Escape Capsule: Explicit State is Robust and Scalable

Shriram Rajagopalan[†‡], Dan Williams[†], Hani Jamjoom[†], and Andrew Warfield[‡]

[†]IBM T. J. Watson Research Center, Yorktown Heights, NY
[‡]University of British Columbia, Vancouver, Canada

Abstract

Software is modular, and so is run-time state. We argue that by allowing individual layers of the software stack to store isolated runtime state, we cripple the ability of systems to effectively scale or respond to failures. Given the strong desire to build elastic and highly available applications for the cloud, we propose *Slice*, an abstraction that allows applications to declare appropriate granularities of scale-oriented state, and allows layers to contribute the appropriate layer-specific data to those containers. Slices can be transparently migrated and replicated between application instances, thereby simplifying design of elastic and highly available systems, while retaining the modularity of modern software.

1 Introduction

Dynamic scalability and high availability are conjoined goals in distributed systems. In both cases, application state must be packaged in a manner that allows for the nimble and often unanticipated reconfiguration of a system. Consequently, a very large body of research, in both scalable and reliable systems design [1, 2, 7, 8, 14, 18, 23] has struggled with the encapsulation and management of application state. While the problem of application state has been evident for a long time, we point to the fact that systems continue to both fail to scale [28–31], and fail to recover [32–34], as evidence that these problems have not been adequately solved in existing runtime environments.

The position of this paper is that managing application state is the achilles' heel of providing robust, scalable applications today. We argue that this is an *orthogonal* challenge to the largely solved problem of scaling the underlying cloud hosting platform. Cloud providers have demonstrated an excellent capacity to provide low-cost, unreliable computing at enormous scale. This compute model is *ideal* for select stateless applications because of their ability to respond to failures and reconfigurations [37]. If application run times provided explicit support to manage application state across scalability and failure-related reconfigurations, a much broader set of applications would become capable of leveraging infrastructure support for elasticity and high availability.

The principal philosophy of our work is that the application is not the correct granularity to consider either scalability or availability: 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. External factors such as TCP state machines, entropy sources, and helper applications all conspire to produce critical application state *outside* the address space of a given application.

To this end, we propose a new abstraction that allows each software layer to independently identify fine-grained sub-state in a session-based unit called a *capsule*. Capsules are explicitly linked together across layers in the software stack to form a vertical chain, or *Slice*, that represents all important states belonging to a particular session. A Slice can be live migrated or replicated from one running application instance to another. The combined ability migrate and replicate Slices creates new opportunities for achieving efficient elasticity and high availability (HA).

As a proof-of-concept, we use an Apache/PHP web server cluster to show how a legacy application can be restructured in a manner that specifies a Slice that extends through the application's heap, network state in the kernel, and even extending to external OpenFlow forwarding rules. We further discuss how this application—once restructured to use Slices—can achieve dynamic scale and a graceful, load-balanced failure recovery.

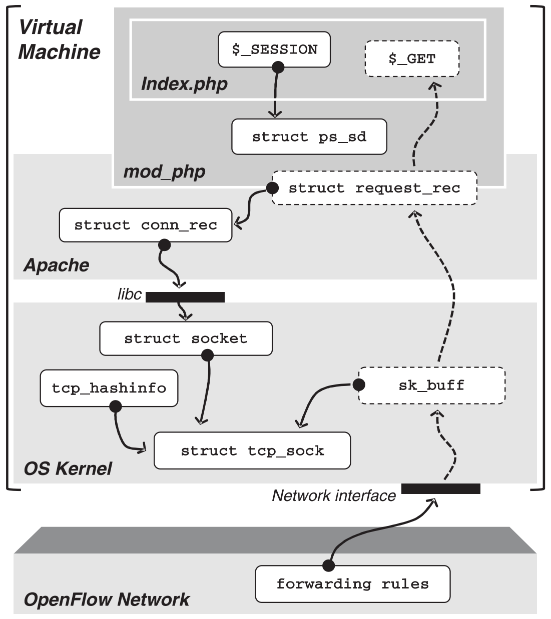

Figure 1: Footprints of a client session, in Apache/PHP. Solid boxes denote session-wide state, while dotted boxes denote per-request (ephemeral) state. Solid arrows indicate dependencies while dotted arrows indicate logical control flow.

2 Slices

Consider, as a running example, a simple Apache/PHP based web application deployed on a cluster of virtual machines (VMs). We assume the following: (a) the VMs have identical operating environments (kernel, process binaries, etc.), (b) the filesystem is shared among the VMs (e.g., NFS), and (c) the cluster is deployed over a Software Defined Network (SDN), such as OpenFlow [38]. As shown in Figure 1, a request-response transaction creates/modifies state in the OpenFlow network, Linux kernel, Apache web server and the PHP application. For the purpose of exposition, we have only considered a representative subset of the data structures in each layer.

2.1 Defining a Slice

As depicted in Figure 2, a Slice is a collection of related states associated with a client session that spans all layers of the software stack. For ease of discussion, we assume that the software stack consists of only one subsystem per layer. At each layer, the Slice is made up of one or more *capsules*, where a capsule is the bare minimum—essential—state associated with the corresponding client session. For example, in the OS kernel in Figure 1, the socket and TCP state associated with each client may be identified as a capsule. Each layer explicitly expresses

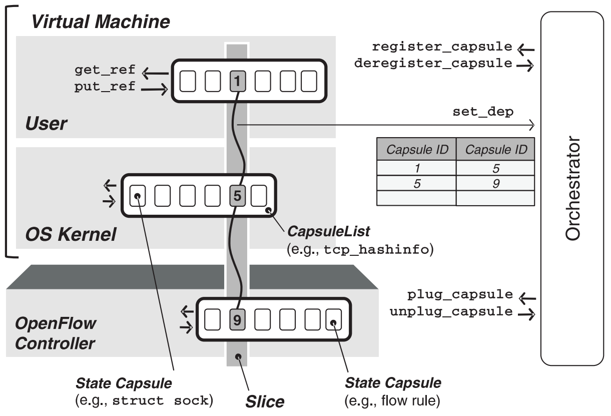

Figure 2: Explicitly defined state capsules at each layer linked together to form a Slice. Every layer exposes a common set of APIs allowing a centralized entity to track dependencies between capsules within a Slice and migrate or replicate it across VMs.

dependencies between its capsules and the capsules in the lower layer. Capsules from each layer are chained together to make a Slice. For the example in Figure 1, each Slice contains the bare minimum state—across all layers—associated with a given client session.

By classifying state in each layer of a VM as (a) belonging to a Slice, (b) global state shared among all Slices, or (c) other,[1] 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 an Orchestrator with a global view of Slices. A Slice, thus, forms the fundamental unit of reconfiguration for scaling and high availability. We discuss the Slice based approach to elasticity and HA in Section 3.

2.2 The Slice API

In order for a software layer to delineate its capsules, we advocate an API based approach to state management. Figure 3 describes a set of APIs that should be implemented by each layer. These APIs allow any layer to delineate which pieces of its state form a capsule, identify their dependencies to other layers' state, and enable capsules (and subsequently Slices) to be migrated or replicated between VMs.

Delineating State. A layer identifies its own capsules by structuring its software such that capsule state is independent from other state used in the layer. To this end, a `CapsuleList` interface is implemented to manage

[1]This is a generalization of the Split/Merge [21] memory abstraction from our earlier work.

```
// DELINEATING STATE
register_capsule(id);              // Register with orchestrator
deregister_capsule(id);

// SPECIFYING DEPENDENCIES
set_dep(myid, capsule_id);
create_slice(slice_key, capsule_id);

// MANAGING CAPSULES
interface CapsuleList {

  // CANONICALIZATION
  get_id(handle);                  // Returns capsule id
  get_handle(id);                  // Returns capsule handle

  // UNPLUGGING/PLUGGING
  unplug_capsule(id, buffer);      // Detach & serialize
  plug_capsule(id, buffer);        // Deserialize & attach

  get_ref(handle);                 // Increment refcnt
  put_ref(handle);                 // Decrement refcnt
};
```

Figure 3: The Slice API

different capsules in a layer. As state is allocated within its structures, the layer communicates the creation and deletion of each capsule to the Orchestrator through the `register_capsule` and `deregister_capsule` calls, respectively. The `CapsuleList` interface forces all capsules to be referenced in a standard manner as part of a Slice (or by other layers) and migrated or replicated, as described below.

Specifying Dependencies. If a capsule has dependencies, they need to be expressed explicitly to the Orchestrator using the `set_dep` call. As it creates capsules, each layer specifies its dependencies from the lower layer to the Orchestrator. The Orchestrator maintains a mapping between a capsule ID and the set of capsules it depends on.[2] When the uppermost layer issues the `create_slice` call with its capsule ID, the Orchestrator runs through the dependency mappings and generates a list of capsules that fall into the Slice.[3]

Canonicalization. To specify a dependency, a layer must obtain a unique capsule ID from the underlying layer. To this end, each capsule implements a mechanism to canonicalize references to state from the underlying layer, as part of the `CapsuleList` interface. Typically, references are specified in the form of handles (e.g., file descriptors, pointers, etc.). The value of a handle may not be unique across layers in the same level, when considering the entire set of VMs in the cluster. The `get_id` call translates a handle to its globally unique ID that was registered with the Orchestrator. The `get_handle` call does the inverse translation.

Unplugging/Plugging. The Orchestrator can detach a Slice from a VM and the network and attach it to a different VM, forming the basis for migration or replication. To accomplish this, each capsule must have the capability to be "unplugged" from its layer in a VM and "plugged" into an equivalent layer in a different VM. To this end, each capsule implements `plug_capsule` and `unplug_capsule` as part of the `CapsuleList` interface. Unplugging/plugging a capsule in the OpenFlow layer translates to deleting the corresponding flow rules and installing updated flow rules, respectively.[4] The plug/unplug operations must ensure that all references to the capsule have been released before unplugging a capsule. To help track references, each capsule forces the software in its layer to explicitly obtain/release references to capsules by using the `get_ref` and `put_ref` API, respectively.

3 Slicing Elasticity and High Availability

The ability to unplug, *live migrate*, and plug a Slice enables two powerful operations:

- **Split** the set of Slices operated by one VM into two or more subsets, such that each subset of Slices can be processed in parallel by multiple VMs.
- **Merge** the set of Slices operated by two or more VMs into a single superset to be processed by one VM.

We now discuss the high level design of a system that leverages splitting and merging of Slices to achieve elasticity and high availability.

3.1 Dynamic Scalability

An application designed around the Slice abstraction can be transparently and rapidly scaled out while avoiding the creation of infrastructure hotspots. Consider the standard scale out operation in the cloud. When the workload crosses a particular threshold, one or more VMs are added to the application cluster to handle new incoming load. By splitting and migrating Slices, not only can an application scale-out, but it can also *shed load* from currently overloaded VMs onto the newly added ones.

When the load on the system decreases, the set of Slices in one or more VMs can simply be merged into other VMs, thereby, improving overall system utilization. This is different from current scale-in strategies that rely on

[2]Note that capsule to capsule dependencies can have only local scope, i.e., within the same VM.

[3]In practice, an implementation would contain a Local Orchestrator Agent inside the VM and a Global Orchestrator. A layer would only interact with the local agent. Once a slice has been created, information can be asynchronously propagated to the Global Orchestrator.

[4]In order to maintain client connectivity as a Slice moves across VMs, each VM is configured with the same MAC and IP address.

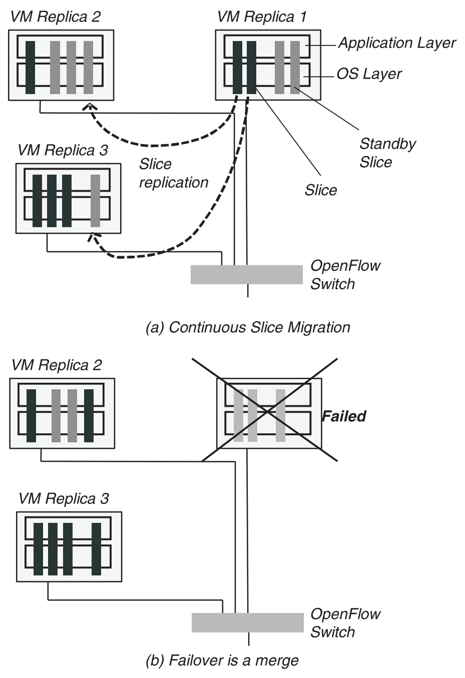

Figure 4: Example of using Slices for HA

waiting for existing sessions to terminate before killing the running VM. Splitting and merging of Slices is transparent to the client such that the network connectivity remains intact. The Orchestrator controls scale-out, load shedding, scale-in at runtime through configurable external policies.

3.2 Fine-Grained High Availability

The ability to live migrate a Slice opens up new avenues for exploring fine-grained high availability (HA) solutions. Instead of creating a new VM to hold backup data in a Slice, we can reuse spare resources in the existing cluster [27].

Lightweight HA. As shown in Figure 4(a), when protecting data in a VM, only the Slice data needs to be backed up. Global state is already synchronized across other VMs in the cluster; other state in every VM is either ephemeral with respect to the Slice or simply independent of a Slice's state. By replicating only Slice data, the resource requirements of HA are drastically reduced compared to VM replication techniques.

Failover is a Merge. Continuous VM replication [7] can be applied at Slice level for high availability. As illustrated in Figure 4(b), when a VM fails, the Orchestrator simply merges the standby Slices into their respective VMs. The associated network flows are also re-routed as part of the merge process. Slice level checkpointing can leverage output buffering [25] and speculative execution, to ensure that only committed state is exposed to the client. As a result, client connectivity remains intact post failover.

Fine-Grained Output Buffering. A VM can be considered as a collection of Slices, one per session, whose outputs are independent of one another. VM replication techniques [7, 14, 17, 22] treat the entire system as one single state machine, for the purposes of output buffering. Such coarse grained output buffering adds a high overhead to the end-to-end latency of a session. Higher checkpoint frequencies can mitigate latency impact, but the suspend/resume overhead per checkpoint can no longer be amortized. Slices can be checkpointed concurrently at a high frequency, thereby enabling individual session outputs to be released independent of other Slices, while reducing the latency overhead of output buffering.

Proactive Load Balancing. Replicating application state *en masse* [27] between two VMs can lead to uneven distribution of load across the system when one or more VMs fail. The Orchestrator distributes the standby Slices from a VM in a load balanced fashion across the cluster. Such a proactive load balancing technique ensures that no single VM will be overloaded when one or more VMs in the cluster fail.

Elastic HA. The HA implementation can be made elastic vis-a-vis the amount of resources available. As the cluster shrinks/grows, the Orchestrator continually rebalances the load on the cluster by splitting and merging both active and standby Slices. The rebalancing of standby Slices maintains the load balanced recovery property stated above.

4 Restructuring the Software Stack

Designing an entire software stack around the notion of Slices requires that each subsystem/layer be modular with respect to its state. The system must identify the state that goes into a Slice, manage synchronized state across Slices, manage consistency during Slice migration, etc. This section discusses the key challenges in restructuring a layer for slicing.

Lists. The `CapsuleList` interface shown in Figure 3 enables tracking the number of accessors of an object and provides a generic mechanism to plug/unplug objects. A software layer managing multiple capsules of the same type generally maintains some form of a list datastructure. For example, the kernel maintains a hash table of open TCP sockets in the `tcp_hashinfo` hash table. References are obtained through the `__inet_lookup_skb` function. In such cases, only minimal modifications are required to the existing implementation to support functions like `unplug_capsule/plug_capsule`. In the absence of such interfaces, the developer needs to take explicit control of state management in the layer and implement the CapsuleList interface.

Unplugging/Plugging Safely. A Slice can be unplugged only when there are no references to it. Generally, by suspending the ingress flow of the client connection, the client session's request pipeline drains out quickly. Inside the VM, any inflight request/response data in the Slice needs to be processed and flushed out before the unplug operation. When unplugging a capsule, any references to state outside of its Slice, in the current VM should be released and reacquired again, upon plugging into a different VM.[5]

Plugging kernel objects, like TCP state, can piggyback on existing code to create a TCP connection and unplugging can reuse the code to close a TCP connection. Care needs to be taken when dealing with resources that generate output. For instance, in the case of a TCP object, SYN or RST packets have to be discarded when plugging and unplugging.

Network Endpoints with Same MAC/IP. Migrating the Slice requires that the target VM have the same MAC/IP, so as to maintain client connectivity. In our earlier work [21], we demonstrated the feasibility of this approach by dynamically scaling a cluster of middleboxes with same network endpoint on an OpenFlow network.[6]

Global State Synchronization. Global state exists in several forms: locks, read-mostly configuration data, counters, and other data structures. Global state can be thought of as belonging to capsules that are eternally plugged into every node in the cluster and kept synchronized. The Slice API can be augmented to specify global state and its consistency requirements. Non-critical global state (e.g., counters) can generally tolerate eventual consistency and can be managed using techniques like combiners [10, 16], sloppy counters [5], application-specific merge procedures [26], etc. State that requires strong consistency can be managed using standard distributed locking techniques [6, 11].

[5]During Slice migration, incoming requests on the client session can be buffered at the target VM's hypervisor.

[6]IP rewriting load balancers will not work in this scenario because they only target solutions that migrate application layer session state; state that is generally independent of the server's MAC/IP.

5 Limitations

Applications that require frequent global synchronization will not benefit from the Slice abstraction, if the synchronization requires strong consistency. We acknowledge that this is a significant barrier towards application scalability. But, we note that the application would have to deal with the synchronization issue in any other design as well, if it wishes to scale across multiple VMs in the cloud.

Similarly, Slices may not be suitable for applications with extremely short lived sessions. The overhead of initiating and terminating Slice replication cannot be amortized if the sessions last for very short periods, such as a few seconds. In the common case, where session durations are on the order of minutes, both stateless and stateful systems can benefit from Slice based approach to elasticity and high availability.

6 Related Work

Our earlier work [21] described system level abstractions to provide transparent elasticity for virtual middleboxes. Along a similar vein, this paper targets a much broader class of end-user applications and aims to provide both elasticity and high availability. Unlike middleboxes, these applications do not easily lend themselves to the data and execution model assumed in our previous work.

Vertical Abstractions. The vertical Slice abstraction shares close similarities with Resource Containers [3]. While both abstractions encapsulate session state at all layers of the software stack, resource containers enable fine-grained control over resource consumption for a given session, while Slices enable load-balanced elasticity and high availability for the entire application cluster.

Migrating Slices vs Processes. Slice migration faces some of the same challenges as process migration. There is a plethora of work on process migration [4,9,15,19,20, 24]. 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. Zap [12, 19] proposes a solution by introducing the notion of process domains—a collection of processes with a virtualized view of the OS. During migration, the entire process domain and its system dependencies are migrated to the target.

Using the Slice abstraction, it is not necessary to provide a virtualized view of the OS. Capsules are created and managed in every layer of the system. The objects referred to by descriptors that can be problematic at the process level reside in capsules in the OS and are explicitly linked to the same Slice as their descriptors. Therefore, as long as capsules are specified at all layers in the stack, (to some extent, Zap [12] has demonstrated the feasibility of such encapsulation on modern kernels, for popular applications like Apache, MySQL, etc.), migrating a Slice leaves no residual dependencies at the source.

Dynamic Scalability. Existing work on application scalability has mostly focused on managing state in the application layer. Stateless applications offload session state management to scalable backends like Dynamo [8] and Sinfonia [1]. Stateful applications [39–42] typically use application clustering to scale in the cloud.

The common theme underscoring all these systems is that a deployment is scaled in *units of virtual machines* [13, 35]. Only the application layer state can migrate across the cluster. 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 slowing down the scale-in process. With Slices, an application can be split/merged dynamically according to load, in an efficient manner, as described in Section 3.1.

High Availability. System level approaches [7, 14, 22] can provide transparent HA by replicating the entire VM to a standby host. The failover process is completely transparent to both end user and the application. However, protecting an entire VM's state is a sledgehammer approach to HA, with two main limitations: (1) it incurs at least 2X resource overhead by maintaining a hot spare for each VM, which limits the scalability of the application cluster, and (2) latency sensitive applications incur high performance overhead [7] as the checkpoints are coarse grained. Application level approaches to HA [27, 36] overcome these limitations. However, they fail to manage system level state resulting in loss of transparency during failover. Slice based HA design provides the same recovery properties as system level HA techniques but at much lower overhead, similar to application level approaches, as described in Section 3.2.

7 Conclusion

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 paper presents 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.

References

[1] 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).

[2] 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).

[3] 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).

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

[5] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An Analysis of Linux Scalability to Many Cores. In *Proc. of USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2010).

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

[7] 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).

[8] 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).

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

[10] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proc. of USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2012).

[11] 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).

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

[13] 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).

[14] LEE, D., WESTER, B., VEERARAGHAVAN, K., NARAYANASAMY, S., CHEN, P. M., AND FLINN, J. Re-Spec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. In *Proc. of ASPLOS* (2010).

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

[16] 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).

[17] 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).

[18] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the Sync. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (2006).

[19] 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).

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

[21] 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).

[22] 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.

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

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

[25] STROM, R., AND YEMINI, S. Optimistic Recovery in Distributed Systems. *ACM Trans. Comput. Syst. 3*, 3 (1985).

[26] 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).

[27] 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).

[28] 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.

[29] Click Frenzy Crashes Under Strain. `http://www.afr.com/p/technology/click_frenzy_crashes_under_strain_C2b9SnGMI3bzR497LRHwsJ`, November 2012.

[30] Dropbox Users Experiencing Slowness and Inaccessible Files. `http://techcrunch.com/2012/08/21/dropbox-users-experiencing-slowness-inaccessible-files/`, August 2012.

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

[32] Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. `http://aws.amazon.com/message/65648/`, April 2011.

[33] 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.

[34] 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.

[35] Amazon EC2: Auto Scaling. `http://aws.amazon.com/autoscaling/`.

[36] Linux-HA Project. `http://www.linux-ha.org/doc/`.

[37] Netflix Tech Blog: Lessons Netflix learned from the AWS Outage. `http://techblog.netflix.com/2011/04/lessons-netflix-learned-from-aws-outage.html`.

[38] The OpenFlow Switch Specification. `http://www.openflow.org`.

[39] Oracle Applications on AWS. `http://aws.amazon.com/enterprise-applications/oracle/`.

[40] SAP and Amazon Web Services. `http://aws.amazon.com/sap/`.

[41] Terracotta: Web Sessions. `http://www.terracotta.org/products/web-sessions`.

[42] Cloud: WebSphere Application Server. `http://www.ibm.com/developerworks/downloads/ws/was/cloud.html`.