PipeCloud: Using Causality to Overcome Speed-of-Light Delays in Cloud-Based Disaster Recovery

Timothy Wood[†]
The George Washington University
timwood@gwu.edu

H. Andrés Lagar-Cavilla
AT&T Labs - Research
andres@research.att.com

K.K. Ramakrishnan
AT&T Labs - Research
kkrama@research.att.com

Prashant Shenoy
University of Massachusetts Amherst
shenoy@cs.umass.edu

Jacobus Van der Merwe
AT&T Labs - Research
kobus@research.att.com

ABSTRACT

Disaster Recovery (DR) is a desirable feature for all enterprises, and a crucial one for many. However, adoption of DR remains limited due to the stark tradeoffs it imposes. To recover an application to the point of crash, one is limited by financial considerations, substantial application overhead, or minimal geographical separation between the primary and recovery sites. In this paper, we argue for cloud-based DR and pipelined synchronous replication as an antidote to these problems. Cloud hosting promises economies of scale and on-demand provisioning that are a perfect fit for the infrequent yet urgent needs of DR. Pipelined synchrony addresses the impact of WAN replication latency on performance, by efficiently overlapping replication with application processing for multi-tier servers. By tracking the consequences of the disk modifications that are persisted to a recovery site all the way to client-directed messages, applications realize forward progress while retaining full consistency guarantees for client-visible state in the event of a disaster. PipeCloud, our prototype, is able to sustain these guarantees for multi-node servers composed of black-box VMs, with no need of application modification, resulting in a perfect fit for the arbitrary nature of VM-based cloud hosting. We demonstrate disaster failover to the Amazon EC2 platform, and show that PipeCloud can increase throughput by an order of magnitude and reduce response times by more than half compared to synchronous replication, all while providing the same zero data loss consistency guarantees.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems

General Terms

Design, Performance

Keywords

Disaster Recovery, Virtualization, Cloud Computing

[†]This work was performed while at University of Massachusetts.

SOCC'11, October 27–28, 2011, Cascais, Portugal.

1. INTRODUCTION

Businesses and government enterprises utilize Disaster Recovery (DR) systems to minimize data loss as well as the downtime incurred by catastrophic system failures. Current DR mechanisms range from periodic tape backups that are trucked offsite, to continuous synchronous replication of data between geographically separated sites. Typical DR solutions incur high infrastructure costs since they require a secondary data center for each primary site as well as high bandwidth links to secondary sites. Further, the "larger" the potential impact of a disaster, the greater is the need for geographic separation between the primary and the secondary replica site. The resulting wide area latency can, however, place a large burden on performance.

The recent emergence of commercial cloud computing has made cloud data centers an attractive option for implementing cost-effective DR due to their "resource-on-demand" model and high degree of automation [26]. During normal operation, the cost of providing DR in the cloud can be minimized, and additional resources only need to be brought online—and paid for—when a disaster actually occurs. In addition, the cloud platform's ability to rapidly activate resources on-demand helps minimize the recovery cost after a disaster. The automation that is designed into accessing cloud services enables the DR service to support *Business Continuity*, with substantially lower recovery times.

Despite these attractive economics, a major barrier to using cloud data centers for DR is their large geographic separation from primary sites—the increased latency in communicating with distant cloud sites can become a major performance bottleneck. This is amplified by the limited control cloud users have over the actual placement of their cloud resources. Consequently, a synchronous replication scheme will expose every data write to the performance impact of this wide-area latency, forcing system administrators to seek alternative solutions. Often, such alternatives trade off loss of data for performance by using asynchronous replication, in which a consistent "snapshot" is replicated to the backup site. Asynchronous replication improves performance since the primary site can proceed without waiting for the replication to complete. However, disk writes at the primary site subsequent to the last replicated snapshot will be lost in case of a disaster. Consequently, to implement cloud-based DR for mission-critical business applications, we must design a mechanism that combines the performance benefits of *asynchronous replication* with the no-data-loss consistency guarantee of *synchronous replication*.

We propose *Pipelined Synchronous Replication* as an approach to provide high performance disaster recovery services over WAN

links connecting enterprises and cloud platforms. Pipelined synchrony targets client-server style applications, and exploits the fundamental observation that an external client is only concerned with a guarantee that data writes related to its requests are committed to storage (both at the primary and the secondary) before a response is received from the system. Because it can potentially take a large amount of time for the secondary site to receive the write, commit and acknowledge, there is a substantial opportunity to overlap processing during this time interval which synchronous approaches ignore. The opportunity is even more compelling when we observe that multi-tier applications can take advantage of pipelined synchronous replication by overlapping remote replication with complex processing across the multiple tiers that are typical in such environments. The key challenge in designing pipelined synchrony is to efficiently track all writes triggered by processing of a request as it trickles through the (multi-tier) system, and to inform the external entity (i.e., a client) only when all these writes have been made "durable". We achieve this by holding up network packets destined for the client until all disk writes that occurred concurrently with request processing have been acknowledged by the backup. This approach imposes causality (i.e., via Lamport's happened-before relation) across externally-bound network packets and disk writes, providing the same consistency guarantee as if the disk writes had been performed synchronously. Since we seek to implement pipelined synchrony in an Infrastructure-as-a-Service (IaaS) cloud environment that relies on Virtual Machines (VMs) to encapsulate user applications, an additional challenge is to employ black-box techniques when providing cloud-based DR.

Our work combines ideas from storage replication, speculative execution, and distributed systems. Our replication scheme builds upon the external synchrony observations by Nightingale et al. [18]: That the dichotomy between synchronous and asynchronous storage is only relevant to the external visibility of storage events, and thus can be efficiently hidden from clients in many cases. Our focus on VM black-boxes also draws inspiration from the speculative execution concepts used in Remus [6] to provide high availability within the LAN. We extend the notions of external synchrony [18] and single-VM LAN-based high availability in Remus [6] to implement cloud-based disaster recovery over WANs for multi-VM multi-tier or distributed applications. Finally, we use a communication-based mechanism enforcing causality as defined by Lamport's *happened-before* relation [13], that borrows techniques from eventually consistent distributed systems [4, 12, 23].

Our pipelined synchronous replication-based disaster recovery system, *PipeCloud*, effectively exploits cloud resources for a cost-effective disaster recovery service. PipeCloud makes the following contributions: (i) a replication system that offers clients synchronous consistency guarantees at much lower performance cost by pipelining request processing and write propagation; (ii) a communication based synchronization scheme that allows the state of distributed or multi-tier applications to be replicated in a consistent manner; (iii) a formal analysis of the consistency guarantees of pipelined synchrony; (iv) an implementation that efficiently protects the disks of virtual machines without any modifications to the running applications or operating system; and (v) a thorough evaluation of PipeCloud's performance in normal operating conditions and when using Amazon EC2 to recover from a disaster.

Our results illustrate the significant performance benefits of using pipelined synchrony for disaster recovery. PipeCloud substantially lowers response time and increases the throughput of a database by twelve times compared to a synchronous approach. When protecting the TPC-W E-commerce web application with a secondary replica 50ms away, PipeCloud reduces the percentage of requests violating a one second SLA from 30% to 3%, and provides throughput equivalent to an asynchronous approach. PipeCloud can replicate state to backups sites 50ms further away than synchronous replication, providing improved application throughput while increasing the resiliency to large-scale disasters. We demonstrate that PipeCloud offers the same consistency to clients as synchronous replication when disasters strike, and evaluate the potential of using cloud services such as EC2 as a backup site.

2. DISASTER RECOVERY CHALLENGES

The two key metrics that determine the capabilities of a Disaster Recovery (DR) system are *Recovery Point Objective* (RPO) and *Recovery Time Objective* (RTO). The former refers to the acceptable amount of application data that can be lost due to a disaster: a zero RPO means no data can be lost. RTO refers to the amount of downtime that is permissible before the system recovers. A zero RTO means that failover must be instantaneous and transparent and is typically implemented using hot standby replicas.

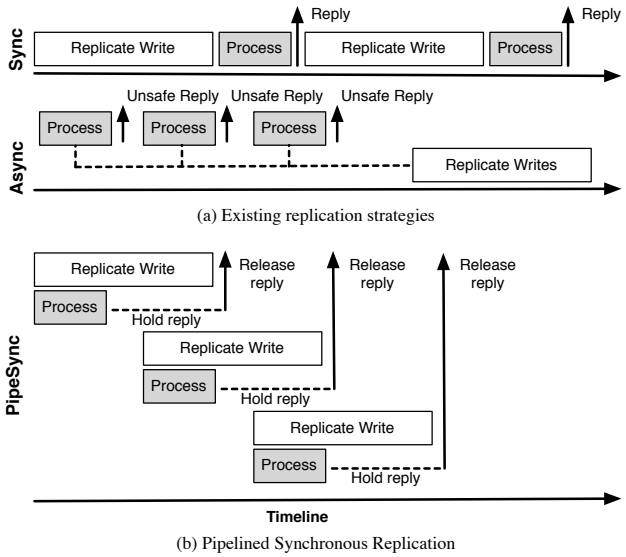
In scenarios where a small downtime is tolerable (i.e., RTO>0), the cost of DR can be reduced substantially by eliminating hot standbys. The limiting factors in optimizing the RTO in this case depend on engineering considerations: how swiftly can we provision hardware resources at the backup site to recreate the application environment? Once resources have been provisioned, what is the bootstrapping latency for the application software environment? Since disasters happen mid-execution, is a recovery procedure such as a file system check necessary to ensure that the preserved application data is in a usable state? As indicated earlier, leveraging cloud automation can significantly improve the RTO metric; we have also shown that the cloud's economics driven by on-demand resource utilization are a natural fit for substantially lowering the cost of DR deployments [26].

Next, in § 2.1, we discuss the impact of latency between the primary and secondary sites on the RPO that can be achieved with different replication strategies. In § 2.2 we introduce a broader RPO definition which takes into consideration the client's view of a DR system. Finally, in § 2.3 we describe the specific DR operational assumptions and system model considered in our work.

2.1 Replication Strategies and Latency

When preserving stored data to a secondary location, the network round trip time (RTT) between the primary and secondary locations significantly impacts the choice of replication algorithm, and thus, the ability to provide an RPO as close as possible to zero (i.e., no data loss). Latency considerations lead to a choice between two primary modes of data replication for disaster survivability: synchronous and asynchronous replication [9]. With synchronous (sync) replication, no data write is reported as complete until it has succeeded in both the primary and secondary sites. With asynchronous (async) replication, writes only need to succeed locally for the application to make progress, and they will be trickled back opportunistically to the secondary replica. The timelines in Figure 1 (a) illustrate the behavior of sync and async replication.

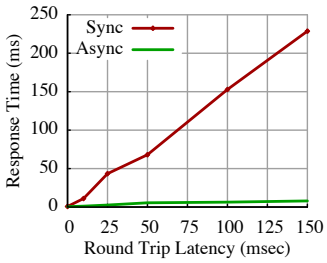
With sync replication, applications obtain an RPO of zero by construction: no application progress is permitted until data has been persisted remotely. However, a higher latency results in corresponding increase in the response time and lower throughput for client-server type applications. Figure 2 (a) shows the performance impact of increasing the latency between the primary and backup sites. For this experiment, we used DRBD [14], a standard block device replication tool which supports both sync and async modes, to protect a MySQL database. The response time of performing



(a) Existing replication strategies

(b) Pipelined Synchronous Replication

Figure 1: Replication strategies. (a) Existing approaches: Synchronous and Asynchronous replication. (b) Pipelined Synchronous Replication.



(a)

Location	RTT (ms)	Distance (km)
Virginia	16	567
California	106	4,250
Ireland	92	4,917
Singapore	281	15,146

(b)

Figure 2: (a) Latency significantly reduces performance for synchronous replication. Table (b) lists round trip latency and approximate distance from UMass to Amazon data centers.

inserts into the database increases linearly with the RTT. Even for relatively short distances, e.g., from Massachusetts to the EC2 data center in Virginia (16msec, Figure 2 (b)), the response time degrades noticeably. For these reasons, while mission-critical applications need to resort to synchronous replication, they incur a high performance cost. To mitigate this overhead, the secondary site is often chosen to be geographically close (within tens of km) to the primary. However, replication to nearby facilities is unlikely to withstand many kinds of disasters that have struck infrastructure in recent memory: hurricanes, earthquakes and tsunamis, and regional energy blackouts. Legislative attempts to enforce backup distances of a few hundred kilometers have failed due to technical considerations. As a result, the positioning of the secondary site must balance the requirement of being a safe distance from the primary against the overhead incurred by longer distance links. For example, many Wall Street companies replicate to data centers within a geographical "doughnut" that ranges from thirty to seventy kilometers around downtown Manhattan [11]. Unfortunately, having that tight a control over data center placement is typically impossible when using cloud resources as clients have only coarse control over their location (e.g., EC2 offers two datacenters, "East coast" and "West coast", in the U.S.).

The alternative is Asynchronous replication, which sacrifices RPO

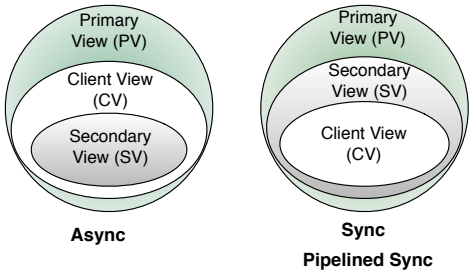


Figure 3: Relationship between application views under different replication algorithms. In particular, SV and CV relations define the RPO guarantees upon disaster.

guarantees, as illustrated with the "unsafe replies" of the second timeline in Figure 1 (a). However, asynchrony also decouples application performance from data preservation. System builders are thus afforded great leeway in how to schedule transfer of state to the secondary site. The area of asynchronous replication has thus been a fertile ground for optimization research [10, 9] that explores the tradeoffs between replication frequency, application RPO demands, financial outlay by application owners, and possibly even multi-site replication as outlined above. Fundamentally, however, async replication exposes applications to a risk of inconsistency: clients may be notified of a request having completed even though it has not yet been preserved at the backup and may be lost in a disaster. Further, the number of these "unsafe replies" increases as latency rises since the backup lags farther behind the primary.

2.2 Client RPO Guarantees

To better illustrate the impact of replication algorithm on application consistency, we formalize here our notion of *Client* RPO guarantees. We define three views of the application state: the primary site view (PV), the secondary site view (SV), and the external clients view (CV), and we illustrate their relationship in Figure 3.

In both synchronous and asynchronous replication, the SV is a subset of the PV: the SV lags in time behind the PV, and reflects a past state of the application data. For asynchronous replication the delta between SV and PV can be arbitrarily large, although in practice it is often bounded by an unpredictable buffer size. For synchronous replication, the delta is at most one write (or one logical write, if we consider a set of scatter-gather DMA writes issued concurrently as one single logical packet): an application cannot make further progress until that write is made durable at the secondary. In all cases, the CV is also a subset of the PV, since the primary performs local processing before updating clients.

The key difference resides in the delta between CV and SV. In asynchronous replication, clients are acknowledged before writes are made durable, and thus the SV is a subset of the CV, reflecting the non-zero RPO. In synchronous replication, clients are acknowledged only after writes have been persisted remotely, and thus the CV is a subset of the SV. As shown in Figure 3, Pipelined Synchrony, which we present in §3, maintains the same client RPO guarantees as synchronous replication.

2.3 System Model

Our work assumes an enterprise primary site that is a modern virtualized data center and a secondary site that is a cloud data center; the cloud site is assumed to support Infrastructure-as-a-Service (IaaS) deployments through the use of system virtualization. The primary site is assumed to run multiple applications, each inside virtual machines. Applications may be distributed across multiple VMs, and one or more of these VMs may write data to disks that

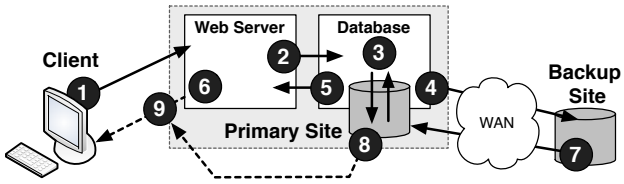


Figure 4: Replication in a simple multi-tiered application. Number sequence corresponds to Pipelined Synchronous Replication.

require DR protection. Data on any disk requiring DR protection is assumed to be replicated to the secondary site while ensuring the same client RPO guarantees as sync replication. We assume that a small, non-zero RTO can be tolerated by the application, allowing us to leverage cloud automation services to dynamically start application VMs after a disaster. We further assume that application VMs are black-boxes and that we are unable to require specific software or source code changes for the purposes of disaster recovery. While this makes our techniques broadly applicable and application-agnostic, the price of this choice is the limited "black-box" visibility into the application that can be afforded at the VM level. Our mechanisms are, however, general, and could be implemented at the application level.

Despite adhering to a black-box methodology, we require applications to be well-behaved with respect to durability. For example, we cannot support a MySQL database configured with the MyISAM backend, which does not support transactions (and durability), and replies to clients while writes may still be cached in memory. More broadly, applications should first issue a write to storage, ensure such write has been flushed from memory to the disk controller, and only then reply to a client the result of an operation. Synchronous replication cannot guarantee adequate data protection without this assumption, and neither can our approach.

3. PIPELINED SYNCHRONY

Given the WAN latencies between the primary and a secondary cloud site, we seek to design a replication mechanism, as part of an overall DR solution, that combines the performance benefits of async replication with the consistency guarantees of sync replication. To do so, we must somehow leverage the overlapping of replication and processing that is typical of asynchrony, while retaining the inherent safety of synchronous approaches.

We make two primary observations. First, from the perspective of an external client it does not matter if the transmission of the write and the processing overlap, as long as the client is guaranteed that the data writes are durably committed to the backup before it receives a reply. Second, the potential for performance improvements compared to synchronous replication is substantial when there is a large delay to overlap, as is the case of DR systems with high WAN latencies. The case becomes stronger for multi-tiered applications and, more generally, clustered applications or distributed systems interfacing with external clients via some form of frontend. These applications often require complex processing across multiple tiers or components. We apply these observations to realize a technique called *pipelined synchronous replication*.

3.1 Pipelined Synchronous Replication

Pipelined synchronous replication is defined as blocking on an externally visible event until all writes resulting from the (distributed) computation that generated the external event have been committed to disk at the primary and the secondary. When processing a request, pipelined synchronous replication allows overlapping of computation and remote writes—i.e., writes to the primary are asynchronous and pipelined with the remote writes, allowing subsequent processing to proceed. Upon generating an externally visible event (such as a network packet or a reply), however, the event must be blocked, and not released to the client until all pending writes have finished. In essence, our approach mitigates the performance penalties associated with speed-of-light delays by overlapping or pipelining computation and remote writes, like in async replication, while ensuring the same relation between client view and secondary view as synchronous replication. Figure 1 (b) depicts an illustrated timeline of the pipelined synchronous approach.

To contrast pipelined synchronous replication with existing replication strategies, consider the illustrative example in Figure 4. Here a client, Alice, goes through the process of buying a ticket from a travel website, by submitting her credit card information in step 1. As is common, Alice interacts with a frontend web server which may perform some processing before forwarding the request on to a backend database (step 2) to record her purchase. In step 3, the DB writes the transaction to disk. Since this is critical state of the application, in step 4 the disk write is also replicated across a WAN link to the backup site, to be preserved. Here the system behavior depends on the type of replication used. With sync replication, the system would wait for the replication to complete (i.e., for the acknowledgement from the remote site in step 7), before continuing processing (step 5) and responding to the client (step 6). With async replication the system would immediately continue with steps 5 and 6 after the DB write has succeeded locally, deferring the replication in step 4 for later. In contrast, with pipelined synchronous replication, the transfer in step 4 is performed immediately yet asynchronously, allowing the database to return its reply to the front tier server in step 5 concurrently. The front tier continues processing the request, for example combining the ticket information with a mashup of maps and hotel availability. Eventually, in step 6, the web tier produces a reply to return to Alice. In the pipelined synchrony case, this reply cannot be returned to the client until the database write it was based on has been persisted to the remote site. Only after step 7 completes and the remote server has acknowledged the write as complete can the reply to the client be released (step 8) and returned to Alice's web browser to show her the purchase confirmation (step 9).

The use of pipelined synchrony means that steps 5 and 6, which may include significant computation cost, can be performed in parallel with the propagation of the disk write to the backup site. This can provide a substantial performance gain compared to synchronous replication which must delay this processing for the length of a network round trip. Since Pipelined Synchrony defers replying to Alice until after the write is acknowledged in step 7, she is guaranteed that the data her reply is based on has been durably committed.

Thus the key challenge is to track which *durable* write requests, i.e., those that need to be persisted to the secondary site, are causally related (dependent) on which *externally-visible* network packets. In other words, Pipelined Synchrony replication must guarantee a causal ordering between externally-visible and durable events as defined by Lamport's *happened-before* $\rightarrow$ relation [13]: if any write request $\rightarrow$ a network packet, then the write must complete before the packet is released.

To intuitively understand how such dependencies can be tracked, first assume a global clock in the system. Further assume that every write is timestamped using this global clock. In this case, a total ordering of all events is obtained. Hence, if a network packet is generated at time t, then it is sufficient to hold this packet until

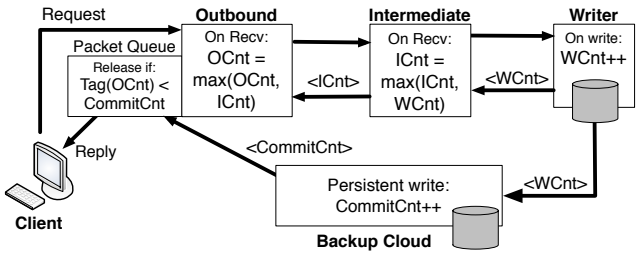


Figure 5: Multi-Tier pipelined replication. The reply to the client is buffered until the dependent write has been committed.

all disk writes that have a timestamp $\leq t$ have finished at the secondary site. Observe that not all of these writes are causally related to the network packet; however, by waiting for all previously issued writes to complete at the secondary, we ensure that all causally related writes will also finish, thereby ensuring safety.

In practice, a multi-tier application does not have the luxury of a global clock and techniques such as Lamport's logical clocks only yield a partial, rather than a total, ordering of events. Thus we must devise a scheme to perform this timestamping, using logical clocks, so as to identify and track causally dependent writes in multi-tier distributed applications. The problem is further complicated by the fact that our approach provides *black box* protection of VMs.

3.2 PipeCloud

In this section we present the design of PipeCloud, our pipelined-synchronous disaster recovery engine, while in § 4 we elaborate on the implementation details. As explained above, in a single VM application, the local machine clock may be used as a global clock, allowing a simple approach for tracking writes that are causally dependent on a network packet. This can be easily extended to support multi-tier applications with only a single protected writer, but becomes more complex for multi-tier multi-writer applications. We first explain the case of providing DR protection to a multi-tier application with a single writer in § 3.2.1, and then generalize to the multi-tier, multi-writer case in § 3.2.2.

3.2.1 Single Writer Protection

Our primary target is the classical multi-tier web service, e.g., an Apache, application, and database server setup (a.k.a. "LAMP" stack). Most services structured this way use web servers as frontends to serve static content, application servers to manipulate session-specific dynamic content, and a DB as a data backend. In this case, the only tier that is necessary to protect in our model is the DB. This benefits our approach because further work performed by the upper tiers can be overlapped with the replication of DB writes.

To protect application state while allowing computation to overlap, we must track which outbound network packets (externally-visible events) depend on specific storage writes that are made durable at a backup site. We assume that PipeCloud is running in the VMM of each physical server and is able to monitor all of the disk writes and network packets being produced by the VMs. PipeCloud must (i) replicate all disk writes to a backup server, (ii) track the order of disk writes at the primary site and the dependencies of any network interaction on such writes and (iii) prevent outbound network packets from being released until the local writes that preceded them have been committed to the backup.

The procedure described above relies on being able to propagate information about disk writes between tiers along with all communication, and is depicted in Figure 5. We divide the tiers along three roles, namely *writer*, *intermediate* and *outbound*; to simplify exposition, Figure 5 shows one intermediate and outbound tiers, but there could be multiple tiers assuming those roles. Each tier maintains its own logical *pending write counter*, i.e., $WCnt$, $ICnt$ and $OCnt$. Note that $WCnt$ represents the true count of writes performed to the disk, while $OCnt$ and $ICnt$ represent the (possibly outdated) view of the disk state by the other tiers. In addition, the backup maintains a committed write count, $CommitCnt$. These counters are essentially monotonically increasing logical clocks, and we use these terms interchangeably.

Without loss of generality, assume all counters are zero when the system receives the first client request, which propagates through the tiers, ultimately resulting in a DB update at the writer tier. The writer tier increases the value of its pending counter after it has performed a local write, and before that write is issued to the backup site. Each tier appends the current value of its pending count to all communication with other elements in the system. Thus the writer tier propagates its pending counter through the intermediate nodes so they can update their local view: the pending counter at each non-writer tier is updated to the maximum of its own pending clock, and any pending clocks it receives from other tiers.

On receiving the write request from the writer tier at the primary site, the DR system at the backup site will commit the write to disk and then increase its committed write counter. The current value of the committed counter is then communicated back to the outbound node(s) at the primary site.

The outbound tier implements a packet buffering mechanism, tagging packets destined to external clients with its own version of the pending counter, $OCnt$, as this represents the number of system writes which could have causally preceded the packet's creation. Packets can be released from this queue only when their pending clock tag is less than or equal to the committed clock received from the backup site. This guarantees that clients only receive a reply once the data it is dependent on has been saved. Finally, note that protection of the single VM case is covered by this scheme: the single VM becomes both a writer and outbound node.

3.2.2 Multiple Writer Protection

The most challenging case is when the application is in effect a distributed system: multiple nodes cooperate, and more than one of the nodes issue writes that need to be persisted. Examples include a LAMP stack with a master-master DB replication scheme or a NoSQL-style replicated key-value store.

We cater to this case by extending the notion of a logical counter to a count vector maintained by each node[1]. The pending write count for node i thus becomes the vector $P_i =< p_1, ..., p_n >$, with an entry for each of the n writers in the system. When issuing disk writes, node i increments its local counter in $P_i[i]$. All packets are tagged with the count vector of pending writes, and local knowledge is updated with arriving packets by merging the local and arriving vectors: each entry becomes the maximum of the existing and arriving entry. By exchanging information about writes in this way, the entry $P_i[j]$ indicates the number of writes started by node j that i is aware of. Thus at any given time, the pending count vector represents the *write frontier* for node i—the set of writes at other nodes and itself that any computation or network packet might be causally dependent on. Note that non-writer nodes never increment write counts, only merge on packet reception, and that the single-writer case equates to a single-entry count vector.

In this manner, causality spreads across the system in a gossiping

[1]Our count vector is similar to a vector clock [16], but vector clocks are traditionally updated on every message send or receive, while ours only count disk writes events, similar to [12].

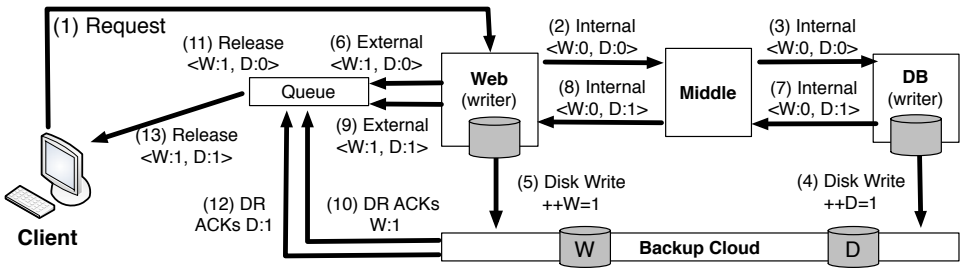


Figure 6: Multi-writer case with count vectors.

or anti-entropy manner [12, 23], all the way to outbound packets. As before, an outbound packet is tagged with the pending count vector at the moment of its creation. Outbound nodes maintain similar count vectors of committed writes $C_i =< c_1, ..., c_n >$, which represent the set of writes known to have been safely persisted. A client-bound packet can only be released once every entry in C_i is greater than or equal to that in the packet's tag. This approach allows for a partial ordering of unrelated writes and network packets, but it guarantees that no packet is released until any write it is causally related to has been committed.

Figure 6 illustrates a multi-writer system. The DB and Web tiers issue writes that are persisted to the secondary in steps 4 and 5, respectively. The web tier generates an outbound packet with no knowledge of the DB write (step 6), that is buffered. A causal chain of communication emanates all the way from the DB to another outbound buffered packet, in steps 7 to 9. Acknowledgement of write commits arrive out of order, with the web tier write arriving first (step 10) and thus allowing the outbound packet dependent on the web write (but not on the DB write) to leave the system in step 11. Finally, when the DB write is acknowledged in step 12, the packet buffered in step 9 leaves the system in step 13.

3.3 Other DR Considerations

We end the design section by enumerating some aspects of a full disaster recovery solution that lie outside the scope of this paper.

Detecting Failure: The usual approach to deciding that disaster has struck involves a keep-alive mechanism, in which the primary has to periodically respond to a ping message [15]. We note our system is no different from other DR solutions in this aspect.

Failure in the secondary: Failure of the secondary site, or a network partition, will impede propagation of data writes. Without a suitable remedial measure, externally-visible application response will be stalled waiting for replies. PipeCloud is no different from sync replication in this aspect: a reverse keep-alive is usually employed to trigger a fallback to async replication.

Memory Protection: We protect only the application state recorded to disk at the primary site. We assume that applications can be fully restored from disk in the case of disruption, as in standard database behavior. We do not attempt to protect the memory state of applications as this entails a significant overhead in WANs. Remus [6] is able to provide memory and disk protection within a LAN, but requires significant bandwidth and minimal latency. Our experiments with Remus in an emulated WAN environment with 100ms of RTT from primary to backup, show that average TPC-W response times exceeded ten seconds and replication of both disk and memory consumed over 680 Mbps of bandwidth. Providing Remus-like black-box memory protection over WAN remains an open problem due to these performance issues.

Transparent handoff and RTO: Our focus is on ensuring that storage is mirrored to the backup site and minimizing RPO. As shown before [26], when disaster strikes our backup system uses existing VM automation techniques to rapidly instantiate compute resources and recover application state. Enabling seamless network redirection to the secondary site requires network virtualization mechanisms such as NAT, MobileIP, or MPLS VPNs. Because our storage replication is crash-consistent, disk checking utilities like `fsck` or self-consistent file systems like ZFS must be used.

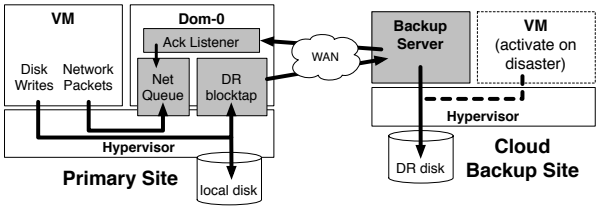


Figure 7: PipeCloud's implementation components.

4. IMPLEMENTATION

Our PipeCloud prototype is split between a replication-aware virtualization system that is run at the primary site and a backup server at the cloud backup location. PipeCloud requires modifications to the virtualization platform at the primary, but only needs to run a simple user space application at the backup. This allows PipeCloud to be used today on commercial clouds which do not give users control over the low level platform.[2]

At the primary site, PipeCloud is based on the Xen Virtual Machine Monitor (VMM) version 4.0.0. VMs in Xen perform IO using a split driver architecture with components in the guest operating system (OS) and *dom0*, a privileged domain that runs hardware drivers and a control stack. Frontend drivers in the VM's OS issue requests through shared memory and virtual interrupt lines to backend drivers. The backend drivers in dom0 unpack the requests and re-issue them against the hardware drivers. As depicted in Figure 7, our implementation only requires hooks at the level of virtual backend drivers; *NetQueue* and *DR Blocktap* in the figure. While we chose Xen due to familiarity, we do not foresee any problems porting our approach to VMMs like kvm or VMware ESX. While the majority of our code is new extensions to the base Xen distribution, we also utilize the packet queueing mechanism from Remus (the sch_queue kernel module) to simplify our implementation of network buffering [6]. The Remus code holds and releases groups

[2]Note, however, that current clouds do not yet support mechanisms that would facilitate seamless fail-back to the original site after the disaster has passed. Without support for migration *out* of the cloud, the fail-back procedure would need to be scheduled for a time that minimizes the impact of application downtime.

of packets for each checkpoint interval; we have extended this to support finer grain control based on our logical clocks.

In keeping with the goal of a black-box approach, we do not mandate source code changes within protected VMs. However, we benefit from information regarding application deployment; we term this *configuration gray box*. We need a specification of the topology of the multi-tier application being persisted: which VMs make up the system, identified by their IP or MAC addresses; which VMs (or virtual disks) need to be persisted, e.g., the data disk of the DB VMs; and which nodes are allowed to perform outbound communications, e.g., the load-balancer gateway, or the Apache pool. We expect that VMs for which storage needs to be persisted will be backed by two sets of virtual disks: one storing critical application data, such as a DB bit store; and the other backing temporary files, and other miscellaneous non-critical data. Given our black box nature, this setup alleviates replication overhead (and noise) by differentiating critical disk writes which must be preserved to the backup site. All the information we require is deployment-specific, and provided by the sys-admin who configured the installation (as opposed to a developer modifying the code).

We structure this section following the nomenclature described in § 3.2.1. We first describe the implementation details for *writer* nodes, then *intermediate* and finish with *outbound* nodes. We close with a discussion of the secondary site backup component.

4.1 Replicating Disks – VM Side

To track disk writes and replicate them to the backup server, PipeCloud uses a new virtual disk driver backend which we dub *DR Blocktap*. This is a user-space dom0 daemon utilizing the blocktap Xen infrastructure. As a VM performs reads or writes to its protected disk, the requests are passed to our disk driver. Read requests are processed as usual.

Writes, however, are demultiplexed: they are issued both to local storage and sent through a socket to the remote site. After issuing each write, the local logical clock of pending writes, maintained as a kernel data structure in dom0, is increased. Local writes are then performed with caching disabled, so requests do not return until DMA by the local hardware driver has succeeded. At this point, we indicate to the VM that the write has completed, regardless of the status of the write traveling to the secondary site.

We note that typical OS behavior (Windows or UNIX) consists of issuing multiple writes simultaneously to leverage scatter-gather DMA capabilities. There are no expectations about ordering of writes in hardware, and a successful response is propagated upstream only after all writes in the batch have succeeded. In the absence of disk synchronization barriers (a hardware primitive that is not yet supported in Xen virtual drivers), the OS achieves `sync()`-like behavior by waiting for the batch to finish. We tag each individual write in a batch with its own value, and thus need to respect write ordering, when processing backup acknowledgements, to maintain the expected `sync()` semantics.

4.2 Propagating Causality

In order to track causality as it spreads throughout a multi-tier system, we need to propagate information about disk writes between tiers along with all communication. Our implementation does this by injecting the value of the local logical clock into packet headers of inter-VM communication, specifically through the addition of an IP header option in IP packets.

Virtual networking in Xen is achieved by creating a network interface in dom0. This interface injects in the dom0 network stack replicas of the Ethernet frames emanating from the VM. It replicates a frame by first copying to the dom0 address space the Ethernet, IP and TCP headers. By copying these bytes, dom0 can now modify headers (e.g., to realize NAT or similar functionality). The remainder of the packet is constructed by mapping the relevant VM memory pages read-only.

For our purposes, we split the copying of the header bytes right at the point of insertion of an IP header option. We construct our option, relying on an unused IP header option ID number (`0xb`). The IP header option payload simply contains the logical clock. We then copy the remaining header bytes. We do not introduce any extra copying, mapping, or reallocation overhead when expanding packets in this way. We modify length fields in the IP header, and recalculate checksums. Typically, the VM OS offloads checksum calculations to "hardware", which in reality is the backend virtual driver. Therefore, we do not incur additional overhead in the data path by computing checksums at this point.

Not all packets are tagged with the logical clock. Non-IP packets are ignored; in practice, non-IP packets (ARP, STP, etc) do not represent application-visible messages. Packets that already contain an IP header option are not tagged. This was done to diminish implementation complexity, but is not a hard constraint – in practice, we do not frequently see packets containing an IP header option. Optionally, empty TCP segments may not be tagged. These typically refer to empty TCP ACKS, which do not affect causality because they do not represent application-visible events. Nagle's algorithm and large bandwidth-delay products mean that success of a `write()` syscall on a TCP socket only guarantees having copied the data into an OS buffer. Additionally, most application protocols include their own application-level ACKs (e.g. HTTP 200). Empty TCP segments with the SYN, FIN, or RST flags, which do result in application visible events, are tagged.

We also considered using a "tracer" packet to communicate logical clock updates to other tiers. We ruled this out primarily due to the need to ensure correct and in-order delivery of the tracer before any other packets are allowed to proceed. Our approach, instead, does not introduce new packets, is resilient to re-ordering since logical clocks are only allowed to increase, and ensures that as long as existing packets are delivered, logical clock updates will propagate between tiers. We could not observe measurable overhead in latency ping tests, or throughput netperf tests.

Our current space overhead is 20 bytes per packet, with 16 bytes dedicated to the logical clock in the IP header option. In the case of vector clocks for multi-writer systems, this limits the size of the vector to four 32 bit entries. This is not a hard limit, although accommodating hundreds of entries would result in little useful payload per packet, and a noticeable bandwidth overhead.

4.3 Buffering Network Packets

Outbound nodes maintain two local logical clocks. The clock of pending writes is updated by either (or both of) the issuing disk writes or propagation of causality through internal network packets. The second clock of *committed* writes is updated by acknowledgments from the DR backup site. Comparing the two clocks allows us to determine if a network packet produced by the VM can be released or if it must be temporarily buffered.

We achieve this by appending a queueing discipline to the network backend driver (*NetQueue*), which tags outbound packets with the current pending write clock. Because logical clocks increase monotonically, packets can be added at the back of the queue and taken from the front without the need for sorting. Packets are dequeued as updates to the committed logical clock are received.

We make no assumptions on the behavior of disk IO at the secondary site, and thus need to consider that write completion may be acknowledged out of order – this is particularly relevant given our

previous discussion on `sync()`-like behavior. Out-of-sequence acknowledgements are thus not acted upon until all intermediate acks arrive.

For vector clocks with two or more entries, we use a set of cascading queues, one for each entry in the vector. As different entries in the committed vector clock are updated, the affected packets at the front of the corresponding queue are dequeued and inserted in the next queue in which they have to block. Insertion uses binary search to maintain queue ordering. Once popped from the last queue, packets leave the system.

4.4 Storage in the Cloud Backup Site

At the secondary site, a Backup Server collects all incoming disk writes, commits them to a storage volume which can be used as the disk of a recovery VM if there is a disaster, and acknowledges write commits to the primary site. The Backup Server is a user level process, and thus does not require any special privileges on the backup site; our evaluation demonstrates how we have deployed PipeCloud using the Amazon Elastic Compute Cloud and Elastic Block Store services.

When there is only a single disk to be protected, the Backup Server performs writes directly to the backup storage volume. These writes should be performed in a durable manner which flushes them past the OS level cache. For the multi-writer case, a single Backup Server receives write streams from multiple protected VMs. Unlike the single disk case, this means that writes from multiple disks must be preserved respecting total ordering, if possible. Without special handling, the WAN link, the kernel and the disk controller at the Backup Server may all reorder writes. If a failure occurred, this could result in a write being preserved without causally-precedent writes having survived. While EMC Consistency Groups [7] can create synchronized point-in-time replicas of a set of disks, our approach achieves causally synchronized replicas.

To avoid this problem, we use the vector clocks maintained at each primary server as a guide for how writes to the backup disks should be ordered. When a primary sends a disk write to the backup, it includes its current vector clock. Ordering of the vector clocks indicates causality precedence and allows the Backup Server to enforce the same ordering in its writes. Without having application-level hints, we allow any ordering for writes which are considered to have concurrent vector clocks.

In the ideal case, the Backup Server maintains dedicated hardware (an SSD or a separate log-structured rotating disk) to initially record the writes it receives, with metadata about each write prepending the data. Writes are thus persisted, and acknowledgements returned to the primary, with minimal latency. A separate consumer process then opportunistically transfers the writes to the actual VM disk volumes. The vector clock-based ordering of writes among tiers is performed at this later stage, outside of the critical acknowledgement path. While we have not implemented this, we can preserve a causal history of each disk by retaining all writes with their logical clocks. All storage volumes are backed by RAID or hardware with similarly strong durability guarantees.

Unfortunately, the reality of cloud storage is removed from these requirements. Services such as Amazon's EBS, do not offer many necessary guarantees. There is no user control to ensure disk writes are uncached, committed in-order, and have actually moved from memory to the disk. In the event of infrastructure outages, there are loose availability guarantees [20]—simultaneous use of multiple providers has been proposed to mitigate this [3, 5]. Virtualization hides the details of hardware features which are relevant to the performance of a backup server, such as availability of SSDs, or physical layout for log-structured partitions.

Recent events indicate a shift in cloud providers toward providing SLAs in their storage services [20]. This could be offered as a differential service, and would represent the ideal substrate for cloud-based DR. We believe DR makes for an extremely compelling case to move forward on storage durability and availability guarantees. SLAs already in place by many providers [2], point toward widely realizing these basic primitives in the short term.

5. BLACK-BOX CAUSALITY AND RPO

Guaranteeing that clients will experience the same recovery semantics upon disaster as with synchronous replication is closely tied to our ability to introspect causality relations on a black-box VM. We start analyzing the guarantees we provide, and our limitations, with this basic building block.

As argued earlier, in a single-VM system, we can use the local machine clock as a "global" clock to timestamp writes and network packets and derive a total ordering of events; in this case, holding a packet until all writes with timestamps lower or equal to the packet is sufficient to ensure all causally dependent writes have finished. Next consider the multi-VM single-writer scenario.

LEMMA 1. *In a multi-VM single writer system, it is sufficient to hold a network packet until the commit count at the secondary becomes greater than or equal to the local counter value at the outbound node.*

Proof Sketch: At the writer node, tagging internal messages with the local write counter captures *all* writes that were issued prior to sending out this internal message (and thus, all causally dependent writes as well). As shown in Figure 5, each node computes the max of its local counter and the one on the arriving message; since the counter at other nodes lag the writer node, doing so propagates the counter value from the writer node. At the outbound node, holding the network packet until writes committed by the secondary exceed this counter ensures that all causally dependent writes issued by the writer node have been completed at the secondary. □

Our PipeCloud implementation uses three additional mechanisms to ensure that this property holds in practice even in multi-core environments: (i) the Xen backend driver executes in a serialized manner for a given virtual device (ii) we limit VMs to a single network interface and a single protected block device (iii) the clock is updated with atomic barrier instructions.

Finally consider the general multi-VM multi-writer case. Here we resort to using count vectors that track a "write frontier" of causally dependent writes at each writer. Like in the single writer case, the dependencies are propagated by piggybacking count vectors on internal messages. Upon message receipt, a node computes the max. of the local and piggybacked count vector thereby capturing the union of all causally dependent writes that need to be tracked. Thus, the following applies:

LEMMA 2. *In a multi-writer system, it is sufficient for PipeCloud to release a network packet once all writes with a count vector less than or equal that of the network packet have finished.*

Proof Sketch: The count vector, m, that an outgoing message is tagged with, represents the events at each writer node that happened-before the packet was created. If $m[i] = k$, then all writes up to k at writer node i must have causally preceded packet m. Likewise, any write with value greater than k at node i is considered to have happened concurrently (or after) packet m. Hence, holding the packet until all writes in m have finished on all machines ensures that all causally related writes complete before network output becomes visible. □

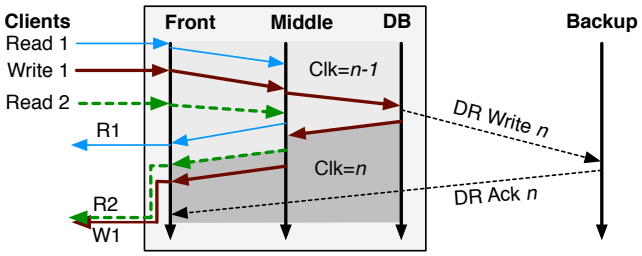
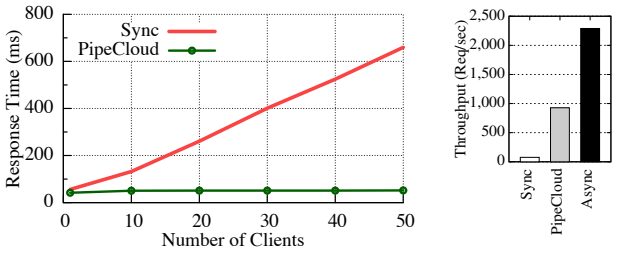


Figure 8: Pipelined Synchrony determines dependency information from communication between tiers. The replies R2 and W1 must be buffered because they are preceded by the database write, but reply R1 can be sent immediately.



(a) MySQL Response Time (b) Throughput

Figure 9: Clients must wait for multiple WAN delays with Sync, but PipeCloud has a consistent response time just barely over the RTT of 50 ms. By reducing the time DB tables must be locked for each transaction, PipeCloud is able to provide a much higher throughput relative to Sync.

Armed with these conclusions, we revisit client RPO guarantees from § 2.2. First, the Secondary view (SV) remains a subset of the Primary (PV): writes are propagated to the secondary after issuance in the primary. Second, and most importantly, the Client View (CV) remains a subset of the SV: the CV is only updated when client-bound packets are released. Thus, PipeCloud is no worse than sync replication, and therefore yields a client RPO of zero. Per lemma 2, clients will not receive updates that are not contained in the secondary site, and therefore no inconsistencies will arise after disaster recovery. Clients thus perceive synchronous replication and PipeCloud as indistinguishable.

One limitation of our black-box causality tracking is that our approach conservatively marks all writes issued before an outgoing message as dependent; while this set of writes contains all causally dependent writes, it may include other independent writes as well. Since our black-box system has no application visibility, we are unable to discern between dependent and independent writes, requiring us to conservatively mark all prior writes as dependent for safety. To illustrate, consider two separate application threads processing requests on different data items. PipeCloud cannot differentiate between these threads and, as a result, may conservatively mark the network packets from one thread as being dependent on the writes from the other thread which happened to precede them, regardless of actual application level dependence. This is illustrated in Figure 8: while both reads are independent from "Write 1", "Read 2" is affected by communication between the middle and DB tier after "Write 1" is propagated to the secondary. As a result, the reply to "Read 2" is unnecessarily delayed.

6. EVALUATION

We have evaluated the performance of PipeCloud under normal operating conditions, as well as its failure properties, using both a local testbed and resources from Amazon EC2. On our local testbed, we use a set of Dell servers with quad core Intel Xeon 2.12GHz CPUs and 4GiB of RAM. The servers are connected by a LAN, but we use the Linux *tc* tool to emulate network latency between the hosts; we have found that this provides a reliable WAN network emulation of 50 or 100 ms delays. Our EC2 experiments use "Large" virtual machine instances (having two 64-bit cores and 7.5GiB of memory) in the US East region. Virtual machines in both the local testbed and EC2 use CentOS 5.1, Tomcat 5.5, MySQL 5.0.45, and Apache 2.23. We compare three replication tools: DRBD 8.3.8 in synchronous mode, our pipelined synchrony implementation, and an asynchronous replication tool based on our pipelined synchrony but without any network buffering.

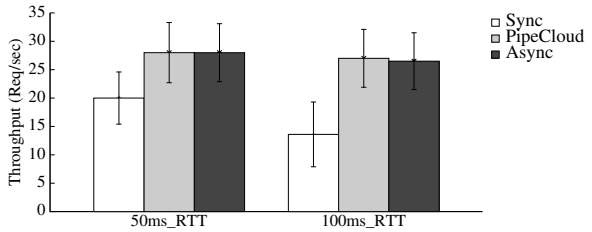
Our evaluation focus is on client-server applications and we consider three test applications. 1) We use the **MySQL** database either by itself or as part of a larger web application. We use the InnoDB storage engine to ensure that database writes are committed to disk before replying to clients, and we store all of the InnoDB data and log files on a protected disk partition. In single VM experiments, we communicate directly with the MySQL database via a Java application running on an external client machine. 2) **TPC-W** is an e-commerce web benchmark that emulates an online bookstore. TPC-W is composed of two tiers, a Tomcat application server and a MySQL database, that each run in separate virtual machines; we only protect the disk used for the database files. TPC-W includes a client workload generator which we run on a server which is considered external to the protected system. 3) We have also written a PHP based web application called **CompDB** which allows us to more precisely control the amount of computation and database queries performed when processing requests. The application can be deployed across one, two, or three tiers, each of which runs an Apache server and a MySQL database. Requests are generated by the *httperf* tool and access a PHP script on the front-end server which performs a configurable amount of computation and insert queries to the local database before being propagated to the next tier which repeats the process. Together, these applications allow us to emulate realistic application workloads and perform controlled analyses of different workload factors.

6.1 Single Writer Database Performance

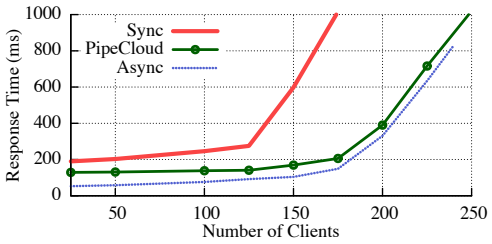
We first measure the performance of PipeCloud when protecting the disk of a MySQL database. We imposed a 50ms RTT from primary to backup in our local testbed; this limits the minimum possible response time to 50ms for the non-asynchronous approaches.

Figure 9(a) shows how the response time changes as the client load on the database increases. Each client connects to the database server and repeatedly inserts small 8 byte records into a table. Since the protected application is a database that must ensure consistency, the table must be locked for each individual transaction so they can be performed serially. With Sync, the table must be locked for at least one RTT because a transaction cannot complete until the remote disk acknowledges the write being finished. This means that when a client request arrives, it must wait for a round trip delay for each pending request that arrived before it. This causes the response time, when using Sync, to increase linearly with a slope based on the round trip delay to the backup.

In PipeCloud, the database table only needs to be locked until the local disk write completes, allowing for much faster request processing. This results in a much lower response time and higher throughput because many requests can be processed during each

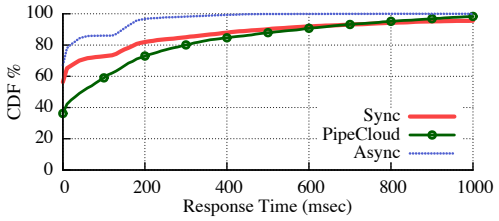


(a) TPC-W Maximum Throughput

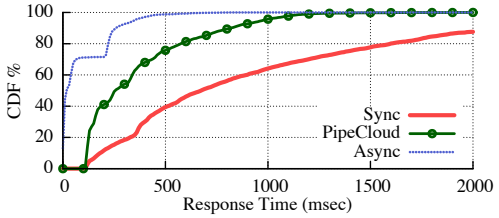


(b) TPC-W Response Time (50 ms RTT)

Figure 10: PipeCloud has higher TPC-W throughput than synchronous, and performs almost equivalently to an asynchronous approach when there is a 50 ms round trip delay.



(a) Read Requests



(b) Write Requests

Figure 11: With 100 ms RTT, the black-box network buffering in PipeCloud causes some read-only requests to have higher response times, but it provides a significant performance improvement for write requests compared to synchronous.

round trip delay to the backup. Figure 9(b) shows that PipeCloud achieves a maximum throughput over twelve times higher than Sync. While using an asynchronous approach may allow for an even higher throughput, PipeCloud provides what asynchronous approaches cannot: *zero data loss*.

6.2 Multi-Tier TPC-W Performance

We use PipeCloud to protect a set of virtual machines running the TPC-W online store web benchmark to see the performance of a realistic application with a mixed read/write workload.

We first measure the overall performance of TPC-W as we vary the latency between the primary and backup site when using different replication mechanisms. Figure 10(a) shows the maximum throughput achieved by the different replication schemes. PipeCloud's maximum throughput is nearly identical to that of an asynchronous scheme—the ability to pipeline request processing and state replication effectively masks the overhead of disaster recovery. When the round trip delay increases to 100 ms, the throughput of synchronous drops even further, but PipeCloud's performance is effectively unaffected. PipeCloud is able to maintain a throughput two times better than the synchronous approach.

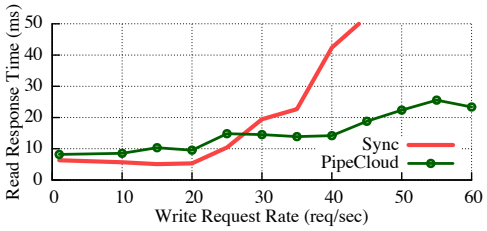
In Figure 10(b) we see that PipeCloud's pipelining also reduces response times compared to a synchronous approach, even for relatively low client loads where the throughput of each approach is similar. The load-response time curve for PipeCloud closely follows the asynchronous approach, offering a substantial performance benefit compared to synchronous and the same level of consistency guarantees.

We next categorize the request types of the TPC-W application into those which involve writes to the database and those which are read-only. The workload contains a mix of 60% reads and 40% writes, and we measure the response times for each category. Figure 11(a) shows a CDF of the response times for read-only requests when there are 50 active clients and there is a 100 ms roundtrip time to the backup. PipeCloud has a slightly higher base response time because some read-only requests are processed concurrently with requests which involve writes. Since PipeCloud cannot distinguish between the packets related to read-only or write requests, it must conservatively buffer both types. However, even with some requests being unnecessarily delayed, PipeCloud's overall performance for reads is very close to synchronous DRBD.

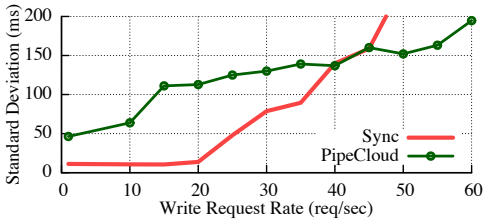
PipeCloud's greatest strength shows when we observe the response time of requests that involve at least one database write in Figure 11(b). PipeCloud's ability to overlap work with network delays decreases the median response time by 50%, from over 600 ms to less than 300 ms. Only 3% of requests to PipeCloud take longer than one second; with synchronous replication that rises nearly 40% . This improved performance allows PipeCloud to be used with much more stringent performance SLAs.

6.3 Impact of Read and Write Rates

This experiment explores how the network buffering in PipeCloud can unnecessarily delay read-only requests that are processed concurrently with writes. We use our CompDB web application in a single-VM setup and send a constant stream of 100 read requests per second as well as a variable stream of write requests that insert records into a protected database. The read requests return static data while the writes cause a record to be inserted to the database. There is a 50 ms RTT between primary and backup. Figure 12(a) shows how the performance of read requests is impacted by the writes. When there is a very low write request rate, the response time of Sync and PipeCloud are very similar, but as the write rate rises, PipeCloud sees more read-only packets being delayed. However, the increased write rate also has a performance impact on the read requests in Sync because the system quickly becomes overloaded. PipeCloud is able to support a much higher write workload and still provide responses to read requests within a reasonable time as reflected by a lower average response time for PipeCloud under load. However, PipeCloud's network buffering impacts those read requests which happen to coincide with write requests, resulting in a higher standard deviation in the response time as shown in Figure 12(b). We believe that the trade-off provided by PipeCloud is a desirable one for application designers: a small reduction in read performance at low request rates is balanced by a significant reduction in write response times and support for higher overall throughput.



(a) Average Response Time



(b) Response Time Standard Deviation

Figure 12: PipeCloud's black box network buffering causes read requests to initially see higher delay and greater variance than Sync, but pipelining supports a much larger write workload than Sync.

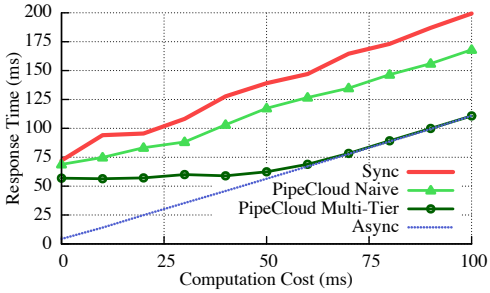


Figure 13: PipeCloud continues processing as writes are sent to the backup site, allowing it to provide equivalent performance to asynchronous replication if there is sufficient work to do.

6.4 Multi-tier Sensitivity Analysis

To verify PipeCloud's ability to hide replication latency by overlapping it with useful work, we performed an experiment in which we arbitrarily adjust the amount of computation in a multi-tier server. We use the CompDB application split into two tiers; the front tier performs a controlled amount of computation and the backend inserts a record into a database. We also compare PipeCloud against a naïve version that only applies pipelining to the DB tier. Only an unsafe asynchronous approach is able to provide a response time faster than the round trip delay of 50ms.

Figure 13 shows how the average response time changes as a function of the controlled amount of computation. Naïve PipeCloud has only a slight performance gain over synchronous because it can overlap only the DB's work with its own writes. The computation performed by the front tier cannot be pipelined since the boundary separating internal and external events has naïvely divided the two tiers, enforcing a stricter than necessary serial ordering.

However, when PipeCloud is applied jointly across the two tiers, it is able to perform processing from multiple tiers concurrently with replication, essentially providing up to 50 ms of "free computation". For requests that require more processing than the round trip time, PipeCloud provides the same response time as an asynchronous approach, with the advantage of much stricter client RPO guarantees.

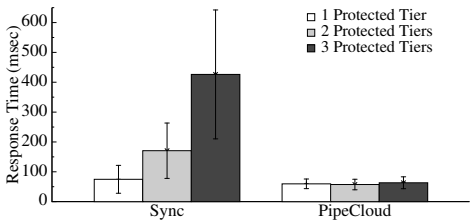


Figure 14: Each tier protected with synchronous replication increases response time by at least one RTT. Pipelining the replication of writes provides a much lower response time.

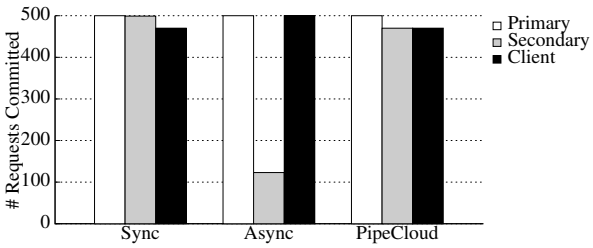


Figure 15: Primary, secondary, and client view of an application state when a disaster strikes after 500 MySQL updates. Both sync- and pipelined sync-based approaches guarantee that the client view is bounded by the state of the secondary.

6.5 Protecting Multiple Databases

With current approaches, often only a single tier of an application is protected with DR because it is too expensive in terms of cost and performance to replicate the state of multiple application tiers. To evaluate PipeCloud's support for multiple servers with protected storage we consider a 3-tier deployment of our CompDB application configured so each tier includes both a web and database component. Figure 14 shows the average response time of requests to this application when either one, two, or all three of the tiers are protected by a DR system. There is a 50 ms RTT, and we use a single client in order to provide a best case response time. With synchronous replication, the response time increases by more than a round trip delay for every tier protected since the writes performed at each tier must be replicated and acknowledged serially.

PipeCloud on the other hand, is able to pipeline the replication processes across tiers, providing both better overall performance and only a minimal performance change when protecting additional tiers. When protecting all three tiers, PipeCloud reduces the response time from 426 ms to only 63 ms, a 6.7 times reduction. Being able to pipeline the replication of multiple application components allows PipeCloud to offer zero data loss guarantees to applications which previously would have resorted to asynchronous replication approaches simply due to the unacceptable performance cost incurred by serial, synchronous replication.

6.6 Failure evaluation

6.6.1 Comparing replication strategies

PipeCloud seeks to provide performance on-par with asynchronous, but it also seeks to assure clients of the same consistency guarantee as a synchronous approach. Figure 15 compares the consistency

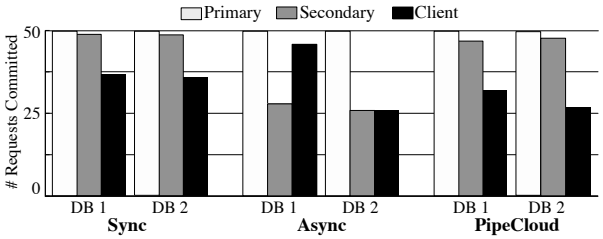


Figure 16: When protecting multiple databases, PipeCloud still guarantees that clients do not receive replies unless data has been durably committed to the backup.

guarantees provided by Sync, Async, and Pipelined Synchrony. As in §6.1, a single MySQL database is backed up to a secondary site 50 msecs away. Out of a set of 500 database inserts that occur immediately prior to a disaster, we examine the number of records recorded at the primary and secondary sites and the number of confirmations received by the client. Sync guarantees that the secondary view (SV) is almost identical to the primary (PV), but allows the client (CV) to lag behind the secondary. With Async, the client is not limited by the rate of confirmations from the secondary site, causing the client to receive many unsafe replies (CV > SV). However, with Pipelined synchrony, these unsafe replies do not occur because network packets are buffered until the secondary's acks are received; as a result, PipeCloud is able to provide clients with the same guarantee as synchronous—the data for any response they receive will always have been safely committed to the backup site.

6.6.2 Recovering to EC2

Next we emulate a disaster scenario in which the secondary site takes over request processing for a CompDB stack, configured with a frontend node that performs database insertions on a master-master database split between two backend nodes. We run the primary site within our local testbed and use Amazon EC2 VMs to run both the Backup and the failover servers. The network latency measured from our primary site in western Massachusetts to the backup server in EC2's northern Virginia site was 16 ms. We use EBS volumes for the two protected DB disks. Prior to the failure, a single EC2 VM acting as the Backup Server applies the write streams received from both of the protected VMs to the EBS volumes.

Upon failure, the Backup Server disables replication and uses the EC2 API to detach the EBS volumes. It then reattaches the two backup volumes to a set of new cloud instances. During bootup, the database VMs perform a consistency check by launching the `mysqld` process. Once this is complete, the application resumes processing requests. The table below details the time (in seconds) required for each of these steps; in total, the time from detection until the application is active and ready to process requests took under two minutes.

	Detach	Reattach	Boot	**Total**
Time (s)	27	13	75	**115**

Figure 16 shows the consistency views for the last fifty requests sent to each of the DB masters prior to the failure. As in the single writer case described previously, PipeCloud's multi-writer DR system is able to provide the consistency guarantee that client view will never exceed what is safely replicated to the secondary.

This experiment illustrates the potential for automating the recovery process using cloud platforms such as EC2. The API tools provided by such clouds automate steps such as provisioning servers and reattaching storage. While the potential to reduce recovery time using cloud automation is very desirable, it remains to be seen if commercial cloud platforms can provide the availability and durability required for a disaster recovery site.

7. RELATED WORK

Disaster Recovery is a key business functionality and a widely researched field. We have previously mentioned the work by Keeton and Wilkes on treating asynchronous replication as an optimization problem, and balancing the two primary concerns: financial objectives and RPO deltas [10]. In Seneca [9] the space of asynchronous replication of storage devices is studied, with a focus on optimizations such as write and acknowledgment buffering and coalescing. A similar study is carried out, although at the file system level, in SnapMirror [19]. Recent studies show a large potential for high yield of write coalescing in desktop workloads [21]. We have not directly leveraged the insights of write coalescing in our work due to our stringent zero-RPO objectives.

Another area of interest for storage durability is achieving survivability by architecting data distribution schemes: combinations of striping, replication, and erasure-coding are used to store data on multiple sites [27]. These concerns become paramount in cloud storage, in which the durability and availability guarantees of providers are, to say the least, soft. Two recent systems attack these deficiencies by leveraging multiple cloud providers: DepSky [3] provides privacy through cryptography, erasure coding and secret sharing, while Skute [5] aims to optimize the cost/response time tradeoff. We highlight that while availability and durability are increased, so is latency – unfortunately. Replication chaining techniques previously used in industry [8] may complement these techniques and ameliorate the latency overhead, possibly at much higher cost. Error correcting codes within a file system's replication stream have also been proposed to improve reliability despite network loss [24].

A long tradition of distributed systems research underpins our work. We use logical clocks to track causality, originally introduced by Leslie Lamport [13]. Further, we use techniques traditionally associated with eventual consistency [4, 12] to enforce pipelined synchronous replication throughout the nodes that make up a distributed service. We employ a vector clock-style approach [16] to allow each node to represent its knowledge of the system at the moment of producing data, and we allow nodes to propagate their knowledge to peers, as in anti-entropy or gossiping protocols [23]. Previous work in inferring causality in a distributed systems of black-boxes [1] focused on performance diagnosis as opposed to consistency enforcement during replication.

The concept of speculative execution has been used to reduce the impact of latency in a variety of domains [25, 18]. These approaches typically require application support for rolling back state if speculation must be cancelled. Pipelined synchrony also uses speculative execution, but it must cancel speculated work only if the primary fails; since it is the primary which performs the speculation, the roll back process is implicit in the failover to the secondary site. This allows PipeCloud to perform speculation and rollback in a black box manner without requiring any special support from clients, nor the protected application or OS. The concept of combining asynchronous replication with causality tracking was introduced by Strom and Yemini [22] to build a highly available distributed system that could recover from failure using rollback and message replay. These ideas are also used in the external synchrony work [18], which shows that in many cases the benefits of synchronous and asynchronous IO can be simultaneously reaped by intelligently overlapping IO with processing. Their treatment is focused on file system activity in a single host, and requires operating system support for tracking dependencies between processes.

We draw inspiration from Remus [6], which implements VM lockstep replication for LAN-based fault tolerance, and we make use of its network buffering code in our implementation. The mechanics of Remus rely heavily on low latency between nodes, allowing for replication of a VM's memory in addition to its disk. While this allows for complete protection of the VM's live state, our experience suggests memory propagation becomes too costly in terms of both bandwidth and performance overhead when network latency is high. Some of these issues have been resolved by RemusDB [17], which optimizes the replication process for virtual machines running database systems. However, this is no longer a completely application agnostic approach, which is one of our goals. We note though that many of these optimizations may still prove useful for replication in WAN environments. Overall, our work expands the concepts of external synchrony [18] and Remus-style black-box protection of virtual machines [6] to support WAN replication, disaster recovery, and multi-tier causality tracking.

8. CONCLUSION

Cloud computing platforms are desirable to use as backup locations for Disaster Recovery due to their low cost. However, the high latency between enterprise and cloud data centers can lead to unacceptable performance when using existing replication techniques. Our pipelined synchronous replication overcomes the deleterious effects of speed-of-light delays by overlapping or pipelining computation with replication to a cloud site over long distance WAN links. Pipelined synchrony offers the much sought after goal: performance of asynchronous replication with the same guarantees to clients as synchronous replication. It does so by ensuring network packets destined for external entities are only released once the disk writes they are dependent on have been committed at both the primary and the backup. Our evaluation of PipeCloud demonstrates dramatic performance benefits over synchronous replication both in throughput and response time for a variety of workloads. MySQL database throughput goes up by more than an order of magnitude and the median response time for the TPC-W web application drops by a half. Recreating failures also shows PipeCloud delivers on the promise of high performance coupled with the proper consistency in the client's view of storage.

Acknowledgements: We would like to thank our reviewers for their insightful comments. We also thank Brendan Cully for his assistance in configuring and running Remus during the early stages of this project. This work was supported in part by NSF grants CNS-0916972, CNS-0720616, CNS-0834243, OCI-1032765 and by AT&T.

9. REFERENCES

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proc. of SOSP*, 2003.

[2] AT&T. Synaptic Storage as a Service. `http://bit.ly/nt1JS0`.

[3] A. Bessani, M. Correia, B. Quaresma, F. Andre, and P. Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. In *Proc. of Eurosys*, 2011.

[4] A. D. Birrell, R. Levin, M. D. Schroeder, and R. M. Needham. Grapevine: an Exercise in Distributed Computing. *Communications of the ACM*, 25(4), 1982.

[5] N. Bonvin, T. Papaioannou, and K. Aberer. A Self-Organized, Fault-Tolerant and Scalable Replication scheme for Cloud Storage. In *Proc. of SOCC*, 2010.

[6] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proc. of NSDI*, 2008.

[7] EMC. EMC CLARiiON Storage Solutions. `http://bit.ly/oBNNel`.

[8] EMC. Symmetrix Remote Data Facility (SRDF) Product Guide. `http://scr.bi/oMVxIA`.

[9] M. Ji, A. Veitch, and J. Wilkes. Seneca: Remote Mirroring Done Write. In *Proc. of Usenix ATC*, 2003.

[10] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. Designing for Disasters. In *Proc. of FAST*, 2004.

[11] D. C. Knowledge. New York "Donut" Boosts NJ Data Centers. `http://bit.ly/1dzdau`.

[12] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. *ACM TOCS*, 10(4), 1992.

[13] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. of the ACM*, 21(7), 1978.

[14] Linbit. DR:BD Software Development for High Availability Clusters. http://drbd.org.

[15] Linux-HA. Heartbeat. http://linux-ha.org/wiki/Heartbeat.

[16] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*, 1989.

[17] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield. Remusdb: Transparent high availability for database systems. *Proc. of VLDB*, 2011.

[18] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the Sync. In *Proc. of OSDI*, 2006.

[19] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. SnapMirror: File System Based Asynchronous Mirroring for Disaster Recovery. In *Proc. of FAST*, Monterey, CA, Jan. 2002.

[20] Rightscale. Amazon EC2 Outage: Summary and Lessons Learned. `http://bit.ly/mhFvKY`.

[21] M. Shamma, D. T. Meyer, J. Wires, M. Ivanova, N. C. Hutchinson, and A. Warfield. Capo: Recapitulating Storage for Virtual Desktops. In *Proc. of FAST*, 2011.

[22] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3:204–226, 1985.

[23] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. of SOSP*, Copper Mountain, CO, Dec. 1995.

[24] H. Weatherspoon, L. Ganesh, T. Marian, M. Balakrishnan, and K. Birman. Smoke and mirrors: reflecting files at a geographically remote location without loss of performance. In *Proc. of the FAST*, 2009.

[25] B. Wester, J. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. Tolerating latency in replicated state machines through client speculation. In *Proceedings of NSDI*, Berkeley, CA, USA, 2009. USENIX Association.

[26] T. Wood, E. Cecchet, K. Ramakrishnan, P. Shenoy, and J. Van der Merwe. Disaster Recovery as a Cloud Service: Economic Benefits & Deployment Challenges. In *Proc. of HotCloud*, Boston, MA, June 2010.

[27] J. J. Wylie, M. Bakkaloglu, V. Pandurangan, M. W. Bigrigg, S. Oguz, K. Tew, C. Williams, G. R. Ganger, and P. K. Koshla. Selecting the Right Data Distribution Scheme for a Survivable Storage System. Technical report, CMU, 2001. CMU-CS-01-120.