by the local replica and a set of backup flow states for flows that are currently being processed elsewhere. Second, the *Packet Management Module* (PMM) is responsible for ensuring that packets enter and exit the middlebox at appropriate times. In particular, the PMM buffers incoming and outgoing packets in one of a set of per-flow or per-group-of-flow buffers to maintain output consistency in case of failure. Third, the *Replication Module* (RM) implements a replication policy by instructing the PMM to plug or unplug input buffers, while copying state from the SMM over the network to other replicas.

Finally, each replica is placed on a Software Defined Network (SDN) (e.g., an OpenFlow [165] network). The SDN controller assigns network flows to replicas and sets up flow forwarding paths in the network. The SDN controller also detects replica failure and reroutes network flows to failover.

The three fundamental operations involved in any state replication based HA system are: *identify state*, *replicate state* and *failover*. The Pico Replication framework is no exception; Table 4.2 describes the components responsible for providing each of these operations. The remainder of this section describes each component in more detail.

4.2.1 State Management Module

Pico Replication relies on the middlebox application to interface with the State Management Module to identify flow-related state. State identification reduces the overhead of replication and eliminates the need to suspend an entire VM for

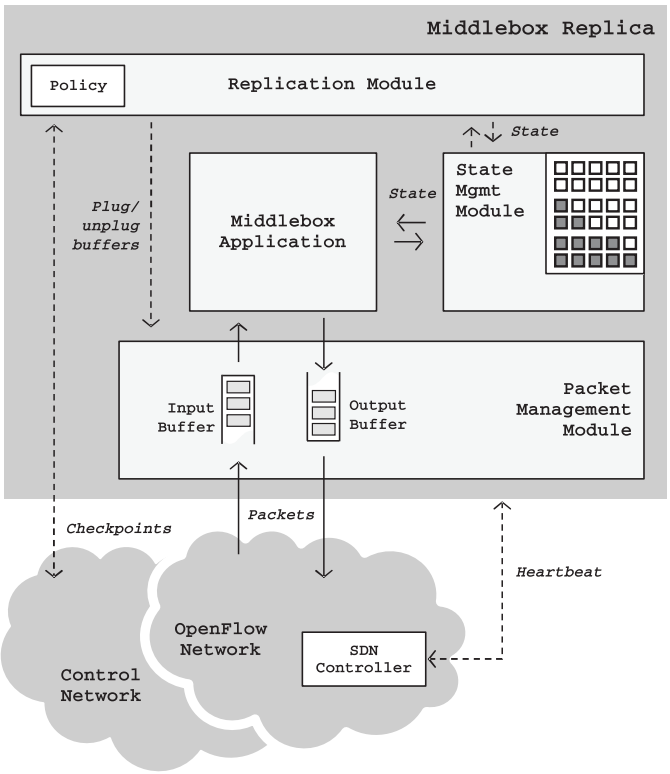


Figure 4.2: High level design of Pico Replication

HA Operation	Pico Replication Component
Identify State	State module
Replicate State	State, packet, and replication modules
Failover	State module and SDN

Table 4.2: Pico Replication components involved in each HA operation

checkpointing. To identify critical pieces of state in the middlebox, we leverage concepts from our earlier work, Split/Merge [85].

Identifying Flow State. Figure 4.3 shows the Split/Merge classification of middlebox state. Split/Merge classifies the state inside a middlebox VM into two types: *internal* and *external*. Internal state is specific to every replica in a middlebox clus-

ter, such as the operating system's buffer cache, other processes in the replica, etc. and does not need to be replicated. External state is further classified into *partitioned* or *coherent*. Partitioned state represents the set of per-flow states maintained by the middlebox application. Each replica exclusively owns a subset of the per-flow states. The per-flow states (partitioned state) represent the critical pieces of data that need to be persisted across failures, to ensure uninterrupted end-to-end connectivity. Coherent state represents global shared data such as static configuration information, non-critical statistics counters, etc. Since it is already shared (and likely replicated), we do not consider further replication of coherent state.

Flow-State Transactions. Like the Split/Merge system, the State Management Module exposes an interface to the application to both identify and control access to flow states. Importantly, it maintains a notion of a *transaction*. In other words, the State Management Module keeps track of when the middlebox is in the middle of processing a packet belonging to a particular flow and accessing the corresponding state. This transaction boundary provides information as to whether or not it is safe to replicate flow state over the network. The Replication Module (Section 4.2.3) will not copy flow state for flows that are in the middle of a transaction.

One-time Checkpointing. As described in Section 4.1.1, some flow dependent middleboxes create the per-flow state once during flow establishment (e.g., stateful NAT) and the state remains read-only throughout the lifetime of the flow. The SMM allows an application to indicate whether the flow's state was modified or not during the course of packet processing. By querying the SMM, the Packet Management Module (Section 4.2.2) or Replication Module (Section 4.2.3) will perform less work for unmodified state.

Active vs. Standby States. Unlike Split/Merge, the State Management Module also contains flow state that consists of backups of other replica's state, called *standby state*. Figure 4.4 depicts three replicas, each storing a set of active and standby state. In contrast to active state, standby state is not released to the application unless a failure has been signaled to the SMM from the SDN controller (Section 4.2.4). The transition from standby state to active state is simple: the SMM simply begins responding with the state when a middlebox application requests access to it. This transition is depicted in Figure 4.5.

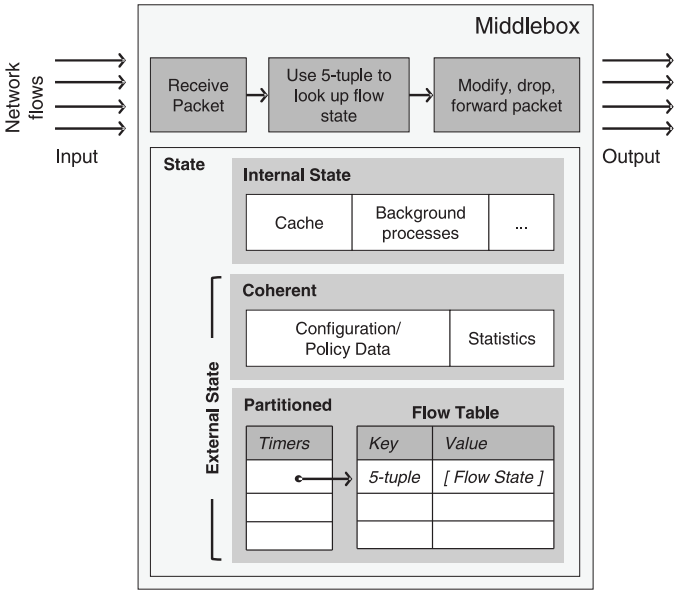


Figure 4.3: Anatomy of middlebox state

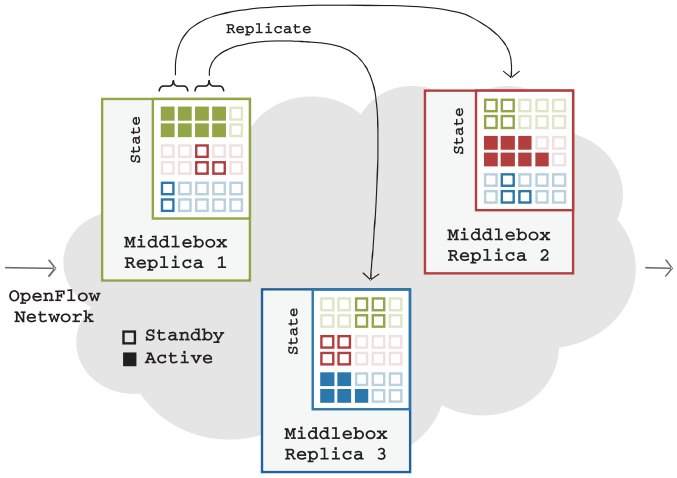


Figure 4.4: Pico replication of state

Overall, the Split/Merge [85] system provides the ability to classify middlebox application state and control access to it; however, it does not provide HA capabilities. PR and Split/Merge together provide robust system support for both elasticity and HA.

4.2.2 Packet Management Module

The Packet Management Module ensures per-flow state replication maintains consistency. It does this through input and output buffers and acting as an intermediary between a middlebox application and the network.

Output Buffering. Using output buffering, the system ensures that the network output from a middlebox replica is not seen by the external world until a checkpoint of the flow states is committed at the backup node. The rate at which the output buffer is released corresponds to the checkpoint frequency. Output buffering can introduce delay and burstiness into the egress network traffic if the checkpoint frequency is lower than the arrival rate of packets at the middlebox. In order to minimize the impact of output buffering, the checkpoint frequency has to be high.

PR leverages the independence of flows in a middlebox to perform efficient output buffering using two techniques. First, given that a middlebox is composed of hundreds of flows operating independently of one another, PR can maintain a different output buffer for each flow, eliminating the need to wait for other flows to be replicated. While one set of flows are being checkpointed, the middlebox can continue processing packets belonging to other flows. Second, individual flows can be checkpointed at a very high frequency, since their state is small relative to the rest of the state in the system. High frequency flow-level checkpointing allows the flow's output buffers to be released quickly, resulting in lower latency overhead and minimal impact on the shape of egress flow traffic.

Certain classes of applications may have reasons to require flow state checkpointing without output buffering. One example is Bro [78], an IDS, that analyzes every packet in the flow and maintains extensive state information. However, its state is not critical to forwarding of the flow. As a result, the PMM exposes an interface for an application to explicitly disable output buffering.

Input Buffering. A typical checkpoint begins by suspending any execution that may affect the state that comprises the checkpoint. In Pico Replication, the fact that no packets belonging to a particular flow are being processed implies that the flow state associate with that flow will not be accessed. Therefore, halting input of a particular flow is effectively a suspend operation and is sufficient for a checkpoint of the flow state to commence. Once a flow is suspended and its state is copied

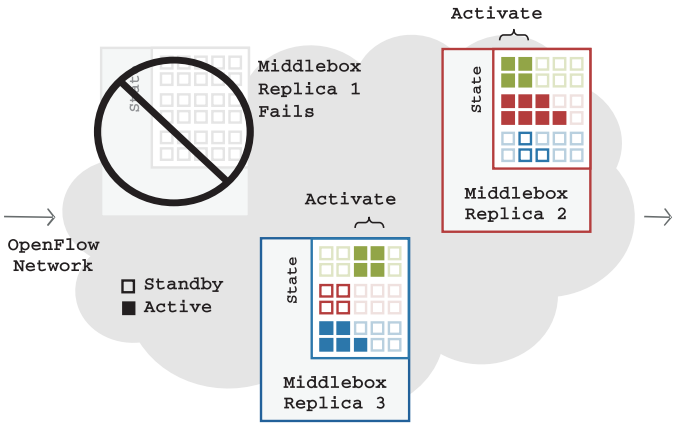


Figure 4.5: Transparent failure recovery

elsewhere, the appropriate output buffer is released.

Replication Groups. One or more flows can be grouped together into a single logical unit that can be replicated to the same target with the same checkpoint frequency. Batching flows into replication groups reduces the computational overhead of maintaining multiple replication streams. The replication target and the checkpoint frequency of each group can be configured independent of other replication groups in the system. Flows can also be moved across replication groups, allowing the system to dynamically adapt to changes in network traffic.

4.2.3 Replication Module

The Replication Module interacts with the State Management Module and the Packet Management Module in order to implement a policy for replication. A replication policy controls flow checkpointing in the following manner. First, the RM instructs the PMM to halt a flow. Then, the RM obtains the flow state from the SMM. Unless the SMM reports that no changes have been made to the flow state, the RM copies the flow state to an SMM elsewhere in the network. Finally, after receiving confirmation that the flow state is backed up, the RM instructs the PMM to release the output buffer. The RM also informs the SDN Controller of the flow backup targets so that it can quickly recover from failure.

Replication policies can modulate a number of variables, including the number

of replication groups in the system, assignment of flows to replication groups, the checkpoint frequency and the replication target for each group.

4.2.4 SDN Controller

The SDN controller is responsible for detecting and recovering from failure. Failure detection has been extensively researched in the past and there are several well known algorithms [1,43,106] and tools [111,113,182] to detect failures in a timely manner. When a replica failure is detected, the SDN controller signals the SMMs in other replicas to activate the hot-standby copies of flow states that belonged to the failed replica. The SDN then re-routes the flows to the (new) instances responsible for them. Apart from temporary packet loss, end-to-end connectivity remains intact.

4.3 Implementation

Our Pico Replication implementation extends FreeFlow, an implementation of the Split/Merge design paradigm [85]. FreeFlow provides some of the necessary building blocks: namely, it is able to identify and migrate flow states across replicas and reprogram packet forwarding rules in the network accordingly. The FreeFlow system consists of two main components: (a) an application library to interact with the middleboxes and the underlying hypervisor, (b) an OpenFlow [165] SDN controller, based on Floodlight [153]. Figure 4.6 shows how the FreeFlow library contributes to the implementation of Pico Replication. Figure 4.7 sketches the execution of a middlebox application that uses a combination of FreeFlow API's (`get_flow`, `put_flow`) and new APIs from our Packet Management Module implementation (`pkt_read`, `pkt_write`). The remainder of this section describes the implementation in more detail.

4.3.1 State Management Module

The State Management Module is implemented as an extension to the FreeFlow user space library. By default, the FreeFlow library exposes a set of APIs that allow middlebox applications to offload state management to FreeFlow, while focusing on the application logic. Specifically, the APIs enable the application to create per-

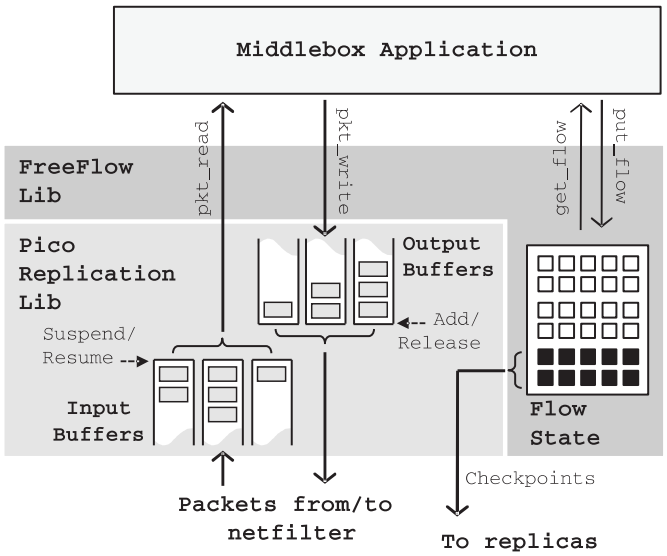


Figure 4.6: Input and output buffers in Pico Replication

```
while (1)
{
  pkt *p = pkt_read();              //read packet from network

  flow_key k = extract_key(p);      //get flow key from packet

  flow_state *f = get_flow(k);      //increment refcnt of flow k

  ... process packet ...

  pkt_write(k, p);                  //write packet to network

  put_flow(k);                      //decrement refcnt of flow k
}
```

Figure 4.7: A skeletal code demonstrating the control flow inside the middlebox application using the augmented FreeFlow API.

flow and coherent states (Figure 4.3).[1] The FreeFlow `get_flow`/`put_flow` in-

[1]Depending on application needs, the coherent state can either be strongly or eventually consistent. In our experience analyzing and building middleboxes, most shared middlebox state, statistics counters for example, require only eventual consistency. Similar to the Bayou [102] storage system approach, FreeFlow uses application-specific combiner functions to merge updates on eventually

terface enables the application to access the relevant state bound to a flow, as shown in Figure 4.6. This interface also enables the application to define the boundaries of a state transaction. We augmented the `put_flow` interface with an optional flag that allows the application to indicate whether the flow state was modified during the transaction (i.e., packet processing). Unmodified flows will not be replicated.

4.3.2 Packet Management Module

In order to be able to suspend and resume flows and control the release of application's outputs, the PMM interposes on the ingress path of a packet from the network to the application and vice versa. PR organizes flows in the system into replication groups, as described in Section 4.2.2. Every replication group has an associated input and output buffer. Packets typically enter an input buffer in the PMM before reaching the application and enter an output buffer in the PMM before exiting to the network.

The PMM is implemented using the Netfilter [163] kernel module to copy packets arriving at the network interface to a user space memory region, corresponding to an input buffer. Similarly, when packets are eventually sent from the output buffer out onto the network, the kernel module copies packets from the user specified buffer into kernel space and forwards it onto the appropriate network interface. Copying to/from user space incurs a high throughput overhead on the order of 50%. We are working on supporting additional frameworks for fast user space packet processing using direct I/O, like netmap [87], Intel's Data Plane Development Kit [152] and vPF_RING [164]. Alternatively, input and output buffers could be implemented at the Open vSwitch layer in the hypervisor, at the cost of increased hypercall overhead to manage the buffers.

Shown in Figure 4.6, from the middlebox's perspective, packets are received from an input buffer or sent to an output buffer using a library interface consisting of two primary functions: `pkt_read` and `pkt_write`, respectively. The application issues `pkt_read` to obtain a reference to packets arriving from the network. The PMM scans the input buffers for available packets and returns a reference to a packet from one of the buffers. If all input buffers are currently suspended,

consistent state from multiple replicas.

the API call blocks or returns an error code if called in non-blocking mode. The application can forward modified or unmodified packets or inject new ones onto an output buffer using the `pkt_write` functions. If the output buffer is full, the packet is dropped and the application is informed accordingly. We also supply a `pkt_write_direct` function to explicitly inform the system that a particular packet can be sent out immediately, rather than being sent to an output buffer. A worker thread scans the output buffers of various replication groups for available packets that can be injected back into the network.

4.3.3 Replication Module

The Replication Module implementation associates a separate thread with each replication group for checkpointing and replicating flow state to the group's target. At the start of a checkpoint epoch, the replication group's worker thread creates an output buffer. At the end of the epoch, the thread suspends the flows in the replication group by instructing the PMM to suspend the respective input buffer. Once the application has released any lingering references to flows in the replication group (this condition is exposed by the FreeFlow library based on reference tracking with `get_flow`/`put_flow`), the flow states are copied from the SMM to a temporary buffer. The PMM is instructed to resume the input buffer, thereby the flows in the group are also resumed. The RM uses the control network for carrying replication stream traffic between various nodes in the cluster. Once the checkpoint (in the temporary state buffer) is replicated to the target, the PMM is instructed to release the output buffer corresponding to the checkpoint.[2]

If the backup replica for a given replication group fails, the RM at the primary retries a few times before terminating the checkpointing thread and disabling input/output buffering for the associated replication group. It informs the SDN controller of this failure and waits until instructed by the controller to restart the replication process to a new backup target. During this period the flows associated with the replication group will not be protected against failures.

[2]Flows that are not protected are put into a default group whose input and output buffers are not subject to suspend/resume and add/release respectively.

4.3.4 SDN Controller

We augment the FreeFlow SDN controller to detect failure and activate standby flows in other replicas. When a middlebox replica VM fails, the SDN controller receives an OpenFlow [165] `PORT_DOWN` message from the host hypervisor's Open vSwitch [80]. Host failures (i.e., switch failures) are detected using the OpenFlow protocol's `ECHO` messages that serve as heartbeats between the controller and OpenFlow switches. The failure recovery process involves leveraging the FreeFlow SDN controller to re-route the flows—using OpenFlow—to the (new) instances responsible for them. Since the recovery procedure and the FreeFlow scale-in procedure are the same from the standby replica's standpoint, no special recovery logic is needed beyond that implemented in the FreeFlow library. The application is unaware of both scale-in and failure recovery.

4.4 Example Replication Policies

Using the tuning knobs provided by Pico Replication, one can implement a variety of policies ranging from a simple daisy chain replication to much more sophisticated and adaptive approaches, that can take into account parameters such as network load distribution, QoS, HA resource overhead, network topology, fault domains, etc. In our implementation, policies are specified to the SDN controller, which co-ordinates replication related tasks across the cluster, such as configuring replication groups and failure recovery. Here we describe two example policies.

Differentiated Pico Streams. Platform as a Service (PaaS) providers like Heroku [144] and Cloud Foundry [138] offer services such as key-value data stores, media streaming, SMS and push notification, big data analytics, and so on. When protecting middleboxes operating in these environments, the HA policy has to take into account the QoS requirements of the network flows and the system resources available for HA purposes.

A naïve HA policy could set a very high replication frequency and apply it to all flows in the system irrespective of their QoS requirements. While the performance impact on end-to-end throughput and latency would be minimal, the HA resource overhead (CPU and replication bandwidth) can quickly overwhelm the system.

Using Pico Replication's ability to replicate subsets of flows at different fre-

quencies, an adaptive policy can be implemented to take advantage of the variegated QoS demands of the flows. By monitoring the flows in the system, replication can be configured to checkpoint the flows as groups with different frequencies based on their QoS requirements. As a simple example, consider a scenario wherein the middlebox application cluster is processing a large number of HTTP flows and a relatively small number of flows involving memcached [142] clients and servers. Since a memcached transaction could complete in just one RTT, it is essential to replicate the flow state at a very high frequency, to keep latency impact negligible. HTTP transactions on the other hand could transfer hundreds of kilobytes of data [155]. If the latency impact on page load time is not perceivable, the HTTP flows can be replicated at a lower frequency, thereby conserving system resources.

Elastic Pico Chords. This policy is similar to techniques used in systems like Chord [99], Dynamo [34] and FAWN [7]. In an N node cluster, the nodes can be arranged in a ring using consistent hashing. To ensure even distribution of load, multiple virtual nodes can be assigned to each physical node, such that each virtual node owns a single non-contiguous slice of the ring. Incoming flows are assigned to the virtual node that follows the flow in the ring, in the clockwise direction. The backup for a given flow is placed at the virtual node immediately following its current virtual (primary) node.

Replication groups on a node can be automatically formed by grouping flows according to their backup destinations. Such groups can be split into smaller groups if subsets of its flows are to be replicated at different replication frequencies. When a node fails, its load is evenly dispersed across the rest of the cluster, such that the overall load balance in the cluster is maintained. However, unlike the DHT style query forwarding approach adopted by Chord, in an SDN environment, the network controller can be leveraged to automatically (re)route a flow to the appropriate replica (primary or backup).

Since the Chord style algorithm adapts to changes in cluster memberships, elastic middlebox applications can maintain the load distribution of both the primary and backup as the cluster scales out or scales in according to load.

4.5 Evaluation

We evaluated Pico Replication across different middlebox setups. We focused on the following goals:

- Demonstrate Pico Replication's ability to transparently failover a flow dependent middlebox, while incurring very low performance overhead during normal operation. (Requirements R1& R2)
- Demonstrate a simple Differentiated Pico Stream HA policy that provides low overhead HA to a class of flows, while maintaining performance isolation at scale. (Requirement R3)
- Using a homogeneous workload, study the impact on system utilization and performance as the replication frequency and the number of replication groups vary.

We evaluate PR's performance with applications that require continuous [78, 120, 125–128, 171], rather than one-time synchronization (Table 4.1), as the former fully exercises the replication subsystem. When comparing PR with other HA techniques, we do not evaluate HA approaches used in commercial middlebox systems. Some do not preserve flow state on failover [180] and many resort to in-house proprietary implementations [179] whose details are not disclosed publicly. Open source implementations such as Netfilter's Connection Tracking [163] perform one-time or highly coarse-grained synchronizations (e.g., during TCP state machine transitions). At the time of this writing, we were unable to find non-proprietary flow state preserving HA solutions for middleboxes requiring continuous synchronization. As a baseline comparison, we evaluate the performance of Remus [28], a VM replication based HA solution, as it provides transparent failure recovery (Requirement R1).[3]

Middlebox Application and Workload. We implemented a generic middlebox service, MBServ, that performs packet inspection and request routing. The functions performed by MBServ are representative of the operations carried out by

[3]The version of Remus used in our evaluation uses checkpoint compression, an optimization introduced in RemusDB [69] to performance overhead.

popular middlebox applications like the Suricata [171] intrusion prevention system and ADCs like Riverbed's Stingray Traffic Manager [127] that are used for layer-7 firewalling, request rate shaping and load balancing. Note that, by default, these systems do not support stateful failure recovery [180]. While traffic load is redistributed, ongoing connections handled by the failed node are dropped by other nodes in the cluster, due to lack of related flow state.

The MBServ application runs inside a VM that sits in front of a pool of back-end servers on a protected network. MBServ interposes on Client TCP requests from an external network. MBServ interacts with the Pico Replication library. It reads packets from the network interface, using the `pkt_read` call. The application maintains a TCP state machine for every flow and inspects the payload for occurrences of a predefined set of malicious strings. It then forwards packets to the destination network using the `pkt_write` call.

Incoming packets that do not conform to the protocol state machine of their respective flows, or have no corresponding flow in the system, are dropped. Hence, lack of middlebox state replication will result in loss of end-to-end connectivity on failover because, on failure, the flow will be reassigned to a new middlebox that has no prior knowledge of the flow.

Unless otherwise noted, all experiments use a fixed request and response size of 1400 bytes. Latency and CPU utilization values reported in this section correspond to the 95th percentile values, indicating peak latency and peak utilization, respectively.

Experimental Setup. Our experimental setup consists of a cluster of 8 heterogeneous physical hosts running the Xen [12] 4.2 hypervisor, Linux 3.4 kernel and Open vSwitch [80] based software OpenFlow switches. A hardware OpenFlow switch, BNT G8264, connects the physical hosts together. Pico Replication HA policies and FreeFlow's SDN modules are implemented on top of the Floodlight [153] OpenFlow controller. Unless stated explicitly, the middlebox VMs used throughout this evaluation were provisioned with two CPUs, on physical hosts with a quad-core Intel Xeon processor.

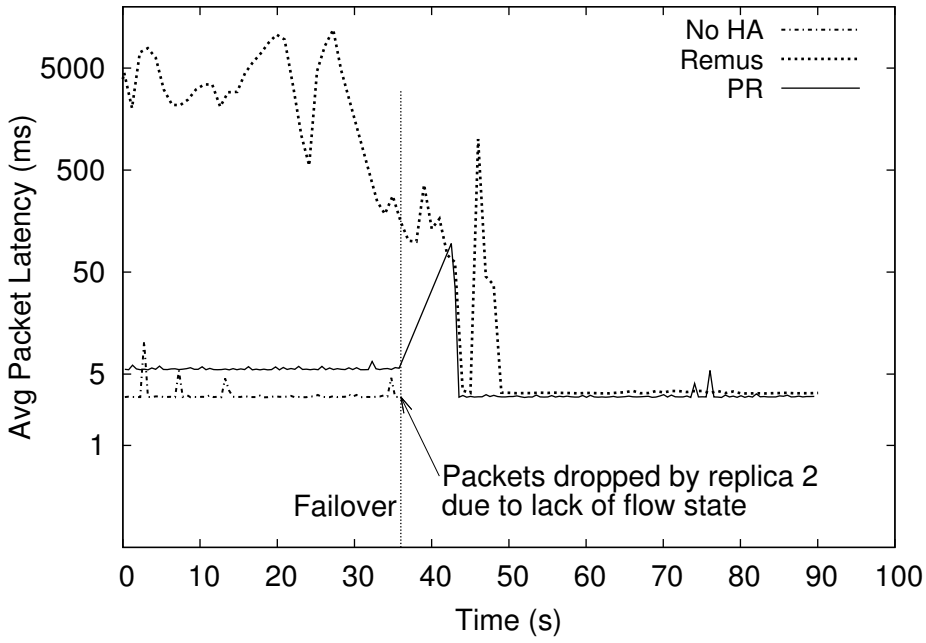


Figure 4.8: Transparent failure recovery in Pico Replication.

4.5.1 Stateful Failover

In this experiment, we demonstrate PR's ability to transparently failover a set of flows to another middlebox, without disrupting end-to-end connectivity. We provision a middlebox cluster with three VMs (MBServ1, MBServ2, and MBServ3) on three different hosts (H1, H2, and H3). Figure 4.8 shows the average end-to-end latency experienced by the client over a 500ms sampling window. MBServ1 is subjected to 50Mbps network traffic spread over 10 flows. After 35 seconds into the experiment, MBServ1 is destroyed. Failure is detected when the SDN controller receives an OpenFlow `PORT_DOWN` message from the host hypervisor's Open vSwitch.

Three different HA configurations are used in this experiment. In each case, the SDN controller reroutes flows from the failed replica (MBServ1) to one or both of the other replicas in the cluster. The HA configurations are as follows:

No HA. In the first case, when MBServ1 fails, its load is redistributed by the SDN

controller equally between MBServ2 and MBServ3. No state is replicated between the replicas in the cluster.

Remus. In the second case, we use Remus [28] as an example of a VM-level HA technique that provides transparent failure recovery. The state of MBServ1 is constantly replicated by Remus at a frequency of 40Hz to a different host in the cluster.[4] We denote MBServ1's passive backup copy created by Remus as MBServ1-B. We applied checkpoint compression [69] to reduce the latency overhead of checkpointing. On failure detection, the SDN controller reroutes the flows accordingly to VM1-B.

PR. In the third case, Pico Replication divides the 10 flows handled by MBServ1 into two replication groups, targeted to MBServ2 and MBServ3, respectively. Since only the flow state is replicated, PR is able to achieve a very high replication frequency of 1000Hz. On failure detection, the SDN controller immediately activates the relevant flow states in MBServ2 and MBServ3 and reroutes the flows accordingly.

Figure 4.8 shows the performance of each HA strategy. In the "No HA" scenario, even though the controller automatically reassigns the flows from MBServ1 to MBServ2 and MBServ3, they do not have the state pertaining to the flows. As a result, the packets belonging to flows from MBServ1 are dropped. With Remus, when MBServ1 fails, its backup copy MBServ1-B resumes execution in host H2, from the most recent and consistent checkpoint of MBServ1. Since a consistent copy of all the flow states is available at MBServ1-B, traffic processing continues uninterrupted.[5] With PR, each replica absorbs a share of the load from MBServ1 and is able to continue service the flows, since it possesses the up-to-date copy of the flow states required to process the packets.

Without any replication, the end-to-end latency is approximately 3ms. With Remus, the overhead is prohibitively high due to the much lower replication fre-

[4]It is possible to reach replication frequencies of 100Hz with Remus. However, no useful work gets accomplished by the VM at such frequencies since the VM suspend/resume calls themselves take up approximately 7-8ms to complete.

[5]With Remus, on failure recovery, host H2 now has both MBServ2 and MBServ1-B. In other words, the entire load from MBServ1 is shed onto host H2. This is an unfortunate consequence of VM replication solutions—the inability to provide fine grained load balanced failover.

	Replication Freq. (Hz)	Throughput (txn/s)	Latency (ms)
No HA	N/A	6683 [95]	54.48
Pico Replication	1000	6468 [687]	62.95
Remus	100	30 [2]	207.72

Table 4.3: Performance overhead of Pico Replication vs. Remus. Throughput metric is HTTP transactions per second [w/standard deviation]. Latency metric is the application perceived RTT per HTTP transaction. 95th percentile latency values are shown.

quency (40Hz), and the overhead associated with suspending and resuming the entire VM. With Pico Replication, the latency is very close to native case, approximately 5ms. This is due to the fact that PR replicates relevant flow states at a very high frequency (1000Hz) to MBServ2 and MBServ3, thereby ensuring quick release of the output buffers associated with each flow. After failure of MBServ1, all scenarios eventually reach performance matching "No HA" because they are no longer replicating the flow state or the VM.

4.5.2 Pico vs. VM Replication

To demonstrate Pico Replication's performance benefits compared to Remus, we evaluate the performance overhead on a HTTP style flow. The traffic was generated using the Flowgrind [178] tool, for a period of 120 seconds. The request sizes follow a lognormal distribution, with of 10KB, variance of 1KB and a maximum request size of 100KB. The same random seed was used for all trials of the experiment. We measured the application perceived response times per request and the total number of transactions per second.

As shown in Table 4.3, Pico Replication imposes a 15.5% latency overhead and a 3.2% throughput drop compared to a 2.8X latency overhead and a 99.5% drop in throughput imposed by Remus.[6] Again, the low overhead of PR can be attributed to its relatively high checkpoint frequency over Remus (1000Hz vs. 100Hz).

[6] The higher variability in throughput with PR is due to frequent context switches between user and kernel mode, during packet processing. The context switches are an artifact of the PMM's heavy use of the Netfilter [163] framework.

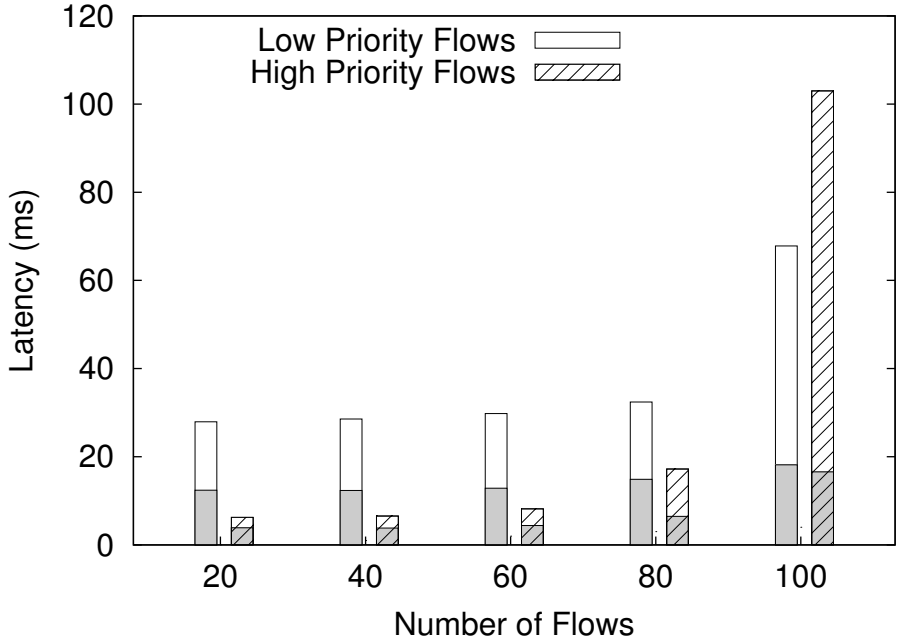


Figure 4.9: A simple policy that creates two replication groups with different frequencies and targets based on flow priority. The shaded portion of the bar at the bottom indicates the baseline (unprotected) system performance. At 100 flows, MBServ handles a sustained load of 175Mbps.

4.5.3 Differentiated Pico Streams - An Example

In this experiment, we showcase PR's ability to support various flexible and adaptive HA policies with a simple example. Consider a hypothetical scenario where MBServ processes a set of low and high priority flows. We assume that the ratio of low to high priority flows is 3:1. We setup the low priority flows with a rate of 1Mbps and the high priority flows with a rate of 4Mbps each. Then, we created a simple HA policy that identifies and classifies flows according to their priority (e.g., by their port numbers) and creates two replication groups for each priority respectively, each with a different replication target. By replicating low priority flows at a lower frequency (100Hz) than high priority flows (1000Hz), the middlebox application can conserve CPU resources (the effect of checkpoint frequency on CPU utilization is examined in more detail in Section 4.5.4).

The performance impact of the differentiated replication policy described above is shown in Figure 4.9 As the number of flows in the system (load) increases from 20 to 80, PR is able to provide performance isolation to the high priority group despite the increasing number of lower priority flows in the system. At 100 flows, the performance begins to degrade quickly, indicating that more resources need to be allocated to the system to maintain the same quality of service. However, given a fixed amount of resources, PR enables a HA policy to be instantiated on a middlebox that will trade performance for resources on a per-flow basis by adjusting checkpoint frequencies.

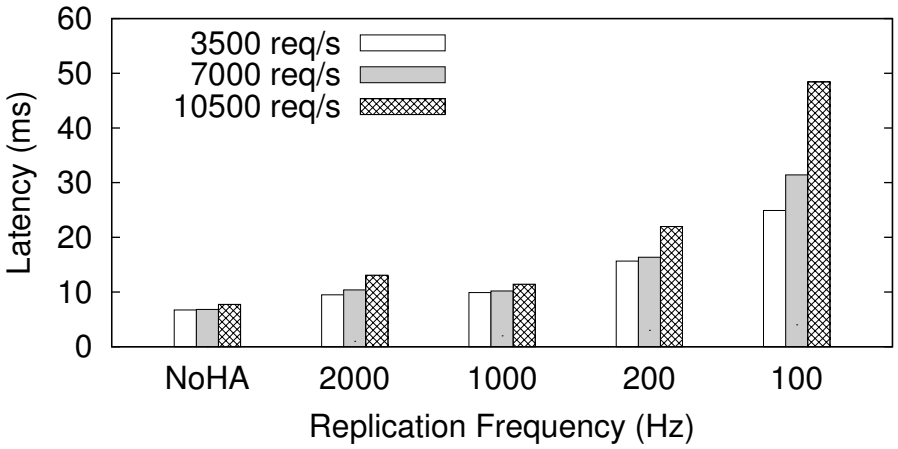
4.5.4 Performance vs. Utilization Trade-Off

In the following set of experiments, we analyze the parameters that are available at a policy designer's disposal, when attempting to strike a balance between performance and system utilization. The workload for this experiment models communication between two tiers of a multi-tiered application deployment, where each tier is in a separate network. The clients in one tier generate traffic at a rate of 3500 requests per second using a fixed sized connection pool. The two tiers are scaled gradually by adding more resources (VMs) such that the request rate increases in units of 3500.

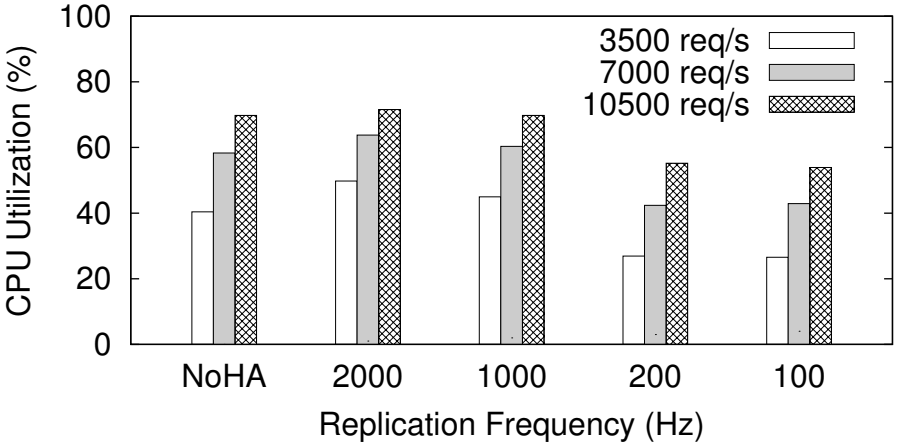
Impact of Replication Frequency. In general, higher checkpoint frequencies (or lower checkpoint intervals) result in quicker release of network output buffers, thereby minimizing the impact on end-to-end latency, at the cost of higher CPU usage. Figure 4.10a shows the impact on end-to-end latency at various replication frequencies when the load on MBServ increases progressively.

Figure 4.10b shows the corresponding CPU utilization. As the replication frequency decreases, the CPU usage decreases. At very low replication frequencies of 200Hz and 100Hz, the TCP endpoints themselves back off, thus reducing the overall load on the system.

While PR can be configured to replicate state at 2kHz (or 500us), the performance improvement is negligible at the cost of increased CPU usage, denoting a point of diminishing return. Also, at high loads (e.g., 10K requests/s), the 2kHz replication frequency begins to have a negative impact on the performance of MB-



(a) Performance



(b) System utilization

Figure 4.10: Impact of replication frequency at various loads

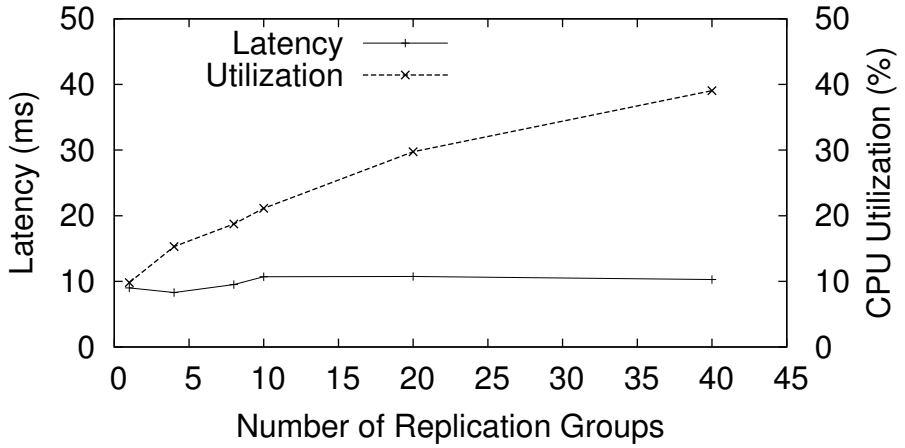


Figure 4.11: Impact of number of replication groups on latency and system utilization on a 8 CPU middlebox VM. (Latency values are computed using the geometric mean.)

Serv, as observed in Figure 4.10a. This is due to the fact that checkpointing at very low intervals leaves no room for the system to do any useful work (packet processing).

Impact of Replication Groups. Figure 4.11 shows the performance impact and CPU utilization as a function of number of concurrent replication groups in the system. We allocate 8 CPUs to the middlebox VM in this experiment. MBServ is processing 80Mbits/s worth of traffic, spread across 40 flows. The flow state is replicated at a rate of 1000Hz. Given a high replication frequency and a fixed load, the number of concurrent replication groups has minimal impact on the performance. However, as the number of replication groups (a thread per replication group) increase, the context switch overhead is no longer negligible, resulting in increased CPU utilization.

4.6 Summary

Middleboxes have become an integral part of networks of various scales, to an extent that today they constitute some of the most critical pieces in enterprise infrastructures [93]. As a result, when failures occur, entire infrastructures crash in

a spectacular fashion [134, 140], taking down hosted applications for hours at a stretch [147]. This chapter described the design and implementation of Pico Replication (Sections 4.2 & 4.3), a system-level HA framework for middleboxes built on the FreeFlow platform (Chapter 3). The independence of flows in a middlebox enables the framework to reduce latency overhead through high frequency replication of individual flows, while the middlebox continues to process other flows. By providing each flow with its own checkpoint frequency, output buffer and backup target, Pico Replication enables a rich set of policies (Section 4.4) that allow the administrator to balance—per-flow—performance and utilization. Pico Replication is an important addition to the growing set of tools that help middleboxes become more "cloud ready": software-defined, extensible, scalable, and highly available.

Chapter 5

Transparent High Availability for Databases

This chapter presents a system-level abstraction that provides transparent high availability to commodity database systems. High availability (HA) is a fundamental requirement when building a database management system (DBMS). Providing HA guarantees as part of the DBMS adds a substantial amount of complexity to the DBMS implementation. A new active-standby HA technique is proposed that is based on running the DBMS in a virtual machine (VM) and pushing much of the complexity associated with HA out of the DBMS, relying instead on the capabilities of the *virtualization layer*. The approach is based on Remus [28], a commodity HA solution implemented in the virtualization layer, that uses asynchronous virtual machine (VM) state replication to provide transparent HA and failover capabilities. While Remus and similar systems can protect a DBMS, database workloads incur a performance overhead of up to 32% as compared to an unprotected DBMS.

The sources of performance overhead are identified and optimizations are proposed to mitigate the problems. The proposed optimizations are implemented in a DBMS-aware VM replication system, called RemusDB [69, 70]. Experimental evaluations using two popular database systems and industry standard benchmarks show that for certain workloads, RemusDB provides very fast failover (≤ 3 seconds of downtime) with performance overhead as low as 3% when compared to an unprotected DBMS. RemusDB enables existing, deployed database systems to be

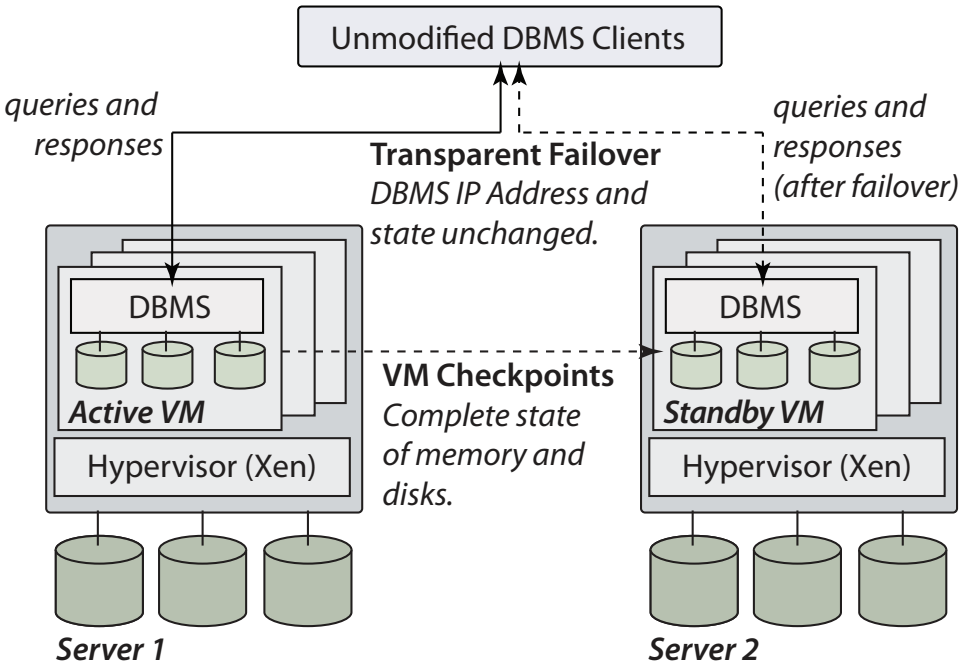


Figure 5.1: RemusDB system architecture

made more reliable with a minimum of risk, cost, and effort.

5.1 Background and System Overview

The RemusDB setup is illustrated in Figure 5.1. Two servers are used to provide HA for a DBMS. One server hosts the *active* VM, which handles all client requests during normal operation. As the active VM runs, its entire state including memory, disk, and active network connections are continuously checkpointed to a *standby* VM on a second physical server. The objective is to tolerate a failure of the server hosting the active VM by *failing over* to the DBMS in the standby VM, while preserving full ACID transactional guarantees. In particular, the effects of transactions that commit (at the active VM) before the failure should persist (at the standby VM) after the failover, and failover should not compromise transaction atomicity.

During normal operation, Remus takes frequent, incremental checkpoints of

the complete state of the virtual machine on the active server. The time between two checkpoints is referred to as an *epoch*. These checkpoints are shipped to the standby server and "installed" in the virtual machine there. The checkpoints also act as heartbeat messages from the active server (Server 1) to the standby server (Server 2). If the standby times out while waiting for a checkpoint, it assumes that the active server has failed. This causes a failover, and the standby VM begins execution from the most recent checkpoint that was completed prior to the failure. This failover is completely transparent to clients. When the standby VM takes over after a failure, it has the same IP address as the active VM, and the standby server's hypervisor ensures that network packets going to the (dead) active VM are automatically routed to the (live) standby VM after the failure, as in live VM migration [25]. In checkpoint-based whole-machine protection systems like Remus, the virtual machine on the standby server does *not* mirror the execution at the active server during normal operation. Rather, the activity at the standby server is limited to installation of incremental checkpoints from the active server, which reduces the resource consumption at the standby.

A detailed description of Remus's checkpointing mechanism can be found in [28]. A brief overview is presented in this section. Remus's checkpoints capture the entire state of the active VM, which includes disk, memory, CPU, and network device state. For disk checkpointing, Remus uses an asynchronous disk replication mechanism with writes being applied to the active VM's disk and at the same time asynchronously replicated and buffered in memory at the standby VM, until the end of the current epoch. When the next checkpoint command is received at the standby VM, it flushes the buffered writes to its local disk. If failure happens in the middle of an epoch, the in-memory state is discarded and the standby VM is resumed with a consistent state from the last committed checkpoint on its local disk. Memory and CPU checkpoints are implemented very similar to live VM migration [25] with several optimizations discussed in [28].

Remus's checkpoints capture both the state of the database and the internal execution state of the DBMS, e.g., the contents of the buffer pool, lock tables, and client connection state. After failover, the DBMS in the standby VM begins execution with a completely warmed up buffer pool, picking up exactly where the active VM was as of the most recent checkpoint, with all session state, TCP state,

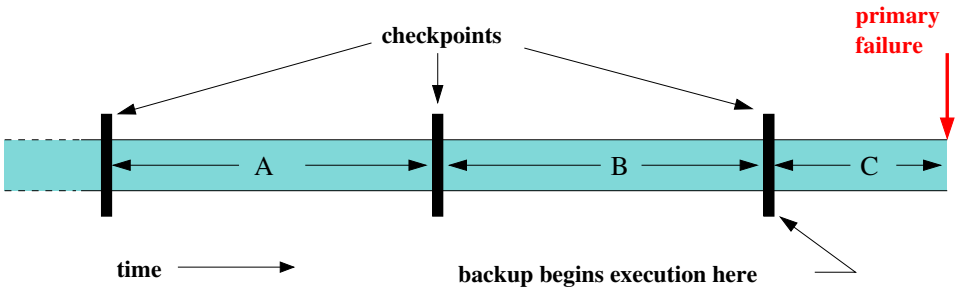


Figure 5.2: A primary server execution timeline

and transaction state intact. This fast failover to a warm backup and with no loss of client connections is an important advantage of our approach. Some DBMS-level HA solutions provide similar features, but these features add more code and complexity to the already complex systems. With our approach, these features are essentially free.

Figure 5.2 shows a simplified timeline illustrating checkpoints and failover. In reality, checkpoint transmission and acknowledgement is carefully overlapped with execution to increase performance while maintaining consistency [28]. However, the simplified timeline shown in Figure 5.2 is sufficient to illustrate the important features of this approach to DBMS high availability. When the failure occurs in Figure 5.2, all of the work accomplished by the active server during epoch C is lost. If, for example, the active server had committed a database transaction T during epoch C, any trace of that commit decision will be destroyed by the failure. Effectively, the execution of the active server during each interval is *speculative* until the interval has been checkpointed, since it will be lost if a failure occurs. Remus controls *output commit* [100] to ensure that the external world (e.g., the DBMS clients) sees a consistent view of the server's execution, despite failovers. Specifically, Remus queues and holds any outgoing network packets generated by the active server until the completion of the next checkpoint. For example, outgoing packets generated by the active server during epoch B in Figure 5.2 will be held by Remus until the completion of the checkpoint at the end of B, at which point they will be released. Similarly, a commit acknowledgement for transaction T, generated during epoch C, will be held by Remus and will be lost when the

failure occurs. This *network buffering* ensures that no client will have been able to observe the speculative commit of T and conclude (prematurely or incorrectly) that T is durably committed. The output commit principle is also applied to the disk writes generated at the active server during an epoch. At the standby server, Remus buffers the writes received from active server during epoch B and releases them to its disk only at the end of the epoch. In the case of failure during epoch C, Remus discards the buffered writes of this epoch, thus maintaining the overall consistency of the system.

For a DBMS, the size of a Remus checkpoint may be large, which increases checkpointing overhead. Additionally, network buffering introduces message latency which may have a significant effect on the performance of some database workloads. RemusDB extends Remus with optimizations for reducing checkpoint size and for reducing the latency added by network buffering. We present an overview of RemusDB in Section 5.2, and discuss the details of these optimizations in Section 5.3 and Section 5.4.

5.2 Adapting Remus to Database Workloads

While the simplicity and transparency with which Remus provides high availability is desirable, applying Remus to database workloads is not an ideal fit for a number of reasons. First, as described above, Remus continuously transmits checkpoints of the running virtual machine to the backup host, resulting in a steady flow of replication traffic that is proportional to the amount of memory that has changed between checkpoints; the large amount of memory churn in database workloads results in a high degree of replication traffic. The large amount of replication data makes checkpoints slower and results in a significant performance overhead for database workloads.

Second, the fact that Remus controls output commit by buffering every transmitted packet is over-conservative for database systems, which already provide higher-level transactional semantics. Client-server interactions with a database system typically involve several round trips on the fast, local area network. Within a transaction, the delay introduced by network buffering on messages from the server in each round trip results in an amplification of Remus's existing latency over-

heads. Moreover, the high memory churn rate of database workloads compounds this problem by requiring longer checkpoint epochs, resulting in longer delays for network buffering. For example, in a run of the TPC-C [174] benchmark on the PostgreSQL DBMS in our experimental setting (described in Section 5.6) we observed that Remus introduced an overhead of 32% compared to the unprotected case. Turning off network buffering for this benchmark run reduced the overhead to 7%.

In designing RemusDB, we aimed to adapt Remus to address these two issues. In both cases, we observed that Remus's goal of providing HA that is completely transparent to the DBMS was excessively conservative and could be relaxed, resulting in a large reduction in overhead. More precisely, we made the following two observations:

1. **Not all changes to memory need to be sent to the backup.** In attempting to maintain an exact replica of the protected VM on the backup system, Remus was transmitting every page of memory whose contents changed between epochs. However, many page updates can either be reconstructed, as with clean pages in the buffer pool that can be reloaded from disk, or thrown away altogether, in the case of working memory that can be recomputed or safely lost.

2. **Not all transmitted messages need output commit.** Buffering transmitted messages until the checkpoint that generated them has been protected prevents the system from exposing execution state that is rolled back (and so lost) in the event of failure. In a DBMS environment, this intermediate state is already protected by transaction boundaries. In light of this, we may relax output commit to the point that it preserves transactional semantics.

In addition to relaxing the comprehensiveness of protection in Remus to reduce overhead, our analysis of database workloads revealed one additional insight about these workloads that allowed further optimization:

3. **While changes to memory are frequent, they are often small.** Remus uses hardware page protection to identify the pages that have changed in a given

checkpoint, and then transfers those pages to the backup at page granularity. Our analysis revealed that memory updates in database workloads were often considerably smaller than page size, and could consequently be compressed fairly effectively.

Remus was adapted in light of each of these observations, in order to provide more efficient high availability for database workloads. Section 5.3 discusses optimizations related to how memory is tracked on the primary VM and replicated over the network to the backup. Section 5.4 describes how latency overheads have been reduced by relaxing network buffering in some situations.

5.3 Memory Optimizations

Remus takes a deliberately simple approach to memory checkpointing: at every checkpoint, it copies all the pages of memory that change from the active host and transmits them over the network to the backup host. The authors of Remus argue that this simplicity is desirable: it provides high availability with an acceptable degree of overhead, with an implementation that is simple enough that one can have confidence in its correctness, regardless of the target application or hardware architecture. This is in stark contrast to the complexity of previous systems, even those implemented in the hypervisor [16]. And while this argument for simplicity holds for database systems, the overhead penalty is higher: database workloads tend to modify more memory in each checkpoint epoch than other workloads. This section describes a set of optimizations designed to reduce this overhead.

5.3.1 Sending Less Data

Compressing checkpoints is beneficial when the amount of data to be replicated is large, and the data contains redundancy. Our analysis found that both of these conditions apply to database workloads: (a) they involve a large set of frequently changing pages of memory (most notably buffer pool pages), and (b) the memory writes often change only a small part of the pages on which they occur. This presents an opportunity to achieve a considerable reduction in replication traffic by only sending the actual changes to these pages.

When the Xen hypervisor runs on a physical machine, it creates a privileged VM called *domain 0* for controlling other VMs running on that physical machine. Domain 0 serves as an administrative front-end for the Xen hypervisor and manages the creation, configuration, and control of other VMs on the physical machine. In RemusDB, we implement checkpoint compression by maintaining in domain 0 an LRU-based cache of frequently changing pages in the protected VM obtained from previous checkpoints in that VM. A per-VM cache is maintained in domain 0 if there are multiple protected VMs. Our experimentation showed that a cache size of 10% of VM memory offers the desired performance improvement while maintaining an acceptable memory footprint in domain 0. When sending pages to the backup, we first check to see if the previous version of the page exists in this cache. If it does, the contents of the two pages are XORed, usually resulting in a page that contains mostly zeros, reflecting the large amount of identical data. The result is then run-length encoded for transmission. If the page is not found in the cache, it is sent uncompressed, and is added to the cache using the standard LRU eviction policy.

The original Remus work maintained that asynchronous, pipelined checkpoint processing while the active VM continues to execute is critical to minimizing the performance impact of checkpointing. The benefits of this approach were evident in implementing checkpoint compression: moving the implementation into an asynchronous stage and allowing the VM to resume execution in parallel with compression and replication in domain 0 halved the overhead of RemusDB.

5.3.2 Protecting Less Memory

Compressed checkpoints help considerably, but the work involved in taking and sending checkpoints is still proportional to the amount of memory changed between checkpoints. In this section, we discuss ways to reduce checkpoint size by selectively *ignoring* changes to certain parts of memory. Specifically, a significant fraction of the memory used by a DBMS goes into the buffer pool. Clean pages in the buffer pool do not need to be sent in Remus checkpoints if they can be regenerated by reading them from the disk. Even dirty buffer pool pages can be omitted from Remus checkpoints if the DBMS can recover changes to these pages from the

transaction log.

In addition to the buffer pool, a DBMS uses memory for other purposes such as lock tables, query plan cache, working memory for query operators, and connection state. In general, memory pages whose contents can be regenerated, or alternatively can be safely thrown away may be ignored during checkpointing. Based on these observations, we developed two checkpointing optimizations: *disk read tracking* and *memory deprotection*.

Disk Read Tracking

Remus, like the live VM migration system on which it is based [25], uses hardware page protection to track changes to memory. As in a copy-on-write process fork, all of the page table entries of a protected virtual machine are set to read only, producing a trap when any page is modified. The trap handler verifies that the write is allowed, then updates a bitmap of "dirty" pages, which determines the set of pages to transmit to the backup server at each checkpoint. This bitmap is cleared after the checkpoint is taken.

Because Remus keeps a synchronized copy of the disk on the backup VM, any pages that have been read from disk into memory may be safely excluded from the set of dirty pages, as long as the memory has not been modified after the page was read from disk. Our implementation interposes on disk read requests from the virtual machine and tracks the set of memory pages into which the reads will be placed, and the associated disk addresses from which those pages were read. Normally, the act of reading data from disk into a memory page would result in that page being marked as dirty and included in the data to be copied for the checkpoint. Our implementation does *not* mark that page dirty, and instead adds an annotation to the replication stream indicating the sectors on disk that may be read to reconstruct the page remotely.

Normally, writes to a disk pass through the operating system's (or DBMS's) buffer cache, and this will inform Remus to invalidate the read-tracked version of the page and add it to the set of pages to transmit in the next checkpoint. However, it is possible that the contents of the sectors on disk that a read-tracked page refers to may be changed without touching the in-memory read-tracked page. For exam-

ple, a process different from the DBMS process can perform a direct (unbuffered) write to the file from which the read-tracked page is to be read after failure. In this case, read tracking would incorrectly recover the newer version of the page on failover. Although none of the database systems that we studied exhibited this problem, protecting against it is a matter of correctness, so RemusDB maintains a set of backpointers from read-tracked pages to the associated sectors on disk. If the VM writes to any of these sectors, we remove the page from the read tracking list and send its contents normally.

Memory Deprotection

Our second memory optimization aims to provide the DBMS with a more explicit interface to control which portions of its memory should be deprotected (i.e., not replicated during checkpoints). We were surprised to find that we could not produce performance benefits over simple read tracking using this interface.

The idea for memory deprotection stemmed from the Recovery Box [9], a facility for the Sprite OS [75] that replicated a small region of memory that would provide important recent data structures to speed up recovery after a crash (PostgreSQL session state is one of their examples). Our intuition was that RemusDB could do the opposite, allowing the majority of memory to be replicated, but also enabling the DBMS to flag high-churn regions of working memory, such as buffer pool descriptor tables, to be explicitly deprotected and a recovery mechanism to be run after failover.

The resulting implementation was an interesting, but ultimately useless interface: The DBMS is allowed to deprotect specific regions of virtual memory, and these addresses are resolved to physical pages and excluded from replication traffic. On failover, the system would continue to run but deprotected memory would suddenly be in an unknown state. To address this, the DBMS registers a *failover callback handler* that is responsible for handling the deprotected memory, typically by regenerating it or dropping active references to it. The failure handler is implemented as an idle thread that becomes active and gets scheduled only after failover, and that runs with all other threads paused. This provides a safe environment to recover the system.

While we were able to provide what we felt was both a natural and efficient implementation to allow the deprotection of arbitrary memory, it is certainly more difficult for an application writer to use than our other optimizations. More importantly, we were unable to identify any easily recoverable data structures for which this mechanism provided a performance benefit over read tracking. One of the reasons for this is that memory deprotection adds CPU overhead for tracking deprotected pages during checkpointing, and the savings from protecting less memory need to outweigh this CPU overhead to result in a net benefit. We still believe that the interface may be useful for other applications and workloads, but we have decided not to use it in RemusDB.

To illustrate our reasoning, we ran a TPC-H [173] benchmark on PostgreSQL with support for memory deprotection in our experimental setting. Remus introduced 80% overhead relative to an unprotected VM. The first data structure we deprotected was the shared memory segment, which is used largely for the DBMS buffer pool. Unsurprisingly, deprotecting this segment resulted in roughly the same overhead reduction we achieved through read tracking (bringing the overhead down from 80% to 14%), but at the cost of a much more complicated interface. We also deprotected the dynamically allocated memory regions used for query operator scratch space, but that yielded only an additional 1% reduction in overhead. We conclude that for the database workloads we have examined, the *transparency vs. performance* tradeoff offered by memory deprotection is not substantial enough to justify investing effort in complicated recovery logic. Hence, we do not use memory deprotection in RemusDB.

5.4 Commit Protection

Irrespective of memory optimizations, the single largest source of overhead for many database workloads on the unmodified Remus implementation was the delay introduced by buffering network packets for controlling output commit. Client-server interactions in DBMS environments typically involve long-lived sessions with frequent interactions over low-latency local area networks. For example, a TPC-C [174] transaction on PostgreSQL in our experiments has an average of 32 packet exchanges between client and server, and a maximum of 77 packet ex-

changes. Remus's network buffering delays all these packets; packets that might otherwise have round trip times on the order of hundreds of microseconds are held until the next checkpoint is complete, potentially introducing two to three orders of magnitude in latency per round trip.

In RemusDB, we exploit database transaction semantics to avoid much of Remus's network buffering, and hence to eliminate much of the performance overhead that network buffering introduces. The purpose of network buffering in Remus is to avoid exposing the client to the results of speculative server processing until it has been checkpointed. In RemusDB, we relax this behavior by allowing server communications resulting from speculative processing to be released immediately to the client, *but only within the scope of an active database transaction*. If the client attempts to commit a transaction, RemusDB will buffer and delay the commit acknowledgement until that transaction is safe, i.e., until the processing of that transaction has been checkpointed. Conversely, if a failure occurs during the execution of such a transaction, RemusDB will ensure that the transaction is aborted on failover. Relaxing Remus's network buffering in this way allows RemusDB to release most outgoing network packets without delay. However, failover is no longer completely transparent to the client, as it would be in Remus, as a failover may necessitate the abort of some in-progress transactions. As long as failures are infrequent, we expect this to be a desirable tradeoff.

To implement this approach in RemusDB, we modified the hosted DBMS to implement a protocol we call *commit protection*. The commit protection protocol requires fine (message level) control over which outgoing messages experience network buffering and which do not. To support this, RemusDB generalizes Remus's communication abstraction. Stream sockets, which are implemented on top of TCP, guarantee in-order message delivery, and database systems normally use stream sockets for communication with clients. In RemusDB, each stream socket can be in one of two states: *protected* or *unprotected*. RemusDB provides the hosted DBMS with *protect* and *deprotect* operations to allow it to change the socket state. Outgoing messages sent through a protected socket experience normal Remus network buffering, i.e., they are delayed until the completion of the next Remus commit. Messages sent through an unprotected socket are not subjected to network buffering and are released immediately. RemusDB preserves

At COMMIT WORK:

```
protect the client's socket
perform normal DBMS commit processing
send the COMMIT acknowledgement
deprotect the socket
```

On failover (at the standby host):

```
for each active transaction t do
    if t is not committing then ABORT t
end for
```

Figure 5.3: The Commit Protection protocol

in-order delivery of all messages delivered through a socket, regardless of the state of the socket when the message is sent. Thus, an unprotected message sent shortly after a protected message may be delayed to ensure that it is delivered in the correct order.

The hosted DBMS implements the commit protection protocol using the socket protection mechanism. The commit protocol has two parts, as shown in Figure 5.3. The first part of the protocol runs when a client requests that a transaction commit. The server protects the transaction's socket before starting commit processing, at it remains protected until after sending the commit acknowledgement to the client. All transaction output up until the arrival of the commit request is sent unprotected. The second part of the protocol runs at the standby server after a failover, and causes all active transactions that are not committing to abort. Remus is designed to run a recovery thread in the standby VM as soon as it takes over after a failure. In RemusDB, the recovery thread runs inside the standby DBMS and implements the failover part of the commit protection protocol. Once the recovery thread finishes this work, the DBMS resumes execution from state captured by the most recent pre-failover checkpoint. Note that, on failover, the recovery handler will see transaction states as they were at the time of the last pre-failure checkpoint.

5.4.1 Correctness of Commit Protection

In this section we state more precisely what it means for the commit protection protocol to behave correctly. Essentially, if the client is told that a transaction has committed, then that transaction should remain committed after a failure. Furthermore, if an active transaction has shown speculative results to the client and a failure occurs, then that transaction must ultimately abort. These guarantees are formalized in the following lemmas.

Lemma 1 (Fail-safe Commit)**.** *For each transaction T that is created at the active server prior to the point of failure, if a client receives a commit acknowledgement for T, then T will be committed at the standby site after failover.*

Proof. COMMIT WORK acknowledgements are always sent using a protected socket, which does not release messages to the client until a checkpoint has occurred. If the client has received a commit acknowledgement for T, then a server checkpoint must have occurred after T's commit message was sent and thus after the active server made the commit decision for T. Thus, the active server's commit decision for T (and T's effects) will be captured by the checkpoint and reflected at the standby site after the failure. Furthermore, at the time of the checkpoint, T will either have been committing at the active site or it will have finished. Since the recovery thread at the standby site only aborts active transactions that are not committing, it will not attempt to abort T. □ □

Lemma 2 (Speculation)**.** *For each transaction T that is created at the active server prior to the point of failure, if T's client does not submit a COMMIT WORK request for T prior to the failure, then either T will be aborted at the standby server after the failure, or it will not exist there at all.*

Proof. Let C represent the last checkpoint at the active server prior to its failure. There are three cases to consider. First, T may have started after C. In this case, T will not exist at the time of the checkpoint C, and therefore it will not exist at the standby server after failover. Second, T may have started before C and remained active at C. In this case, some of T's effects may be present at the standby site because they are captured by C. Since the client has not submitted a COMMIT WORK request, T cannot have been committing at the time of C. Therefore, the

	Virtualization Layer	Guest VM Kernel	DBMS
Commit Protection	13	396	103(PostgreSQL), 85(MySQL)
Disk Read Tracking	1903	0	0
Compression	593	0	0

Table 5.1: RemusDB source code modifications (lines of code)

recovery thread will see T as an active, non-committing transaction at failover and will abort T. The third case is that T may have started, aborted, and finished prior to C. In this case, the checkpoint will ensure that T is also aborted at the standby site after failover. □ □

5.4.2 Implementation of Protection and Deprotection

To provide a DBMS with the ability to dynamically switch a client connection between protected and unprotected modes, we added a new `setsockopt()` option to Linux. A DBMS has to be modified to make use of protection and deprotection via the commit protection protocol shown in Figure 5.3. We have implemented commit protection in PostgreSQL and MySQL, with minor modifications to the client connection layer. Because the changes required are for a small and well-defined part of the client/server protocol, we expect them to be easily applied to any DBMS. Table 5.1 provides a summary of the source code changes made to different subsystems to implement the different optimizations that make up RemusDB.

One outstanding issue with commit protection is that while it preserves complete application semantics, it exposes TCP connection state that can be lost on failover: unbuffered packets advance TCP sequence counters that cannot be reversed, which can result in the connection stalling until it times out. In the current implementation of RemusDB we have not addressed this problem: only a small subset of connections are affected, and the transactions occurring over them will be recovered when the connection times out just like any other timed out client connection. In the future, we plan to explore techniques by which we can track sufficient state to explicitly close TCP connections that have become inconsistent at failover time, in order to speed up transaction recovery time for those sessions.

5.5 Reprotection After Failure

When the primary host crashes, the backup host takes over and becomes the new primary. When the original, now-crashed primary host comes back online, it needs to assume the role of backup host. For that to happen, the storage (i.e., disks) of the VMs on the two hosts must be resynchronized. The storage of the VM on the host that was failed and is now back online must catch up with the storage of the VM on the other, now-primary host. After this storage synchronization step, checkpointing traffic can resume between the primary and backup host.

For the protected DBMS to remain available during storage synchronization, this synchronization must happen online, while the DBMS in the primary VM is running. The storage replication driver used by Remus is based on Xen's Blktap2 driver [150] and does not provide a means for online resynchronization of storage. One way to perform the required online resynchronization is to use a brute-force approach and copy all the disk blocks from the primary to the backup. This is sufficient to ensure correctness, but it would impose unnecessary load on the disk subsystem and increase the time to restart HA. A better approach, which we adopt in this chapter, is used by the SecondSite system [83]. In this approach, only the disk blocks changed by the new primary / old backup VM after failure are copied over. The system also overwrites disk blocks written by the old primary VM during the last unfinished checkpoint with data from the backup.

5.6 Experimental Evaluation

In this section we present an evaluation of RemusDB. The objectives of this evaluation are as follows:

- First, we wish to demonstrate that RemusDB is able to survive a failure of the primary server, and to illustrate the performance of RemusDB during and after a failover.

- Second, we wish to characterize the performance overhead associated with RemusDB during normal operation. We compare the performance of unoptimized Remus and optimized RemusDB against that of an unprotected DBMS to measure this overhead. We also consider the impact of specific

RemusDB optimizations on different types of database workloads.

- Third, we consider how key system parameters and characteristics, such as the size of the DBMS buffer pool and the length of the Remus checkpoint interval, affect the overhead introduced by Remus.

5.6.1 Experimental Environment

Our experimental setup consists of two servers each equipped with two quad-core Intel Xeon processors, 16GB RAM, and two 500GB SATA disks. We use the Xen 4.0 hypervisor (64-bit), Debian 5.0 (32-bit) as the host operating system, and Ubuntu 8.04 (32-bit) as the guest operating system. XenLinux Kernel 2.6.18.8 is used for both host and guest operating systems, with disks formatted using the *ext3* filesystem.

We evaluate RemusDB with PostgreSQL 8.4.0 (referred to as Postgres) and MySQL 5.0, using three widely accepted benchmarks namely: TPC-C [174], TPC-H [173], and TPC-W [175]. We run TPC-C experiments on both Postgres and MySQL while TPC-H and TPC-W experiments are run on Postgres only. We use a Remus checkpointing interval (CPI) of 50ms, 100ms, and 250ms for TPC-C, TPC-W, and TPC-H experiments, respectively. These different CPIs for each type of benchmark are chosen because they offer the best trade-off between overhead during normal execution and availability requirements of that particular workload. We evaluate the effect of varying CPI on the TPC-C and TPC-H benchmarks in Section 5.6.6.

Our default settings for TPC-C experiments are as follows: The virtual machine is configured with 2GB memory and 2 virtual CPUs. For MySQL, we use the Percona benchmark kit [168] with a database of 30 warehouses and 300 concurrent clients (10 clients per warehouse). The total size of the database on disk is 3GB. For Postgres, we use the TPCC-UVa benchmark kit [64] with a database of 20 warehouses (1.9GB database on disk) and 200 concurrent clients. We modified the TPCC-UVa benchmark kit so that it uses one TCP connection per client; the original benchmark kit uses one shared connection for all clients. We choose different scales for MySQL and Postgres due to differences in how they scale to larger workloads when provided with fixed (equal) resources. The database buffer pool

DBMS	Benchmark	Performance Metric	Default Scale	Test Duration (mins)	DB Size (GB)	BP Size (MB)	VM Memory (GB)	vCPUs	Remus CPI (ms)
PostgreSQL	TPC-C	TpmC	20W, 200C	30	1.9	190	2	2	50
	TPC-H	Execution Time	1	–	2.3	750	1.5	2	250
	TPC-W	WIPSb	10K Items	20	1.0	256	2	2	100
MySQL	TPC-C	TpmC	30W, 300C	30	3.0	300	2	2	50

Table 5.2: RemusDB experimental settings for evaluation

is configured to be 10% of the database size on disk. We do not use connection pooling or a transaction monitor; each client directly connects to the DBMS.

Our default settings for TPC-H experiments are a virtual machine with 1.5GB memory, 2 virtual CPUs, and a database with TPC-H scale factor 1. The total size of the database on disk is 2.3GB. We configure Postgres with a buffer pool size of 750MB. Our TPC-H experiments consist of one *warmup* run where we execute the 22 read-only TPC-H queries sequentially, followed by one *power stream* run [173] where we execute the queries sequentially and measure the total execution time. We do not perform TPC-H throughput tests or use the refresh streams.

Lastly, for TPC-W experiments we use the TPC-W implementation described in [157]. We use a two tier architecture with Postgres in one tier and three instances of Apache Tomcat v6.0.26 in the second tier, each running in a separate VM. Postgres runs on a virtual machine with 2GB memory, and 2 virtual CPUs. We use a TPC-W database with 10,000 items (1GB on disk). Postgres's buffer pool is configured to be 256MB. Each instance of Apache Tomcat runs in a virtual machine with 1GB memory, and 1 virtual CPU. In these experiments, when running with Remus, only the Postgres VM is protected. In order to avoid the effects of virtual machine scheduling while measuring overhead, we place the Tomcat VMs on a separate well provisioned physical machine.

Table 5.2 provides a summary of our experimental settings. We use the following abbreviations to refer to different RemusDB optimizations in our experiments: RT – Disk Read Tracking, ASC – Asynchronous Checkpoint Compression, and CP – Commit Protection.

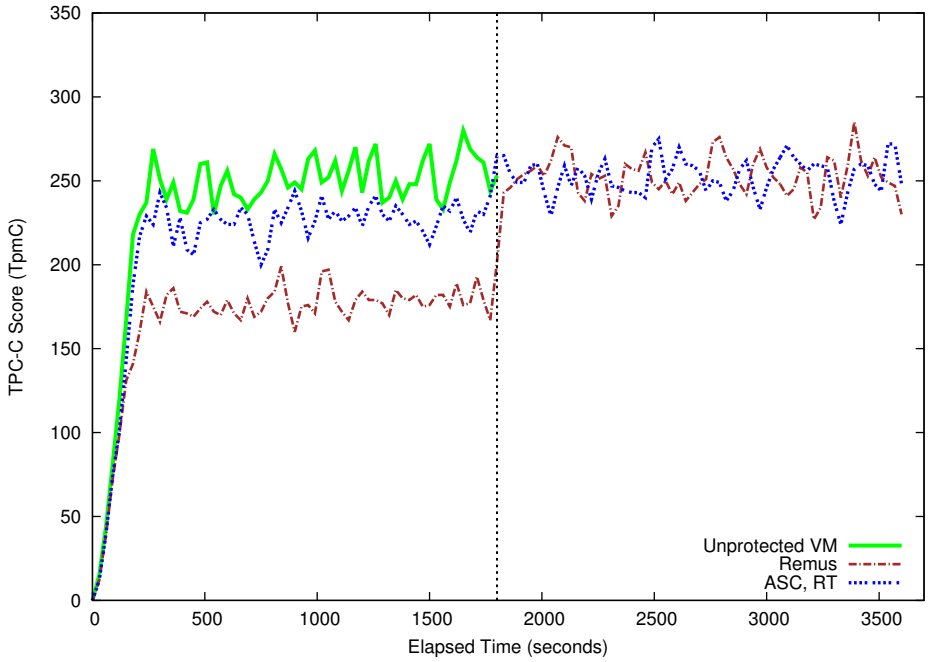


Figure 5.4: TPC-C failover (PostgreSQL)

5.6.2 Behavior of RemusDB During Failover

In the first experiment, we show RemusDB's performance in the presence of failures of the primary host. We run the TPC-C benchmark against Postgres and MySQL and plot throughput in transactions per minute (TpmC). We run the test for 1 hour, a failure of the primary host is simulated at 30 minutes by cutting power to it. We compare the performance of a database system protected by unoptimized Remus and by RemusDB with its two transparent optimizations (ASC, RT) in Figures 5.4 and 5.5. The performance of an unprotected database system (without HA) is also shown for reference. The throughput shown in the figure is the average throughput for a sliding window of 60 seconds. Note that MySQL is run with a higher scale (Table 5.2) than Postgres because of its ability to handle larger workloads when provided with the same resources.

Without any mechanism for high availability in place, the unprotected VM cannot serve clients beyond the failure point i.e., throughput immediately drops to

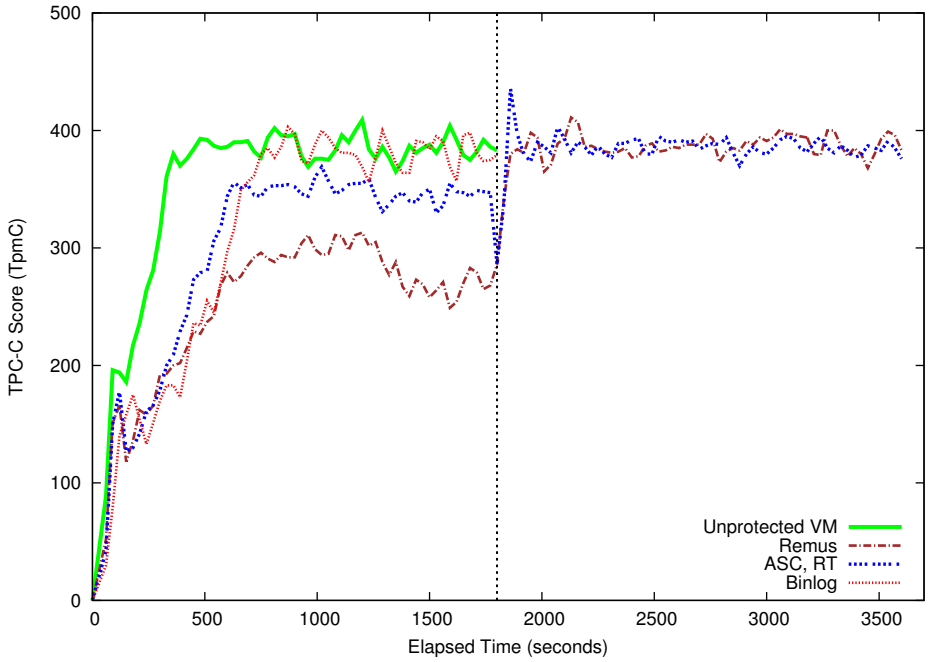


Figure 5.5: TPC-C failover (MySQL)

zero. All clients lose connections to the database server and cannot reconnect until someone (e.g., a DBA) manually restores the database to its pre-failure state. After restart, the database will recover from its crash consistent state using standard log recovery procedures [71]. The time to recover depends on how much state needs to be read from the write-ahead log and reapplied to the database and is usually in the order of several minutes. Furthermore, the unprotected VM will have to go through a warm-up phase again before it can reach its pre-failure steady state throughput (not shown in the graph).

Under both versions of Remus, when the failure happens at the primary physical server, the VM at the backup physical server recovers with ≤ 3 seconds of downtime and continues execution. The database is running with a *warmed up buffer pool, no client connections are lost, and in-flight transactions continue to execute normally* from the last checkpoint. We only lose the speculative execution state generated at the primary server since the last checkpoint. In the worst case,

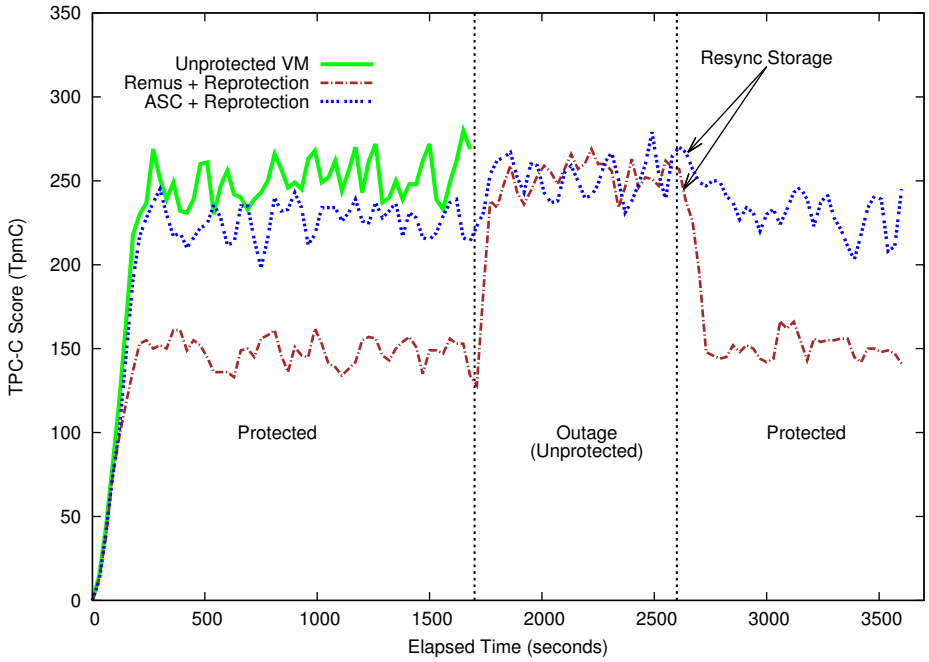


Figure 5.6: TPC-C failover and reprotection after failure (PostgreSQL)

Remus loses one checkpoint interval's worth of work. But this loss of work is completely transparent to the client since Remus (and hence RemusDB) only releases external state at checkpoint boundaries. After the failure, throughput rises sharply and reaches a steady state comparable to that of the unprotected VM before the failure. This is because the VM after the failure is not protected, so we do not incur the replication overhead of Remus.

Figure 5.5 also shows results with MySQL's integrated replication solution, Binlog replication [162, Ch. 5.2.3]. The current stable release of Postgres, used in our experiments, does not provide integrated HA support, although such a facility is in development for Postgres 9. MySQL Binlog replication, in combination with monitoring systems like Heartbeat [182], provides performance very close to that of an unprotected VM and can recover from a failure with ≤ 5 seconds of server downtime. However, we note that RemusDB has certain advantages when compared to Binlog replication, which are discussed in Section 5.7.

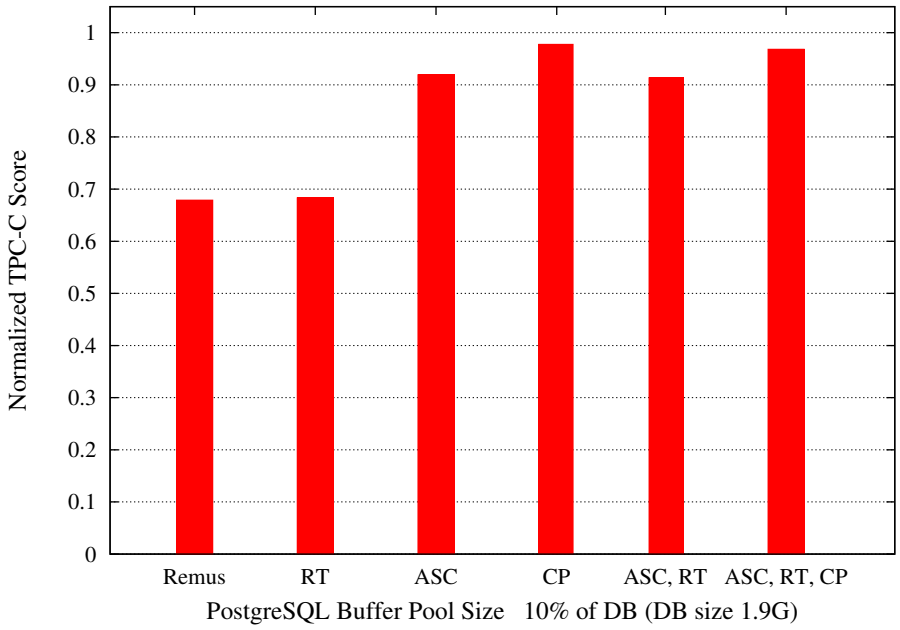


Figure 5.7: TPC-C overhead (PostgreSQL)

5.6.3 Reprotection After a Failure

In Figure 5.6, we show RemusDB's reprotection mechanism in action. Similar to the failover experiment in Section 5.6.2, we run the TPC-C benchmark against Postgres and plot throughput in transactions per minute (TpmC). We run the test for 1 hour, a failure of the primary host is simulated at 30 minutes by cutting power to it. The performance of an unprotected database system (without HA) is also shown for reference. The setting used for these experiments is slightly different than the other experiments in this section. In particular, the storage backend used for reprotection experiments is different since it supports online resynchronization of VM disks after a failure. Because of that, the performance numbers are slightly lower than other experiments, as can be clearly seen from the line for unmodified Remus.

During the outage period, the VM does not incur any checkpointing overhead and hence the throughput rises to that of an unprotected system. After a 15 minute

outage period, the primary host is brought back online. Note that we let the outage last for 15 minutes in this experiment in order to adequately observe performance during an outage. In reality, we can detect a failure within 3 seconds, and resynchronize storage within approximately 29 seconds. The time required to resynchronize can vary depending on the amount of data to be resynchronized. Once storage resynchronization completes, we restart the replication process: all of VM's memory is copied to the backup (the old primary is the new backup), and then Remus checkpoints are resumed. This process takes approximately 10 seconds in our settings for a VM with 2GB of memory over a Gigabit ethernet link. After this point, the VM is once again HA i.e., it is protected against a failure of the backup. Note that after reprotection, the throughput returns back to pre-failure levels as shown in Figure 5.6. For implementing reprotection, we utilized a new storage backend namely DRBD [86] which allows efficient online resynchronization of VM disks. This storage backend does not yet support read tracking. Hence Figure 5.6 only shows RemusDB's performance with the ASC optimization.

5.6.4 Overhead During Normal Operation

Having established the effectiveness of RemusDB at protecting from failure and its fast failover and reprotection times, we now turn our attention to the overhead of RemusDB during normal operation. This section serves two goals: (a) it quantifies the overhead imposed by unoptimized Remus on normal operation for different database benchmarks, and (b) it measures how the RemusDB optimizations affect this overhead, when applied individually or in combination. For this experiment we use the TPC-C, TPC-H, and TPC-W benchmarks.

Figures 5.7 and 5.8 present TPC-C benchmark results for Postgres and MySQL, respectively. In each case the benchmark was run for 30 minutes using the settings presented in Table 5.2. On the x-axis, we have different RemusDB optimizations and on the y-axis we present TpmC scores normalized with respect to an unprotected (base) VM. The normalized score is defined as: (TpmC with optimization being evaluated)/(Base TpmC). The TpmC score reported in these graphs takes into account all transactions during the measurement interval irrespective of their response time requirements. Base VM scores are 243 and 365 TpmC for Post-

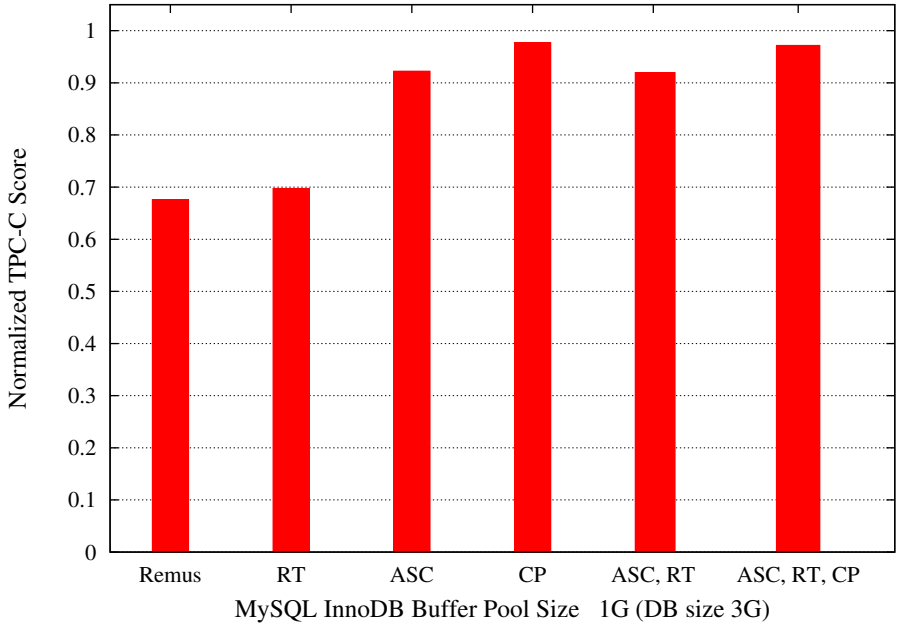


Figure 5.8: TPC-C overhead (MySQL)

gres and MySQL, respectively. The score of unoptimized Remus (leftmost bar) is around 0.68 of the base VM score for both DBMSes – representing a significant performance loss. It is clear from the graph that without optimizations, Remus protection for database systems comes at a very high cost. The next three bars in the graph show the effect of each RemusDB optimization applied individually. RT provides very little performance benefit because TPC-C has a small working set and dirties many of the pages that it reads. However, both ASC and CP provide significant performance gains. Performance with these optimizations is 0.9-0.97 of the base performance. TPC-C is particularly sensitive to network latency and both of these optimizations help reduce latency either by reducing the time it takes to checkpoint (ASC) or by getting rid of the extra latency incurred due to Remus's network buffering for all but commit packets (CP). The rightmost two bars in the graph show the effect of combining optimizations. The combination of all three optimizations (ASC, RT, CP) yields the best performance at the risk of a few trans-

action aborts (not losses) and connection failures. In multiple variations of this experiment we have observed that the variance in performance is always low, and that when the combination of (ASC, RT, CP) does not outright outperform the individual optimizations, the difference is within the range of experimental error. The improvement in performance when adding (ASC, RT, CP) to Remus can be seen not only in throughput, but also in latency. The average latency of the NewOrder transactions whose throughput is plotted in Figures 5.7 and 5.8 for Postgres and MySQL, respectively is 12.9 seconds and 19.2 seconds for unoptimized Remus. This latency is 1.8 seconds and 4.5 seconds for RemusDB. Compare this to the latency for unprotected VM which is 1.2 seconds for Postgres and 3.2 seconds for MySQL. Other experiments (not presented here) show that on average about 10% of the clients lose connectivity after failover when CP is enabled. In most cases, this is an acceptable trade-off given the high performance under (ASC, RT, CP) during normal execution. This is also better than many existing solutions where there is a possibility of losing not only connections but also committed transactions, which never happens in RemusDB.

Figure 5.9 presents the results for TPC-H with Postgres. In this case, the y-axis presents the total execution time of a warmup run and a power test run normalized with respect to the base VM's execution time (921 s). The normalized execution time is defined as: (Base execution time)/(Execution time with optimization being evaluated). Since TPC-H is a decision support benchmark that consists of long running compute and I/O intensive queries typical of a data warehousing environment, it shows very different performance gains with different RemusDB optimizations as compared to TPC-C. In particular, as opposed to TPC-C, we see some performance gains with RT because TPC-H is a read intensive workload, and absolutely no gain with CP because it is insensitive to network latency. A combination of optimizations still provides the best performance, but in case of TPC-H most of the benefits come from memory optimizations (ASC and RT). These transparent memory optimizations bring performance to within 10% of the base case, which is a reasonable performance overhead. Using the non-transparent CP adds no benefit and is therefore not necessary. Moreover, the opportunity for further performance improvement by using the non-transparent memory deprotection interface (presented in Section 5.3.2) is limited to 10%. Therefore, we conclude that it is not

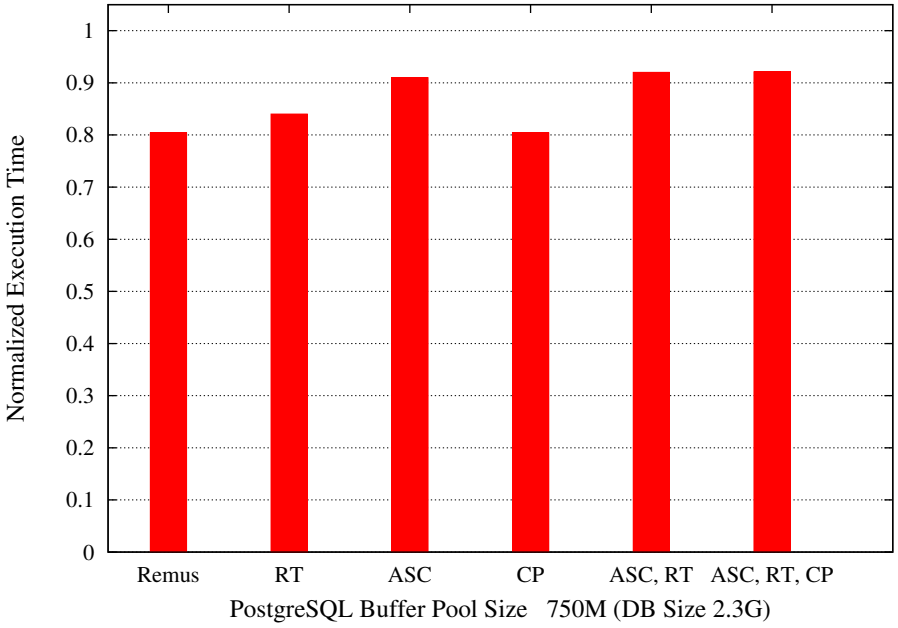


Figure 5.9: TPC-H overhead (PostgreSQL)

worth the additional complexity to pursue it.

Finally, we present the results for TPC-W with Postgres in Figure 5.10. Each test was run with the settings presented in Table 5.2 for a duration of 20 minutes. We drive the load on the database server using 252 Emulated Browsers (EBs) that are equally divided among three instances of Apache Tomcat, which in turn access the database to create dynamic web pages and return them to EBs, as specified by the TPC-W benchmark standard [175]. We use the TPC-W *browsing mix* with image serving turned off at the clients. The y-axis on Figure 5.10 presents TPC-W scores, Web Interactions Per Second (WIPS), normalized to the base VM score (36 WIPS). TPC-W behaves very similar to TPC-C workload: ASC and CP provide the most benefit while RT does not provide any benefit.

RemusDB has a lot to offer for a wide variety of workloads that we study in this experiment. This experiment shows that a combination of memory and network optimizations (ASC and CP) work well for OLTP style workloads, while DSS style

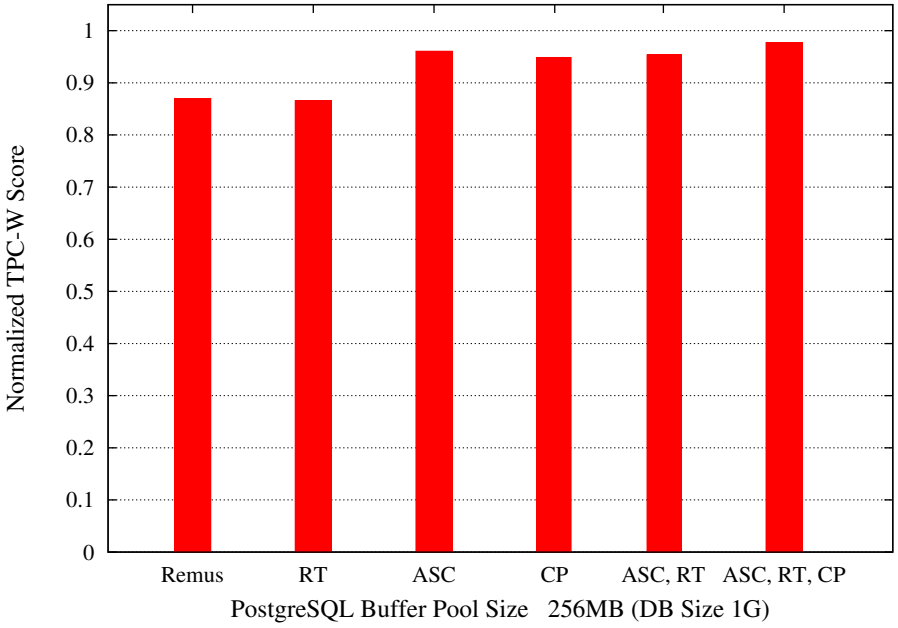


Figure 5.10: TPC-W overhead (PostgreSQL)

workloads gain the most benefit from memory optimizations alone (ASC and RT). It also shows that by using the set of optimizations that we have implemented in RemusDB, we gain back almost all of the performance lost when going from an unprotected VM to a VM protected by unoptimized Remus.

5.6.5 Effects of DB Buffer Pool Size

In the previous experiment, we showed that memory optimizations (ASC and RT) offer significant performance gains for the TPC-H workload. The goal of this experiment is to study the effects of database buffer pool size on different memory optimizations on a micro level. In doing so, we hope to offer insights about how each of these optimization offers its performance benefits.

We run a scale factor 1 TPC-H workload, varying the database buffer pool size from 250MB to 1000MB. We measure the total execution time for the warmup run and the power test run in each case, and repeat this for different RemusDB opti-

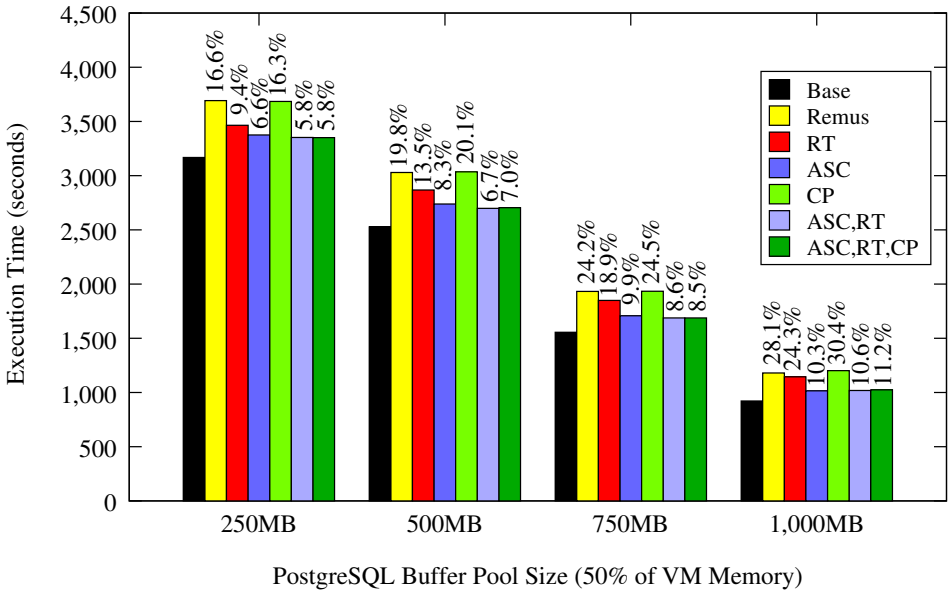


Figure 5.11: Effect of DB buffer pool size on RemusDB (TPC-H)

mizations. To have reasonably realistic settings, we always configure the buffer pool to be 50% of the physical memory available to the VM. For example, for a 250MB buffer pool, we run the experiment in a 500MB VM and so on. Results are presented in Figure 5.11. The numbers on top of each bar show the relative overhead with respect to an unprotected VM for each buffer pool setting. We calculate this overhead as:

$$Overhead\ (\%) = \frac{X-B}{B} \times 100$$

where B is the total execution time for an unprotected VM and X is the total execution time for a protected VM with a specific RemusDB optimization.

Focusing on the results with a 250MB buffer pool in Figure 5.11, we see a 16.6% performance loss with unoptimized Remus. Optimized RemusDB with RT and ASC alone incurs only 9.4% and 6.6% overhead, respectively. The RemusDB memory optimizations (ASC, RT) when applied together result in an overhead of only 5.8%. As noted in the previous experiment, CP does not offer any performance benefit for TPC-H. We see the same trends across all buffer pool sizes. It

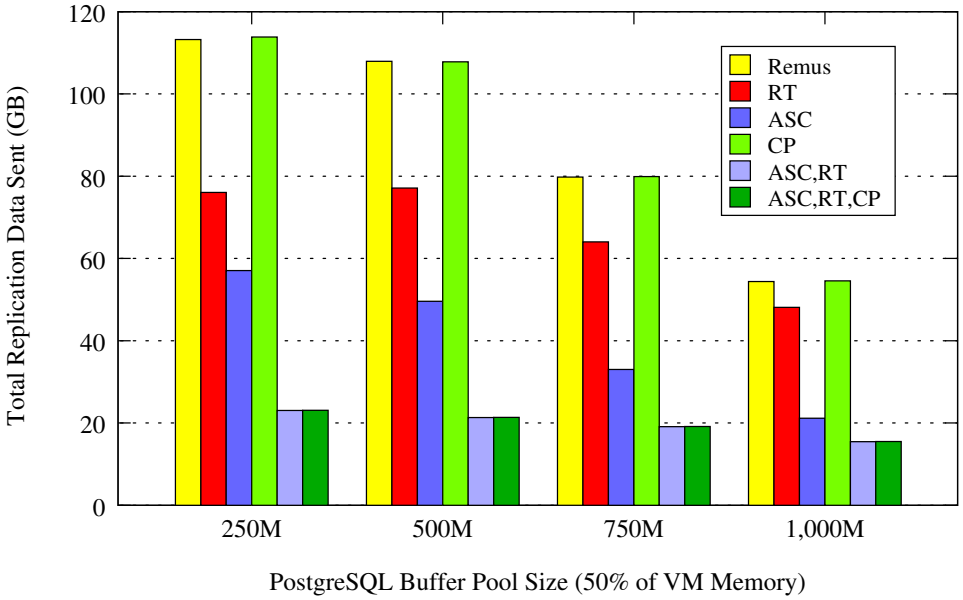


Figure 5.12: Effect of DB buffer pool size on bandwidth (TPC-H)

can also be seen from the graph that the overhead of RemusDB increases with larger buffer pool (and VM memory) sizes. This is because the amount of work done by RemusDB to checkpoint and replicate changes to the backup VM is proportional to the amount of memory dirtied, and there is potential for dirtying more memory with larger buffer pool sizes. However, this overhead is within a reasonable 10% for all cases.

Another insight from Figure 5.11 is that the benefit of RT decreases with increasing buffer pool size. Since the database size is 2.3GB on disk (Table 5.2), with a smaller buffer pool size (250 and 500MB) only a small portion of the database fits in main memory, resulting in a lot of "paging" in the buffer pool. This high rate of paging (frequent disk reads) makes RT more useful. With larger buffer pool sizes, the paging rate decreases drastically and so does the benefit of RT, since the contents of the buffer pool become relatively static. In practice, database sizes are much larger than the buffer pool sizes, and hence a moderate paging rate is common.

In Figure 5.12, we present the total amount of data transferred from the pri-

mary server to the backup server during checkpointing for the entire duration of the experiment. The different bars in Figure 5.12 correspond to the bars in Figure 5.11. With a 250MB buffer pool size, unoptimized Remus sends 113GB of data to the backup host while RemusDB with ASC and RT together sends 23GB, a saving of 90GB (or 80%). As we increase the buffer pool size, the network bandwidth savings for RemusDB also decrease for the same reasons explained above: with increasing buffer pool size the rate of memory dirtying decreases, and so do the benefits of memory optimizations, both in terms of total execution time and network savings. Recall that CP is not concerned with checkpoint size, and hence it has no effect on the amount of data transferred.

5.6.6 Effects of RemusDB Checkpoint Interval

This experiment aims to explore the relationship between RemusDB's checkpoint interval (CPI) and the corresponding performance overhead. We conducted this experiment with TPC-C and TPC-H, which are representatives of two very different classes of workloads. We run each benchmark on Postgres, varying the CPI from 25ms to 500ms. Results are presented in Figures 5.13 and 5.14 for TPC-C and TPC-H, respectively. We vary CPI on the x-axis, and we show on the y-axis TpmC for TPC-C (higher is better) and total execution time for TPC-H (lower is better). The figures show how different CPI values affect RemusDB's performance when running with (ASC, RT) and with (ASC, RT, CP) combined, compared to an unprotected VM.

From the TPC-C results presented in Figure 5.13, we see that for (ASC, RT) TpmC drops significantly with increasing CPI, going from a relative overhead of 10% for 25ms to 84% for 500ms. This is to be expected because, as noted earlier, TPC-C is highly sensitive to network latency. Without RemusDB's network optimization (CP), every packet incurs a delay of $\frac{CPI}{2}$ milliseconds on average. With a benchmark like TPC-C where a lot of packet exchanges happen between clients and the DBMS during a typical benchmark run, this delay per packet results in low throughput and high transaction response times. When run with memory (ASC, RT) and network (CP) optimizations combined, RemusDB's performance is very close to that of unprotected VM, with a relative overhead $\leq 9\%$ for all CPIs.

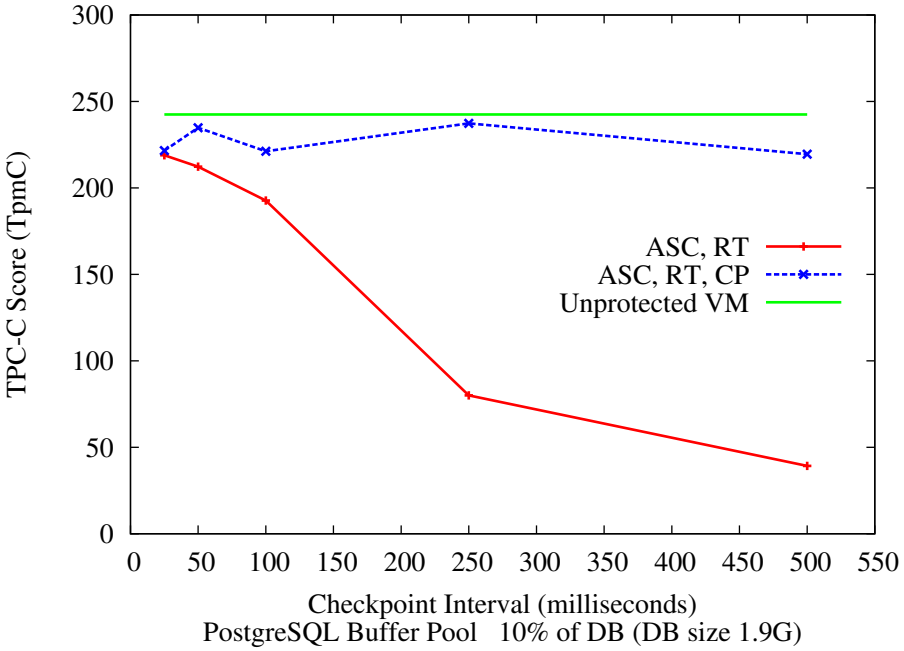


Figure 5.13: Effect of checkpoint interval on RemusDB (TPC-C)

On the other hand, the results of this experiment for TPC-H (Figure 5.14) present a very different story. In contrast to TPC-C, increasing CPI actually leads to reduced execution time for TPC-H. This is because TPC-H is not sensitive to network latency but is sensitive to the overhead of checkpointing, and a longer CPI means fewer checkpoints. The relative overhead goes from 14% for 25ms CPI to 7% for 500ms. We see a similar trend for both (ASC, RT) and (ASC, RT, CP) since CP does not help TPC-H (recall Figure 5.11).

There is an inherent trade-off between RemusDB's CPI, work lost on failure, and performance. Choosing a high CPI results in more lost state after a failover since all state generated during an epoch (between two consecutive checkpoints) will be lost, while choosing a low CPI results in a high runtime overhead during normal execution for certain types of workloads. This experiment shows how RemusDB's optimizations, and in particular the network optimization (CP), helps relax this trade-off for network sensitive workloads. For compute intensive work-

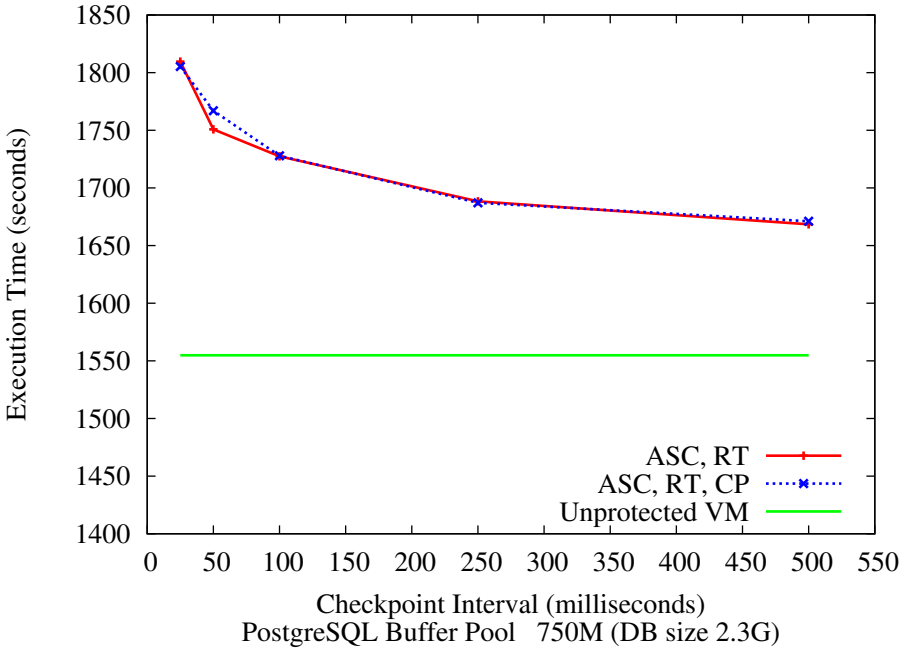


Figure 5.14: Effect of checkpoint interval on RemusDB (TPC-H)

loads that are also insensitive to latency (e.g., TPC-H), choosing a higher CPI actually helps performance.

5.6.7 Effect of Database Size on RemusDB

In the last experiment, we want to show how RemusDB scales with different database sizes. Results for the TPC-C benchmark on Postgres with varying scales are presented in Figure 5.15. We use three different scales: (a) 10 warehouses, 100 clients, 850MB database; (b) 15 warehouses, 150 clients, 1350MB database; and (c) 20 warehouses, 200 clients, 1900MB database. The Postgres buffer pool size is always 10% of the database size. As the size of the database grows, the relative overhead of unoptimized Remus increases considerably, going from 10% for 10 warehouses to 32% for 20 warehouses. RemusDB with memory optimizations (ASC, RT) incurs an overhead of 9%, 10%, and 12% for 10, 15, and 20 warehouses, respectively. RemusDB with memory and network optimizations (ASC,

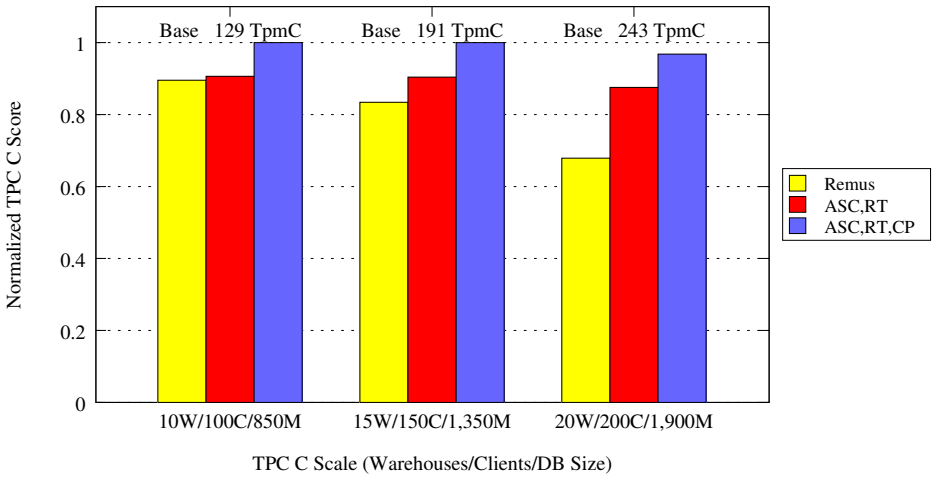


Figure 5.15: Effect of database size on RemusDB (TPC-C)

RT, CP) provides the best performance at all scales, with almost no overhead at the lower scales and only a 3% overhead in the worst case at 20 warehouses.

Results for TPC-H with scale factors 1, 3, and 5 are presented in Figure 5.16. Network optimization (CP) is not included in this figure since it does not benefit TPC-H. Unoptimized Remus incurs an overhead of 22%, 19%, and 18% for scale factor 1, 3, and 5, respectively. On the other hand, RemusDB with memory optimizations has an overhead of 10% for scale factor 1 and an overhead of 6% for both scale factors 3 and 5 – showing much better scalability.

5.7 Related Work

5.7.1 Database HA Techniques

Several types of HA techniques are used in database systems, sometimes in combination. Many database systems [24, 58, 161, 166] implement some form of active-standby HA, which is also the basis of RemusDB. In active-standby systems, update propagation may be synchronous or asynchronous. With synchronous propagation, transaction commits are not acknowledged to the database client until both

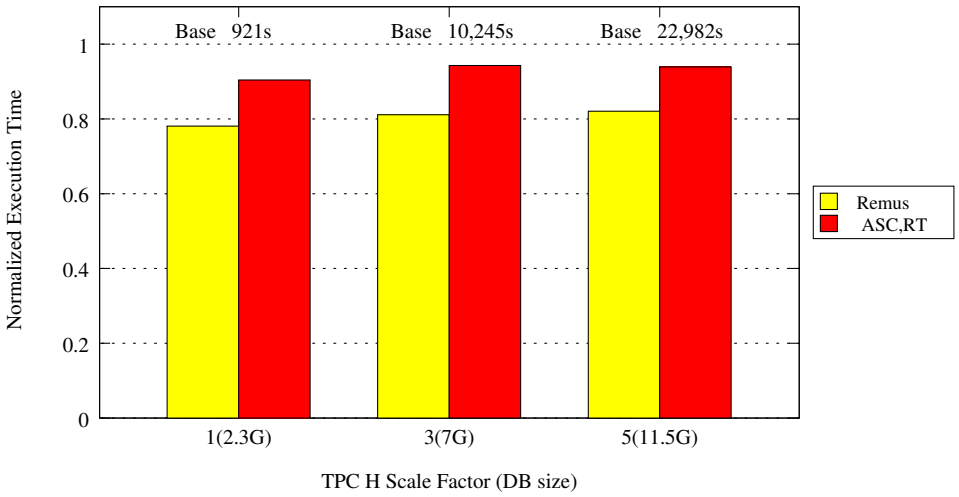


Figure 5.16: Effect of database size on RemusDB (TPC-H)

the primary and backup systems have durably recorded the update, resulting in what is known as a *2-safe* system [46, 81]. A 2-safe system ensures that a single server failure will not result in lost updates, but synchronous update propagation may introduce substantial performance overhead. In contrast, asynchronous propagation allows transactions to be acknowledged as soon they are committed at the primary. Such *1-safe* systems impose much less overhead during normal operation, but some recently-committed (and acknowledged) transactions may be lost if the primary fails. RemusDB, which is itself an active-standby system, uses asynchronous checkpointing to propagate updates to the backup. However, by controlling the release of output from the primary server, RemusDB ensures that committed transactions are not acknowledged to the client until they are recorded at the backup. Thus, RemusDB is 2-safe. RemusDB also differs from other database active-standby systems in that it protects the entire database server state, and not just the database.

MySQL Binlog [159] based master-slave replication, in combination with monitoring systems like Heartbeat [182], provides performance very close to that of an unprotected VM and can recover from a failure with ≤ 5 seconds of server

downtime. However, RemusDB has certain advantages when compared to Binlog replication:

- *Completeness*. On failover, Binlog replication can lose up to one transaction even under the most conservative settings [162, Ch. 16.1.1.1]. In contrast RemusDB never loses transactions.

- *Transparency*. Client-side recovery is more complex with Binlog, which loses all existing client sessions at failure. While the backup database becomes functional immediately, DBMS clients would be likely to take much longer to recover, since they would have to time-out their connections.

- *Implementation complexity*. Binlog accounts for approximately 18K lines of code in MySQL, and is intricately tied to the rest of the DBMS implementation. Not only does this increase the effort required to develop the DBMS (as developers must be cautious of these dependencies), but it also results in constant churn for the Binlog implementation, ultimately making it more fragile. Binlog has experienced bugs proportionate to this complexity: more than 700 bugs were reported over the last 3 years.

Alternatives to active-standby HA include *shared access* and *multi-master* HA. Under the former approach, multiple database systems share access to a single, reliable database image. The latter approach uses multiple database replicas, each handling queries and updates. In shared access approaches, two or more database server instances share a common storage infrastructure, which holds the database. The storage infrastructure stores data redundantly, e.g., by mirroring it on multiple devices, so that it is reliable. In addition, the storage interconnect, through which the servers access the stored data (e.g., a SAN), is made reliable through the use of redundant access pathways. In case of a database server failure, other servers with access to the same database can take over the failed server's workload. Examples of this approach include Oracle RAC [167], which implements a virtual shared buffer pool across server instances, failover clustering in Microsoft SQL Server [58], and synchronized data nodes accessed through the NDB backend API in MySQL Cluster [161]. RemusDB differs from these techniques in that it does not rely on a shared storage infrastructure.

Like active-standby systems, multi-master systems (also known as update anywhere or group systems [45]) achieve high availability through replication. Multi-master systems relax the restriction that all updates must be performed at a single site. Instead, all replicas handle user requests, including updates. Replicas then propagate changes to other replicas, which must order and apply the changes locally. Various techniques, such as those based on quorum consensus [44, 103] or on the availability of an underlying atomic broadcast mechanism [56], can be used to synchronize updates so that global one-copy serializability is achieved across all of the replicas. However, these techniques introduce both performance overhead and complexity. Alternatively, it is possible to give up on serializability and expose inconsistencies to applications. However, these inconsistencies must then somehow be resolved, often by applications or by human administrators. RemusDB is based on the simpler active-standby model, so it does not suffer from problems faced by multi-master systems.

5.7.2 Virtualization based HA Systems

Storage Replication. The established approach for failure/disaster recovery in many virtualized environments today involves the synchronous or asynchronous replication of storage (e.g. SnapMirror [77], PipeCloud [116]). The Recovery Time Objective (RTO) and Recovery Point Objective (RPO) determines the degree of synchrony required and the overhead incurred during normal operation. Generally, some support is expected from the application level in order to restore the disk data to a consistent state.

VM replication. Arbitrary applications can be protected from failures by running them inside a virtual machine and replicating the state of the entire virtual machine. Replication can be achieved either through event logging and execution replay or whole machine checkpointing. While event logging requires much less bandwidth than whole machine checkpointing, it is not guaranteed to be able to reproduce machine state unless execution can be made *deterministic*. Enforcing determinism on commodity hardware requires careful management of sources of non-determinism [16, 42], and becomes infeasibly expensive to enforce on shared-memory multiprocessor systems [5, 37, 131]. Respec [61] does provide determin-

istic execution recording and replay of multi-threaded applications with good performance by lazily increasing the level of synchronization it enforces depending on whether it observes divergence during replay, but it requires intricate modifications to the operating system. It also requires re-execution to be performed on a different core of the same physical system, making it unsuitable for HA applications. For these reasons, the replay-based HA systems support only uni-processor VMs [89]. RemusDB, like Remus [28] uses whole machine checkpointing, so it supports multiprocessor VMs.

5.8 Summary

This chapter presented RemusDB, a system for providing simple transparent DBMS high availability at the virtual machine layer. RemusDB provides active-standby HA and relies on VM checkpointing to propagate state changes from the primary server to the backup server. The system incurs low performance overhead as demonstrated in Section 5.6 by taking advantage of the DBMS' memory access patters (Section 5.3) and its transactional semantics (Section 5.4). RemusDB's approach to high availability opens up new opportunities for DBMS deployments on commodity hardware in the cloud. High availability can be decoupled from the DBMS and offered as a transparent service by the cloud infrastructure on which the DBMS runs.

Chapter 6

Discussion & Future Work

In Chapter 2, we characterized the adaptability properties, namely elasticity and HA, of existing solutions and concluded that applications need to carefully manage both their resources (e.g., VMs) and their runtime state in order to become adaptive. While dynamic resource scaling facilities have been developed, today, it is still the responsibility of the developer to manage state at the application layer as the underlying infrastructure reacts to changing load or failures. Generic application development frameworks [138, 143, 144] address this issue for certain classes of stateless web applications. Several other applications either shoehorn themselves to fit into these frameworks or roll out their own custom solutions.

This dissertation took the position that a domain-specific approach to developing systems abstractions can lead to reusable designs applicable across all applications in a given domain, while simplifying application development. Chapters 3, 4 & 5 described domain-specific system abstractions for two different application domains, namely network middleboxes and relational databases. In each application domain, the approach taken capitalized on the well known properties of the constituting applications. For example, RemusDB (Chapter 5) leveraged the memory access patterns and transactional nature of databases to provide a low overhead hypervisor layer HA solution. FreeFlow (Chapters 3 & 4) exploited the semantics and layout of flow state in middleboxes to enable balanced elasticity and HA. In this chapter, we reflect on our experiences developing adaptability primitives for individual application domains. While meditating on the question of developing

unified system level primitives, we find that the presence of shared state in applications hinders the ability to dynamically scale and forces coarse-grained state partitioning for high availability. The discussion ends with a brief overview of the future research directions in this design space.

6.1 Experiences Developing Adaptability Interfaces

As discussed in Chapter 2, in order to make an internet application adaptive three challenges must be solved:

- **State Partitioning.** Identify the boundaries at which the runtime state of an application can be fragmented into independent partitions that can be concurrently processed.

- **Partition Migration/Replication.** Enable the partitions to become mobile, such that they can be migrated or replicated across nodes in an application cluster without violating consistency of output. When partitions are moved, the network flows related to the partition also need to be moved. Consequently appropriate facilities must be implemented at the network level to dynamically redirect packets to a different application node.

- **Orchestration.** Define sensors that monitor various metrics such as load, liveliness, connectivity, etc. User-defined policies should be in place to act on these sensors to reconfigure the system in case of overload, underload, failures, etc.

Once partitions are identified and made mobile, elastic scaling can be easily accomplished through a combination of dynamic resource scaling (e.g., add/remove VMs) and partition movement across nodes to rebalance the load. High availability can be achieved through replication of partitioned state. The following subsections present various design issues that were encountered during the course of developing interfaces for different applications.

6.1.1 Identifying and Encapsulating Partitions

The size of a partition heavily influences the overhead imposed by the system interfaces on the application's performance. Smaller partitions enable fine-grained elasticity and HA. Larger partitions result in poor load balancing. Larger partitions also take longer time to replicate, thereby increasing the latency experienced by clients connected to the application.[1] In the context of network-facing applications that scale as a function of the number of concurrent client sessions, we find that there are two ways of partitioning the runtime state of a network-facing application: session level and application level.

Session/Flow level. This fine-grained encapsulation model is suitable for applications where the state that needs to be managed is cleanly separable from the rest of the state in the application instance. The application runtime is expected to interact with a state management framework in order to manage the life cycle of the partitionable state. The state management framework is responsible for moving and replicating the partitions while maintaining their consistency. For example, in the case of middleboxes, the scalability of the system depends on the ability to concurrently process as many flows as possible. Each flow is independent of other flows in the system, sharing little or no state with other flows in the application. Consequently, the flow state is cleanly separable from the rest of the system. The Split/Merge framework provided applications with an API based framework through which flow specific state can be created and manipulated in a transactional manner. Transparent to the application, the FreeFlow system (Chapter 3) provided elasticity through flow state migration and Pico Replication (Chapter 4) provided HA through flow-granularity replication.

Application level. This coarse-grained encapsulation model is suitable for systems where the state that needs to be managed is tightly coupled with the rest of the state inside the application instance. These systems typically consist of a large amount of state shared across otherwise independent client sessions. For example, in databases, key-value stores, content management systems, etc., there is a non-negligible amount of shared state in the form of locks, shared memory, data

[1] When replicating state to provide HA, outputs pertaining to a partition need to be buffered until a checkpoint of the partition has been replicated to the backup [28, 69, 84].

on disk, etc. The state of a client session is intricately tied to these global data structures. It is not possible to cleanly isolate and package a client's session state alone such that it can be moved or replicated. Consequently, the fundamental unit of state changes from a single independent session to an entire application instance (i.e., a VM) comprising several sessions and any state shared among them. From a system interface design perspective, only resource scaling techniques like Auto Scaling [148] and live migration [25] can be applied for elasticity. The application is responsible for managing its runtime state. However, these applications can still benefit from system support for HA. RemusDB (Chapter 5) described one such system interface, where the state of the entire virtual machine hosting an application was replicated in a consistent manner. To reduce the HA overhead, RemusDB optimized the replication process by exploiting the memory access patterns and the transactional properties of the application running inside the VM.

Managing non-determinism. Since partitions can be moved or replicated across machines, in order to ensure the consistency of output after a move or failover operation, the partitioned state must encapsulate any source of non-determinism it requires for producing the correct output (e.g., random number seeds). If the application's outputs depend on a non-deterministic input stream that cannot be replicated, such as a processor's timestamp counter, the application cannot be subjected to any system-level adaptability operations.

In summary, the granularity of state encapsulation depends on the amount of shared state and the degree to which individual sessions in the application are dependent on the shared data. The less shared between network sessions in an application, the finer the granularity of partitioning. Fine-grained partitioning is desirable as it enables load balanced elasticity and low overhead high availability.

6.1.2 Routing Flows to Partitions

When partitions are moved, input packets corresponding to the partition also need to be rerouted to the appropriate destination. With the help of software defined networking (SDN) technologies such as OpenFlow the packet forwarding behavior of switching hardware can be manipulated at runtime in a programmatic fashion. A network flow between two endpoints is typically represented by the IP 5-tuple

($< srcip, srcport, dstip, dstport, proto >$). The orchestration system which is responsible for invoking the adaptability operations installs flow forwarding rules at the switches such that the network flows belonging to a client session are routed to the appropriate application node.

When an application is finely partitioned, such as session-level partitioning, there is an unique network flow representing each client session. Ideally, a flow forwarding rule needs to be installed per session so that when the session is moved the forwarding rule associated with that particular session alone can be updated. Unfortunately, current switching hardware is capable of handling only a limited number of flows.[2] Fortunately, with the help of wildcard-based pattern matching [114], multiple flows can be matched using a single rule. For example, if all the flows from the 172.16.1.0/24 subnet are handled by the same application node, then instead of having individual rules to match each session from that subnet, a single wildcard rule can be installed that redirects all sessions from that subnet to the specified application node. The wildcard-based rule aggregation technique minimizes the number of rules installed on the switch, while saving the remainder of the space for special case scenarios. When the application node is overloaded, one or more sessions can be migrated to other nodes. Higher priority rules without wildcards can be installed on the switch enabling the chosen sessions to be redirected to a different application node, while the rest of the sessions continue to be routed to the previous node. Despite the wildcard based rule aggregation technique, today's typical switching hardware cannot handle more than a few hundred rules that match on all fields in a network packet. In the long run, based on the current hardware trends in the data center networking equipment, we expect the rule space limitation to last only for a short term.

6.1.3 Managing Shared State

Shared state in applications is not always an artifact of bad design. Sometimes, it is part of the very DNA of the application. For example, a customer relationship

[2]Flow matching rules are built out of Ternary Content Addressable Memories (TCAM) that are part of the switching hardware. Due to their high power consumption and large silicon footprint, today's OpenFlow-enabled switches typically support a very limited number of TCAM entries (750-1500) [30].

management (CRM) application fetches data pertaining to a customer from its data store, which is shared among all customers using the application. A media streaming server accesses serves the same video content to hundreds of clients. In these systems, the shared state cannot be simply eliminated through application redesign. The shared state can however be delegated to other systems that are specifically designed to manage, scale and protect this type of data [2, 23, 34, 136, 142, 170]. For example, in modern 3-tier applications the web frontend and the application tier deal with session data, while the database backend deals with shared data. Adaptivity properties can now be managed in a modular fashion. By offloading the shared state management to a separate backend component, consumer-facing components (web servers, application servers, etc.) can be made adaptive with much less effort. Section 6.2 describes potential approaches to restructure monolithic systems such as web servers to become dynamically partitionable, similar to middlebox applications.

If the shared state in the application is not monolithic, but is itself partitionable, the application can be partitioned either finely (per-session) or coarsely (partitioned shared state). For example, consider a white board application, where multiple users share a single white board. The application can be viewed as a collection of white boards or as a collection of user sessions. For performance, the application can be partitioned at "white board" granularity, with the ability to fallback to session-level partitioning should any single white board session become overloaded. However, the partitioning becomes complicated when users participate in multiple white board sessions simultaneously. While the framework proposed in this thesis can handle switching from one partitioning granularity to another, it does not provide support for multi-dimensional partitioning. This is an avenue for further exploration.

6.2 Future Work

The Split/Merge design paradigm enables middleboxes to achieve load balanced elasticity and low overhead high availability through fine-grained flow state management. Let us look at how the Split/Merge design paradigm can be extended to other network-facing applications. The principal philosophy of the Split/Merge

design paradigm is that the application or the VM is not the correct granularity to consider either elasticity or availability: network-facing web applications are commonly structured as client-oriented sessions with the (per-client) session state carefully coupled to properties such as load balancing, data consistency, and failure recovery. Consequently, client-oriented sessions form the correct unit for implementing elasticity and high availability. Unfortunately, per-client session state is not confined to the application layer alone. Software is typically modular; its functionality being typically spread across one or more layers. Consequently, per-client session state is also spread across several layers of the software stack.

As a running example, consider a simple Apache/PHP based web application deployed on a cluster of virtual machines (VMs) where the following assumptions hold true: (a) the VMs have identical operating environments (kernel, process binaries, etc.), (b) the file system is shared among the VMs (e.g., NFS), and (c) the cluster is deployed over a Software Defined Network (SDN), such as OpenFlow [165]. As shown in Figure 6.1, a request-response transaction creates/modifies state in the OpenFlow network, Linux kernel, Apache web server and the PHP application.

Consider a vertical abstraction, called a Slice as depicted in Figure 6.2. Similar to Banga's Resource Containers [10], a Slice is a collection of related states associated with a client session that spans all layers of the software stack. At each layer, the Slice is made up of one or more *capsules*, where a capsule is the essential state associated with the corresponding client session. For example, in the OS kernel in Figure 6.1, the socket and TCP state associated with each client may be identified as a capsule. Each layer explicitly expresses dependencies between its capsules and the capsules in the lower layer. Capsules from each layer are chained together explicitly, to make a Slice. For the example in Figure 6.1, each Slice contains the bare minimum state—across all layers—associated with a given client session.

Generalizing the Split/Merge abstraction, let us classify the state in each layer of a VM as (a) belonging to a Slice, (b) global state shared among all Slices, or (c) other. We can view the application cluster as a set of Slices, and not as a set of VMs. A Slice can be migrated from one VM to another or replicated between VMs; this is coordinated by a logically centralized Orchestrator with a global view of Slices. A Slice forms the fundamental unit of state in the web application, similar to a network flow state in a middlebox. We can now apply techniques developed

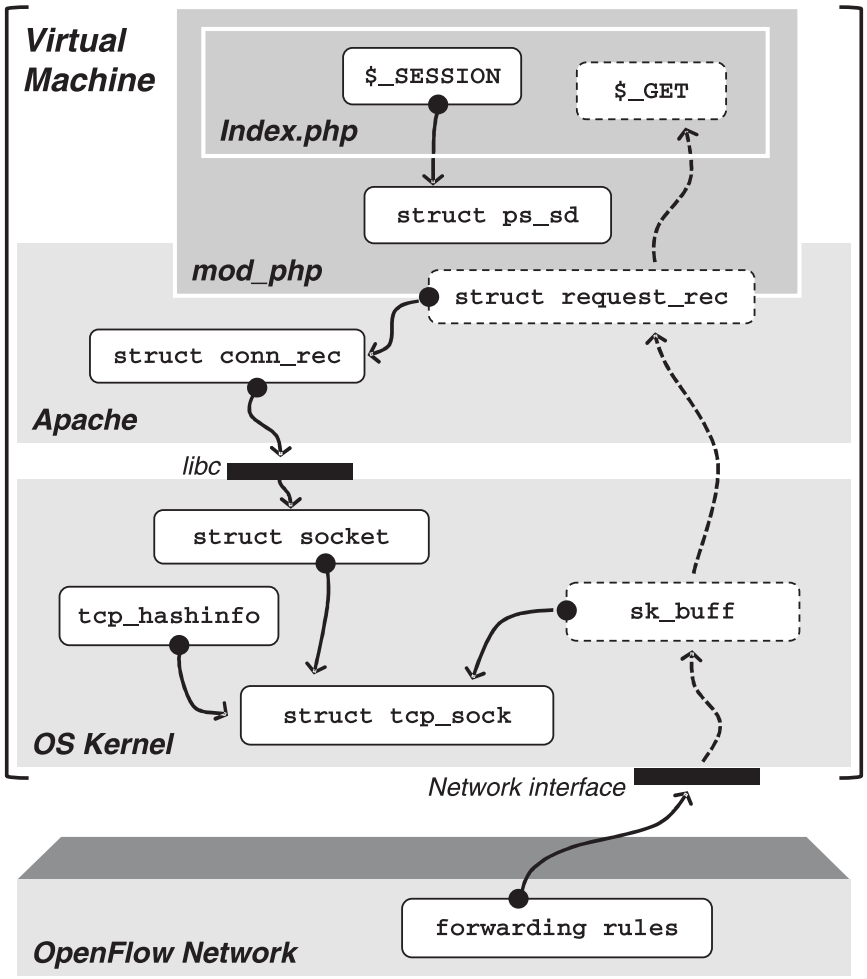


Figure 6.1: Footprints of a client session, in Apache/PHP. Solid boxes denote session-wide state, while dotted boxes denote per-request (ephemeral) state. Solid arrows indicate dependencies while dotted arrows indicate logical control flow.

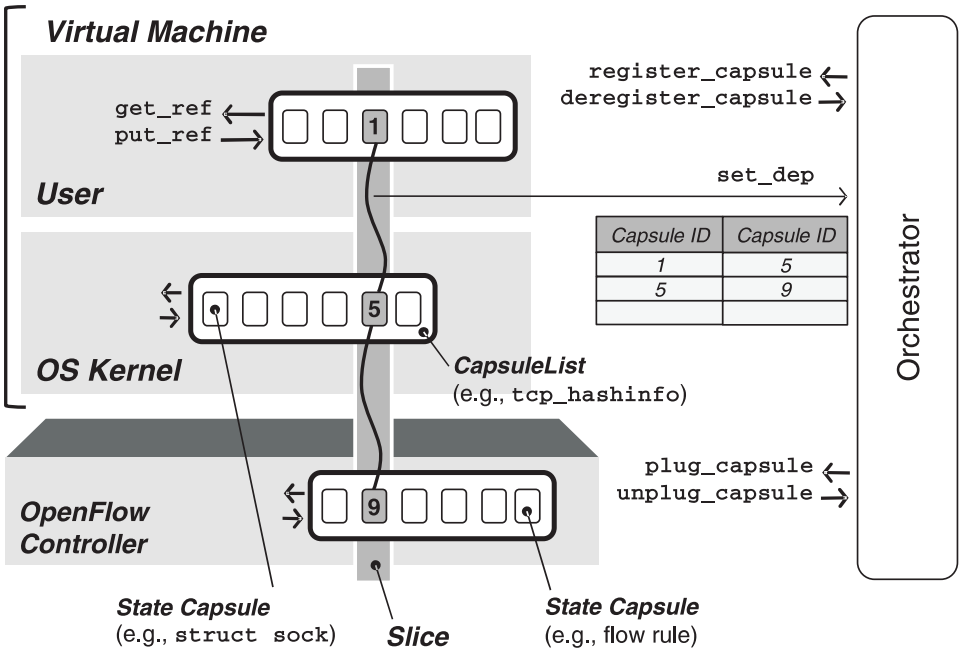


Figure 6.2: Explicitly defined state capsules at each layer linked together to form a Slice. Every layer exposes a common set of APIs allowing a logically centralized entity to track dependencies between capsules within a Slice and migrate or replicate it across VMs.

for middleboxes (Chapters 3 & 4) to achieve dynamic scale and graceful failure recovery for the application as well.

Existing work on application scalability has mostly focused on managing state in the application layer. The system and network level state maintains affinity to the local VM, and restricts the mobility of sessions. As a consequence, sessions turn *sticky*, creating infrastructure hotspots and exposing failures to external clients due to loss of network stack state. In contrast, the Slice abstraction attempts to capture session related state from all parts of the software stack.

Pitfalls. Depending on the structure of the application, Slice migration may face some of the same challenges as process migration [11, 35, 63, 74, 82, 98]. The main challenge in process migration is migrating residual dependencies outside process scope (e.g., shared state between processes, open network connections and files).

For example, a kernel object on the source—referenced by a descriptor—may not exist on the destination. Process migration systems attempted to automatically identify and manage such dependencies using coarse-grained abstractions such as process domains in Zap [59, 74]. The Slice abstraction, on the other hand, expects the developer of each software layer to annotate such dependencies using the Capsule abstraction, with the caveat that shared state may not scale or may even make it impossible to partition the application into Slices.

Summary. Lack of system support for managing session-related state inside the OS and network has led to unnecessary design complexity at the application level. This section presented Slice, a vertical abstraction that connects session related state across all software layers in the system into a single conjoined entity. Slices can be live migrated and replicated across application instances. This creates new opportunities for jointly achieving elasticity and high availability, while allowing a simple and clean design of end applications.

6.3 Concluding Thoughts

Circling back to the question we raised in Chapter 2:

Is it possible to define a set of system primitives that can be used to build adaptive applications of all types?

Based on our experiences developing system interfaces, this dissertation concludes that it is not possible to develop a unified set of primitives that addresses the needs of different applications, even though the fundamental technique to accomplish adaptability is the same. As described in Section 6.1, shared state in the application hinders partitioning and replication, preventing system abstractions from dynamically scaling and replicating fine-grained pieces of the application state. If applications contain shared data that need to be kept synchronized by the system across the application cluster, the synchronization overhead becomes prohibitively high. Network middleboxes, for example, contain very little synchronized state, which enables the Split/Merge system to dynamically partition and replicate the middlebox state finely along flow boundaries. Database systems have a large amount of shared state (e.g., locks, buffer pools, etc) in addition to containing per-client session state. Consequently, the onus of scalability falls on the administrator who has to carefully shard the database schema [160, 167] across the cluster. System level HA can still be achieved through coarse grained replication of the state of an entire VM, as demonstrated by RemusDB.

While unified system support for elasticity and HA is not possible, there is a lot of value in providing domain-specific abstractions. For example, like middleboxes, applications in domains such as e-commerce, streaming media, session-oriented web servers, etc., need to dynamically scale as a function of the number of concurrent client sessions being serviced by the application. As described in Section 6.1.3, if the shared state in such applications can be delegated to dedicated state management systems, these applications can be restructured to effectively resemble middleboxes, such that techniques from Split/Merge can be applied to make them adaptive. Section 6.2 briefly discusses ways to design abstractions for these applications, based on the Split/Merge paradigm.

On the other hand, unlike middleboxes, applications that scale as a function of data, such as key-value stores [2, 23, 34, 73, 142], databases, cloud storage services

like Dropbox, Amazon S3, etc., are highly specialized in terms of their consistency requirements and structure of stateful data. Techniques for scaling, maintaining availability and consistency are typically the distinguishing factors between individual applications. In these systems, application-specific techniques for data partitioning and replication are required to manage data at large scale.

In terms of further research on system support for adaptability, the Slice approach described in Section 6.2 represents one potential solution for session-oriented web applications. As illustrated by the white board example in Section 6.1.3, there is substantial research to be done on understanding and dealing with the shared state in applications. This thesis has taken the initial step towards making developers think explicitly about shared state management while developing their applications for the cloud.

Chapter 7

Conclusion

Netflix weathered the April 2011 Amazon AWS outage with very little impact on its business [26]. Some key aspects of its cloud architecture include completely stateless services and NoSQL based datastores that sacrifice consistency for availability and durability. Building scalable stateless services like Amazon, Netflix, etc., is *hard* and it requires an engineering skill that is generally not affordable by small and medium businesses. Application programmers often use off-the-shelf solutions to rapidly develop and deploy web applications that often end up being stateful. The techniques described in this thesis are a good fit for such applications, as they reduce the burden on the developer by offloading the task of dynamically scaling and preserving the application state to the underlying system. The next step in this general direction is to look at ways of extending the Split/Merge abstraction to applications structured as a collection of independent client-oriented sessions such as web servers, streaming servers, etc. The scalability of such applications depends entirely on the amount of runtime state shared between sessions. In an attempt to eliminate shared state between individual sessions, the abstractions proposed in this thesis force the developer to consciously create and access shared state.

Bibliography

[1] AGUILERA, M. K., CHEN, W., AND TOUEG, S. Using the Heartbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks. *Theoretical Computer Science 220* (1999). → pages 61

[2] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (2007). → pages 8, 119, 124

[3] AL-QUDAH, Z., RABINOVICH, M., AND ALLMAN, M. Web Timeouts and Their Implications. In *Proc. of International Conference on Passive and Active Measurement* (2010). → pages 50

[4] ALLMAN, M. On the Performance of Middleboxes. In *Proc. of Internet Measurement Conference* (2003). → pages 50

[5] ALTEKAR, G., AND STOICA, I. ODR: Output-Deterministic Replay for Multicore Debugging. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (2009). → pages 12, 13, 112

[6] AMZA, C., COX, A. L., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R., YU, W., AND ZWAENEPOEL, W. Treadmarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer 29*, 2 (1996). → pages 16

[7] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. FAWN: A Fast Array of Wimpy Nodes. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (2009). → pages 66

[8] ANDERSON, J. W., BRAUD, R., KAPOOR, R., PORTER, G., AND VAHDAT, A. xOMB: Extensible Open Middleboxes with Commodity Servers.

In *Proc. of Symposium on Architectures for Networking and Communications Systems (ANCS)* (2012). → pages 2

[9] BAKER, M., AND SULLIVAN, M. The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment. In *Proc. of USENIX Summer Conference* (1992). → pages 86

[10] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proc. of USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (1999). → pages 120

[11] BARAK, A. The mosix multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems 13* (1998). → pages 8, 122

[12] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (2003). → pages 28, 68

[13] BENNETT, J. K., CARTER, J. B., AND ZWAENEPOEL, W. Munin: Distributed Shared Memory Based on Type-specific Memory Coherence. In *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)* (1990). → pages 16

[14] BOYD, T., AND DASGUPTA, P. Process migration: A generalized approach using a virtualizing operating system. In *ICDCS '02: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)* (Washington, DC, USA, 2002), IEEE Computer Society. → pages 8

[15] BRADFORD, R., KOTSOVINOS, E., FELDMANN, A., AND SCHIÖBERG, H. Live Wide-Area Migration of Virtual Machines Including Local Persistent State. In *Proc. of ACM Conference on Virtual Execution Environments VEE* (2007). → pages 8

[16] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based Fault-tolerance. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (1995). → pages 12, 13, 54, 83, 112

[17] BRYANT, R., TUMANOV, A., IRZAK, O., SCANNELL, A., JOSHI, K., HILTUNEN, M., LAGAR-CAVILLA, A., AND DE LARA, E. Kaleidoscope:

Cloud Micro-elasticity via VM State Coloring. In *Proc. of ACM European Conference on Computer Systems (EuroSys)* (2011). → pages 8

[18] BURROWS, M. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proc. of USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2006). → pages 30

[19] CARPENTER, B., AND BRIM, S. Middleboxes: Taxonomy and Issues. RFC 3234, `https://tools.ietf.org/rfc/rfc3234.txt`, 2002. → pages 20

[20] CARRIER, N., AND GELERNTER, D. Linda and Friends. *Distributed Shared Memory: Concepts and Systems 21* (1998), 177. → pages 16

[21] CECCHET, E., MARGUERITE, J., AND ZWAENEPOLE, W. C-JDBC: Flexible Database Clustering Middleware. In *Proc. of USENIX Annual Technical Conference (ATC)* (2004). → pages 9, 12, 14

[22] CHANDRA, R., ZELDOVICH, N., SAPUNTZAKIS, C., AND LAM, M. S. The Collective: A Cache-Based System Management Architecture. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)* (2005). → pages 24

[23] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A Distributed Storage System for Structured Data. In *Proc. of USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2006). → pages 8, 9, 12, 13, 119, 124

[24] CHEN, W.-J., OTSUKI, M., DESCOVICH, P., ARUMUGGHARAJ, S., KUBO, T., AND BI, Y. J. High Availability and Disaster Recovery Options for DB2 on Linux, Unix, and Windows. Tech. Rep. IBM Redbook SG24-7363-01, IBM, 2009. → pages 3, 109

[25] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live Migration of Virtual Machines. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)* (2005). → pages 3, 8, 47, 79, 85, 117

[26] COCKROFT, A., HICKS, C., AND ORZELL, G. Lessons Netflix Learned from the AWS Outage. `http://techblog.netflix.com/2011/04/lessons-netflix-learned-from-aws-outage.html`, April 2011. → pages 126

[27] CULLY, B. Generalized High Availability via Virtual Machine Replication. Master's thesis, University of British Columbia, Vancouver, 2007. → pages 12

[28] CULLY, B., LEFEBVRE, G., MEYER, D., FEELEY, M., HUTCHINSON, N., AND WARFIELD, A. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)* (2008). → pages 12, 54, 67, 70, 77, 79, 80, 113, 116

[29] CULLY, B., AND WARFIELD, A. Secondsite: Disaster Protection for the Common Server. In *Proc. of the Conference on Hot Topics in System Dependability (HotDep)* (2006). → pages 12

[30] CURTIS, A. R., MOGUL, J. C., TOURRILHES, J., YALAGANDULA, P., SHARMA, P., AND BANERJEE, S. DevoFlow: Scaling Flow Management for High-performance Networks. In *Proc. of ACM SIGCOMM* (2011). → pages 118

[31] DAS, S., AGRAWAL, D., AND EL ABBADI, A. ElasTraS: An Elastic Transactional Data Store in the Cloud. In *Proc. of the Conference on Hot Topics in Cloud Computing (HotCloud)* (2009). → pages 8

[32] DASGUPTA, P., LEBLANC, R. J., AHAMAD, M., AND RAMACHANDRAN, U. The Clouds Distributed Operating System. *IEEE Computer 24*, 11 (1991). → pages 16

[33] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Communications of ACM 51*, 1 (2008). → pages 30

[34] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's Highly Available Key-Value Store. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (2007). → pages 8, 9, 12, 13, 66, 119, 124

[35] DOUGLIS, F., AND OUSTERHOUT, J. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software: Practice and Experience 21*, 8 (1991). → pages 8, 122

[36] DRAGO, I., MELLIA, M., M. MUNAFO, M., SPEROTTO, A., SADRE, R., AND PRAS, A. Inside Dropbox: Understanding Personal Cloud Storage Services. In *Proc. of Internet Measurement Conference (IMC)* (2012). → pages 7

[37] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M. A., AND CHEN, P. M. Execution Replay of Multiprocessor Virtual Machines. In *Proc. of ACM Conference on Virtual Execution Environments VEE* (2008). → pages 12, 13, 54, 112

[38] FEELEY, M. J., MORGAN, W. E., PIGHIN, E. P., KARLIN, A. R., LEVY, H. M., AND THEKKATH, C. A. Implementing Global Memory Management in a Workstation Cluster. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (1995). → pages 16

[39] FLEISCH, B., AND POPEK, G. Mirage: A Coherent Distributed Shared Memory Design. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (1989). → pages 16

[40] GALANTE, G., AND BONA, L. C. E. D. A Survey on Cloud Computing Elasticity. In *Proc. of the IEEE/ACM International Conference on Utility and Cloud Computing* (2012). → pages 2, 14

[41] GEMBER, A., GRANDL, R., KHALID, J., SHEN, S.-H., AND AKELLA, A. Design and Implementation of a Framework for Software-Defined Middlebox Networking. Tech. Rep. TR1794, University of Wisconsin-Madison, 2013. → pages 2

[42] GEORGE W. DUNLAP AND SAMUEL T. KING AND SUKRU CINAR AND MURTAZA A. BASRAI AND PETER M. CHEN. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proc. of USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2002). → pages 12, 13, 54, 112

[43] GIFFORD, D. K. Weighted Voting for Replicated Data. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (1979). → pages 61

[44] GIFFORD, D. K. Weighted Voting for Replicated Data. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (1979). → pages 112

[45] GRAY, J., HELLAND, P., O'NEIL, P., AND SHASHA, D. The Dangers of Replication and a Solution. In *Proc. of ACM SIGMOD* (1996). → pages 112

[46] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. → pages 110

[47] GREENHALGH, A., HUICI, F., HOERDT, M., PAPADIMITRIOU, P., HANDLEY, M., AND MATHY, L. Flow Processing and the Rise of Commodity Network Hardware. *ACM SIGCOMM Computer Communications Review 39*, 2 (2009). → pages 47

[48] GU, Y., SHORE, M., AND SIVAKUMAR, S. A Framework and Problem Statement for Flow-associated Middlebox State Migration. `http://tools.ietf.org/html/draft-gu-statemigration-framework-02`, 2012. → pages 20

[49] GUDE, N., KOPONEN, T., PETTIT, J., PFAFF, B., CASADO, M., MCKEOWN, N., AND SHENKER, S. NOX: Towards an Operating System for Networks. *ACM SIGCOMM Computer Communications Review 38*, 3 (2008). → pages 32

[50] HARNEY, E., GOASGUEN, S., MARTIN, J., MURPHY, M., AND WESTALL, M. The Efficacy of Live Virtual Machine Migrations Over the Internet. In *Proc. of International Workshop on Virtualization Technology in Distributed Computing (VTDC)* (2007). → pages 8

[51] HERBST, N. R., KOUNEV, S., AND REUSSNER, R. Elasticity in Cloud Computing: What it is, and What it is Not. In *Proc. of the International Conference on Autonomic Computing (ICAC)* (2013). → pages 6

[52] HIROFUCHI, T., NAKADA, H., OGAWA, H., ITOH, S., AND SEKIGUCHI, S. A Live Storage Migration Mechanism Over WAN and its Performance Evaluation. In *Proc. of International Workshop on Virtualization Technologies in Distributed Computing (VTDC)* (2009). → pages 8

[53] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proc. of USENIX Annual Technical Conference (ATC)* (2010). → pages 30

[54] JOSEPH, D. A., AND STOICA, I. Modeling Middleboxes. *IEEE Network 22*, 5 (2008). → pages 20, 45

[55] JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. Fine-grained Mobility in the Emerald System. *ACM Transactions on Computer Systems 6*, 1 (1988). → pages 16

[56] KEMME, B., AND ALONSO, G. A New Approach to Developing and Implementing Eager Database Replication Protocols. *Transactions on Database Systems (TODS) 25*, 3 (2000). → pages 112

[57] KNAUTH, T., AND FETZER, C. Scaling non-elastic Applications Using Virtual Machines. In *IEEE International Conference on Cloud Computing* (2011). → pages 8, 47

[58] KOMO, D. Microsoft SQL Server 2008 R2 High Availability Technologies – White Paper, 2010. → pages 14, 109, 111

[59] LAADAN, O., AND NIEH, J. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *Proc. of USENIX Annual Technical Conference (ATC)* (2007). → pages 13, 123

[60] LAGAR-CAVILLA, H. A., WHITNEY, J. A., SCANNELL, A. M., PATCHIN, P., RUMBLE, S. M., DE LARA, E., BRUDNO, M., AND SATYANARAYANAN, M. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proc. of ACM European Conference on Computer Systems (EuroSys)* (2009). → pages 1, 8, 9, 35, 47

[61] LEE, D., WESTER, B., VEERARAGHAVAN, K., NARAYANASAMY, S., CHEN, P. M., AND FLINN, J. ReSpec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. In *Proc. of ASPLOS* (2010). → pages 12, 54, 112

[62] LING, B. C., KICIMAN, E., AND FOX, A. Session state: Beyond soft state. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)* (2004). → pages 7, 50

[63] LITZKOW, M., AND LIVNY, M. Supporting Checkpointing and Process Migration Outside the UNIX Kernel. In *Proc. of USENIX Conference* (1992). → pages 8, 122

[64] LLANOS, D. R. TPCC-UVa: An Open-Source TPC-C Implementation for Global Performance Measurement of Computer Systems. *SIGMOD Record 35*, 4 (2006). → pages 93

[65] M. BAGNULO, P. MATTHEWS, I. V. B. Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers. `http://tools.ietf.org/id/draft-ietf-behave-v6v4-xlate-stateful-12.txt`, 2010. → pages 22

[66] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A System for Large-Scale Graph Processing. In *Proc. of ACM SIGMOD* (2010). → pages 30

[67] MILLER, R. Car Crash Triggers Amazon Power Outage. *datacenterknowledge.com* (May 2010). → pages 5

[68] MILOJICIC, D. S., DOUGLIS, F., PAINDAVEINE, Y., WHEELER, R., AND ZHOU, S. Process Migration. *ACM Computing Surveys 33* (2000). → pages 8

[69] MINHAS, U. F., RAJAGOPALAN, S., CULLY, B., ABOULNAGA, A., SALEM, K., AND WARFIELD, A. RemusDB: Transparent High Availability for Database Systems. *PVLDB 4*, 11 (2011). → pages 17, 54, 67, 70, 77, 116

[70] MINHAS, U. F., RAJAGOPALAN, S., CULLY, B., ABOULNAGA, A., SALEM, K., AND WARFIELD, A. RemusDB: Transparent High Availability for Database Systems. *The VLDB Journal 22*, 1 (2013). → pages 77

[71] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Transactions on Database Systems 17*, 1 (1992). → pages 96

[72] OLTEANU, V. A., AND RAICIU, C. Efficiently Migrating Stateful Middleboxes. In *ACM SIGCOMM - Demo* (2012). → pages 47

[73] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast Crash Recovery in RAMCloud. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (2011). → pages 12, 13, 124

[74] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proc. of USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2002). → pages 8, 122, 123

[75] OUSTERHOUT, J. K., CHERENSON, A. R., DOUGLIS, F., NELSON, M. N., AND WELCH, B. B. The Sprite Network Operating System. *IEEE Computer 21*, 2 (1988). → pages 86

[76] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (2009). → pages 12

[77] PATTERSON, R. H., MANLEY, S., FEDERWISCH, M., HITZ, D., KLEIMAN, S., AND OWARA, S. SnapMirror: File-System-Based Asynchronous Mirroring for Disaster Recovery. In *Proc. of USENIX Conference on File and Storage Technologies (FAST)* (2002). → pages 11, 12, 14, 112

[78] PAXSON, V. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks 31*, 23-24 (1999). → pages 19, 21, 47, 50, 59, 67

[79] PENG, G. A Distributed Snapshot Protocol for Virtual Machines. Master's thesis, University of British Columbia, Vancouver, 2007. → pages 12

[80] PFAFF, B., PETTIT, J., KOPONEN, T., AMIDON, K., CASADO, M., AND SHENKER, S. Extending Networking into the Virtualization Layer. In *Proc. of ACM Workshop on Hot Topics in Networks* (2009). → pages 28, 32, 65, 68

[81] POLYZOIS, C. A., AND GARCIA-MOLINA, H. Evaluation of Remote Backup Algorithms for Transaction Processing Systems. In *Proc. of ACM SIGMOD* (1992). → pages 110

[82] PRESOTTO, D. L., AND MILLER, B. P. Process Migration in DEMOS/MP. *ACM Operating Systems Review 17* (1983). → pages 8, 122

[83] RAJAGOPALAN, S., CULLY, B., O'CONNOR, R., AND WARFIELD, A. SecondSite: Disaster Tolerance as a Service. In *Proc. of ACM Conference on Virtual Execution Environments VEE* (2012). → pages 54, 92

[84] RAJAGOPALAN, S., WILLIAMS, D., AND JAMJOOM, H. Pico Replication: A High Availability Framework for Middleboxes. In *Proc. of ACM Symposium on Cloud Computing (SoCC)* (2013). → pages 17, 116

[85] RAJAGOPALAN, S., WILLIAMS, D., JAMJOOM, H., AND WARFIELD, A. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)* (2013). → pages 17, 56, 58, 61

[86] REISNER, P., AND ELLENBERG, L. DRBD v8 – Replicated Storage with Shared Disk Semantics. In *Proc. of International Linux System Technology Conference* (October 2005). → pages 11, 12, 99

[87] RIZZO, L. Netmap: A Novel Framework For Fast Packet I/O. In *Proc. of USENIX Annual Technical Conference (ATC)* (2012). → pages 63

[88] ROESCH, M. Snort - Lightweight Intrusion Detection for Networks. In *Proc. of USENIX Conference on System Administration* (1999). → pages 46

[89] SCALES, D. J., NELSON, M., AND VENKITACHALAM, G. The Design and Evaluation of a Practical System for Fault-Tolerant Virtual Machines. Tech. Rep. VMWare-RT-2010-001, VMWare, Inc., 2010. → pages 12, 13, 54, 113

[90] SCHNEIDER, F. B. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys 22*, 4 (1990). → pages 13

[91] SCHUFF, D. L., CHOE, Y. R., AND PAI, V. S. Conservative vs. optimistic parallelization of stateful network intrusion detection. In *Proc. of ACM Symposium on Principles and Practice of Parallel Programming* (2007). → pages 46

[92] SEKAR, V., EGI, N., RATNASAMY, S., REITER, M. K., AND SHI, G. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)* (2012). → pages 2, 46

[93] SHERRY, J., AND RATNASAMY, S. A Survey of Enterprise Middlebox Deployments. Tech. Rep. UCB/EECS-2012-24, EECS Department, University of California, Berkeley, 2012. → pages 19, 49, 54, 75

[94] SHEVADE, U., KOKKU, R., AND VIN, H. M. Run-Time System for Scalable Network Services. In *Proc. of IEEE International Conference on Computer Communications (INFOCOM)* (2008). → pages 20, 46

[95] SMALL, C. FlowScale. GENI Engineering Conference (Poster), `http://groups.geni.net/geni/attachment/wiki/OFIU-GEC12-status/FlowScale_poster.pdf`, 2012. → pages 47

[96] SOMMER, R., CARLI, L. D., KOTHARI, N., VALLENTIN, M., AND PAXSON, V. HILTI: An Abstract Execution Environment for Concurrent, Stateful Network Traffic Analysis. Tech. Rep. TR-12-003, ICSI, 2012. → pages 20, 46

[97] SOMMER, R., AND PAXSON, V. Exploiting Independent State For Network Intrusion Detection. In *Proc. of Computer Security Applications Conference* (2005). → pages 47

[98] STELLNER, G. CoCheck: Checkpointing and Process Migration for MPI. In *Proc. of Parallel Processing Symposium* (1996). → pages 8, 122

[99] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *ACM SIGCOMM Computer Communications Review 31*, 4 (2001). → pages 66

[100] STROM, R., AND YEMINI, S. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems 3*, 3 (1985). → pages 12, 80

[101] SVÄRD, P., HUDZIA, B., TORDSSON, J., AND ELMROTH, E. Evaluation of Delta Compression Techniques for Efficient Live Migration of Large Virtual Machines. In *Proc. of ACM Conference on Virtual Execution Environments VEE* (2011). → pages 8

[102] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (1995). → pages 62

[103] THOMAS, R. H. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *Transactions on Database Systems (TODS) 4*, 2 (1979). → pages 112

[104] TRAVOSTINO, F., DASPIT, P., GOMMANS, L., JOG, C., DE LAAT, C., MAMBRETTI, J., MONGA, I., VAN OUDENAARDE, B., RAGHUNATH, S., AND WANG, P. Y. Seamless Live Migration of Virtual Machines Over the MAN/WAN. *Future Generations of Computer Systems 22* (October 2006). → pages 8

[105] VALLENTIN, M., SOMMER, R., LEE, J., LERES, C., PAXSON, V., AND TIERNEY, B. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In *Proc. of International Conference on Recent Advances in Intrusion Detection* (2007). → pages 1, 14, 47

[106] VAN RENESSE, R., MINSKY, Y., AND HAYDEN, M. A Gossip-style Failure Detection Service. In *Proc. of International Conference on Distributed Systems Platforms and Open Distributed Processing* (1998). → pages 61

[107] VAQUERO, L. M., RODERO-MERINO, L., AND BUYYA, R. Dynamically Scaling Applications in the Cloud. *ACM SIGCOMM Computer Communications Review 41*, 1 (2011). → pages 8, 47

[108] VERDÚ, J., NEMIROVSKY, M., GARCÍA, J., AND VALERO, M. Workload Characterization of Stateful Networking Applications. In *Proc. of International Symposium on High-Performance Computing* (2008). → pages 20, 46

[109] VERDÚ, J., NEMIROVSKY, M., AND VALERO, M. MultiLayer Processing - An Execution Model for Parallel Stateful Packet Processing. In *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2008). → pages 20, 46

[110] VILLELA, D., PRADHAN, P., AND RUBENSTEIN, D. Provisioning Servers in the Application Tier for e-Commerce Systems. *ACM Transactions on Internet Technologies 7*, 1 (Feb. 2007). → pages 1, 14

[111] The Corosync Cluster Engine. `http://corosync.org/`. → pages 11, 12, 61

[112] HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer. `http://haproxy.1wt.eu/`. → pages 12

[113] Pacemaker: A Scalable High Availability Cluster Resource Manager. `http://clusterlabs.org/`. → pages 11, 12, 61

[114] WANG, R., BUTNARIU, D., AND REXFORD, J. OpenFlow-based Server Load Balancing Gone Wild. In *Proc. of USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services* (2011). → pages 47, 118

[115] WELSH, M., AND CULLER, D. Adaptive Overload Control for Busy Internet Servers. In *Proc. of USENIX Symposium on Internet Technologies and Systems* (2003). → pages 47

[116] WOOD, T., LAGAR-CAVILLA, H. A., RAMAKRISHNAN, K. K., SHENOY, P., AND VAN DER MERWE, J. PipeCloud: Using Causality to Overcome Speed-of-Light Delays in Cloud-Based Disaster Recovery. In *Proc. of ACM Symposium on Cloud Computing* (2011). → pages 11, 12, 112

[117] WOOD, T., RAMAKRISHNAN, K. K., SHENOY, P., AND VAN DER MERWE, J. CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines. In *Proc. of ACM Conference on Virtual Execution Environments VEE* (2011). → pages 8

[118] WU, H., AND KEMME, B. A Unified Framework for Load Distribution and Fault-Tolerance of Application Servers. In *Proc. of International Euro-Par Conference on Parallel Processing (Euro-Par)* (2009). → pages 14

[119] Best Buy Website Crashes Under 'Black Thursday' Traffic. http://www.kare11.com/news/article/998789/391/Best-Buy-website-crashes-under-Black-Thursday-traffic, November 2012. → pages 2, 14

[120] F5 Networks Inc., BIG-IP Product Suite. http://www.f5.com/products/big-ip/. → pages 2, 49, 50, 52, 67

[121] F5 Networks Inc., BIG-IP Local Traffic Manager (LTM). http://www.f5.com/products/big-ip/big-ip-local-traffic-manager/. → pages 22

[122] Click Frenzy Crashes Under Strain. http://www.afr.com/p/technology/click_frenzy_crashes_under_strain, November 2012. → pages 2, 14

[123] Dropbox Users Experiencing Slowness and Inaccessible Files. http://techcrunch.com/2012/08/21/dropbox-users-experiencing-slowness-inaccessible-files/, August 2012. → pages 2, 14

[124] High Traffic Crashes Elections Site. http://www.suntimes.com/news/elections/16187822-505/story.html, November 2012. → pages 14

[125] Embrane Inc., heleos. http://www.embrane.com/products/heleos. → pages 2, 49, 50, 52, 67

[126] Citrix Systems Inc., NetScaler ADC. http://www.citrix.com/netscaler. → pages 2, 22, 49, 50, 52, 67

[127] Riverbed Technology, Stingray Product Family. http://www.riverbed.com/products-solutions/products/application-delivery-stingray/. → pages 2, 49, 50, 52, 67, 68

[128] A10 Networks Inc., SoftAX Virtual ADC: Software-based Application Delivery Controller. http://www.a10networks.com/products/axseries-softax.php. → pages 2, 49, 50, 52, 67

[129] Vyatta Inc., Vyatta Network OS for Amazon. http://www.vyatta.com/product/vyatta-network-os/amazon. → pages 2, 49, 50, 52

[130] Vyatta Inc., Brocade Vyatta vRouter available as a Service. http://www.vyatta.com/content/vyatta-rackspace-cloud. → pages 2, 49, 50, 52

[131] XU, M., BODIK, R., AND HILL, M. D. A "flight data recorder" for Enabling Full-System Multiprocessor Deterministic Replay. *SIGARCH Computer Architecture News 31*, 2 (2003). → pages 13, 112

[132] Riverbed Technology: Infographic - ADC as a Service. http://media-cms.riverbed.com/documents/Riverbed-ADCaaS-Infographic-May7-2013.pdf, 2013. → pages 2

[133] Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. http://aws.amazon.com/message/65648/, April 2011. → pages 5, 14

[134] Amazon AWS Outage Takes Down Netflix On Christmas Eve. http://www.forbes.com/sites/kellyclay/2012/12/24/amazon-aws-takes-down-netflix-on-christmas-eve/, December 2012. → pages 5, 14, 49, 76

[135] Amazon AWS Outage Summary. http://aws.amazon.com/message/67457/, June 2013. → pages 5, 49

[136] The Apache Cassandra Project. http://cassandra.apache.org/. → pages 9, 12, 13, 119

[137] Amazon Web Services: Marketplace for Server Software and Services that Run on the AWS Cloud. https://aws.amazon.com/marketplace. → pages 13

[138] Cloud Foundry. http://www.cloudfoundry.com/. → pages 8, 9, 65, 114

[139] Amazon Web Services, Elastic Load Balancing. http://aws.amazon.com/elasticloadbalancing/. → pages 1, 2, 9, 12

[140] Heroku learns the hard way from Amazon EC2 outage. http://SearchCloudComputing.com, January 2010. → pages 76

[141] Heroku - Incidence Report. https://status.heroku.com/incidents/386, June 2013. → pages 5, 49

[142] Memcached - A Distributed Memory Object Caching System. http://memcached.org. → pages 9, 66, 119, 124

[143] Google App Engine. http://code.google.com/appengine/. → pages 8, 9, 114

[144] Heroku. https://www.heroku.com/. → pages 8, 9, 65, 114

[145] Dynos and the Dyno Manager - Heroku Dev Center. https://devcenter.heroku.com/articles/dynos. → pages 9

[146] Rackspace: Load Balancing as a Service. http://www.rackspace.com/cloud/load-balancing/, 2013. → pages 2

[147] RightScale Infographic Shows Average of 7.5 Hours to Recover from Data Center and Cloud Outages. http://www.rightscale.com/news_events/press_releases/2013/rightscale-infographic-shows-average-of-7.5-hours-to-recover-from-data-center-and-cloud-outages.php, 2013. → pages 5, 14, 76

[148] Amazon Web Services, Auto Scaling. http://aws.amazon.com/autoscaling/. → pages 1, 8, 12, 35, 47, 117

[149] Windows Azure Diagnostic Monitoring and Autoscaling. http://www.paraleap.com/. → pages 8, 12

[150] Xen Blktap2 Driver. http://wiki.xensource.com/xenwiki/blktap2. → pages 92

[151] libcurl - The Multiprotocol File Transfer Library. http://www.tcpdump.org/. → pages 39

[152] Intel DPDK: Data Plane Development Kit. http://dpdk.org. → pages 63

[153] Big Switch Networks Inc., Floodlight OpenFlow Controller. http://www.projectfloodlight.org/floodlight/. → pages 61, 68

[154] Hot Standby Router Protocol (HSRP). http://tools.ietf.org/html/rfc2281. → pages 52

[155] HTTP Archive - Interesting Stats. http://httparchive.org/interesting.php. → pages 66

[156] Iperf: TCP and UDP Bandwidth Performance Measurement Tool. http://iperf.sourceforge.net/. → pages 44

[157] Java TPC-W Implementation, PHARM group, University of Wisconsin. http://www.ece.wisc.edu/pharm/tpcw/. → pages 94

[158] Linux Virtual Server. http://www.linuxvirtualserver.org/. → pages 22

[159] Oracle Corporation, MySQL Replication Implementation. http://dev.mysql.com/doc/refman/5.6/en/replication-implementation.html. → pages 3, 14, 110

[160] MySQL Cluster. http://www.mysql.com/products/database/cluster/. → pages 12, 124

[161] MySQL Cluster 7.0 and 7.1: Architecture and New Features. A MySQL Technical White Paper by Oracle, 2010. → pages 1, 12, 14, 109, 111

[162] MySQL 5.0 Reference Manual. Revision 23486, http://dev.mysql.com/doc/refman/5.0/en/, 2010. → pages 97, 111

[163] Netfilter Packet Filtering Framework. http://www.netfilter.org. → pages 38, 63, 67, 71

[164] Virtual PF_RING. http://www.ntop.org/products/pf_ring/vpf_ring/. → pages 63

[165] The OpenFlow Switch Specification. http://www.openflow.org. → pages 32, 55, 61, 65, 120

[166] Oracle Data Guard: Concepts and Administration. http://download.oracle.com/docs/cd/B28359_01/server.111/b28294.pdf, 2008. → pages 3, 109

[167] Oracle Real Application Clusters 11g Release 2. http://www.oracle.com/technetwork/database/clustering/overview/twp-rac11gr2-134105.pdf, 2009. → pages 12, 111, 124

[168] Percona Tools TPC-C MySQL Benchmark. https://code.launchpad.net/~percona-dev/perconatools/tpcc-mysql. → pages 93

[169] POX OpenFlow Controller. http://www.noxrepo.org/pox/about-pox/. → pages 32

[170] Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/. → pages 119

[171] Open Information Security Foundation: Suricata IDS/IPS. http://www.openinfosecfoundation.org/index.php. → pages 50, 52, 67, 68

[172] Terracotta: Web Sessions. http://www.terracotta.org/products/web-sessions. → pages 9

[173] The TPC-H Benchmark. http://www.tpc.org/tpch/. → pages 87, 93, 94

[174] The TPC-C Benchmark. http://www.tpc.org/tpcc/. → pages 82, 87, 93

[175] The TPC-W Benchmark. http://www.tpc.org/tpcw/. → pages 93, 102

[176] VMWare High Availability. http://www.vmware.com/files/pdf/VMware-High-Availability-DS-EN.pdf. → pages 11, 12

[177] Virtual Router Redundancy Protocol (VRRP). http://tools.ietf.org/html/rfc3768. → pages 52

[178] ZIMMERMANN, A., HANNEMANN, A., AND KOSSE, T. Flowgrind: A New Performance Measurement Tool. In *Global Telecommunications Conference (GLOBECOM)* (2010). → pages 71

[179] F5 Networks Inc., BIG-IP Configuring High Availability. http://support.f5.com/kb/en-us/products/big-ip_ltm/manuals/product/tmos_management_guide_10_0_0/tmos_high_avail.html. → pages 52, 54, 67

[180] Riverbed Technology, Stingray Traffic Manager - User Manual. https://support.riverbed.com/software/stingray/trafficmanager.htm. → pages 12, 52, 54, 67, 68

[181] Vyatta Inc., High Availability Reference Guide. http://www.vyatta.com/downloads/documentation/VC6.5/Vyatta-HA_6.5R1_v01.pdf. → pages 52, 54

[182] Linux-HA Project. http://www.linux-ha.org/doc/. → pages 12, 61, 97, 110