Surviving congestion in geo-distributed storage systems

Brian Cho
University of Illinois at Urbana-Champaign

Marcos K. Aguilera
Microsoft Research Silicon Valley

Abstract. We present Vivace, a key-value storage system for web applications that span many geographically-distributed sites. Vivace provides strong consistency and replicates data across sites for access locality and disaster tolerance. Vivace is designed to cope well with network congestion across sites, which occurs because the bandwidth across sites is smaller than within sites. To deal with congestion, Vivace relies on two novel algorithms that prioritize a small amount of critical data to avoid delays due to congestion. We evaluate Vivace to show its feasibility and effectiveness.

1 Introduction

Web applications such as web mail, search, online stores, portals, social networks, and others are increasingly deployed in *geo-distributed* data centers, that is, data centers distributed over many geographic locations. These applications often rely on key-value storage systems [24] to keep persistent user data, such as shopping carts, profiles, user preferences, and others. These data should be replicated across data centers or *sites*, to provide disaster tolerance, access locality, and read scalability.

Traditional replication solutions fall in two categories:

- *Local replication,* designed for local-area networks, which provide strong consistency and easy-to-use semantics. Protocols for local replication include ABD and Paxos [16, 37]. These protocols perform poorly when replicas are spread over remote sites.
- *Remote replication,* designed for replicas at many sites, which provides good performance in that case, by propagating updates asynchronously (e.g., [24, 23, 40]). These protocols provide weaker forms of consistency, such as eventual consistency [51].

Vivace is a key-value storage system that combines the advantages of both categories, by providing strong consistency, while performing well when deployed across multiple remote sites. Strong consistency is desirable: although weak consistency may be adequate in some cases (e.g., [24, 23]), strong consistency is necessary in others, when reading stale data is undesirable.[1] The difficulty with providing strong consistency is that it requires coordination across sites to execute storage requests. At first glance, this coordination appears to be prohibitive, due to the higher network latencies across sites. However, typical round-trip latencies across sites are not too high—around 50–200 ms—which can be reasonable for certain storage systems. In fact, Megastore [17] has demonstrated that synchronous replication can sometimes work across sites. But the real problem occurs when the cross-site links become congested, causing the round-trip latency to increase to several seconds or more—as observed in systems like Amazon's EC2 (Section 2). In that case, Megastore and existing synchronous replication solutions suffer from delays of several seconds, which are unacceptable to users. A study by Nielsen [45] indicates that web applications should respond to users within one second or less.

A solution to this problem is to avoid network congestion, by provisioning the cross-site links for peak usage. This solution can be expensive and wasteful, especially when the system operates at peak usage only rarely, as in many computer systems. A better solution is to provision cross-site links more conservatively—say, for typical usage—and design the system to deal with congestion when it appears. This is the approach taken in Vivace.

The key innovation in Vivace is two novel strongly consistent replication algorithms that behave well under congestion. The first algorithm is simpler and implements a storage system with read and write atomic operations. The second algorithm is more complex and implements a state machine [50], which supports generic read-modify-write atomic operations. These algorithms rely on network prioritization of a few latency-critical messages in the system to avoid delays due to congestion. Because the size and number of prioritized messages is small, only a tiny fraction of the total network bandwidth comprises prioritized data. We evaluate Vivace and its algorithms to show that they are feasible and effective at avoiding delays due to congestion, without consuming resources significantly.

2 Setting and goals

We consider a system with multiple data centers or *sites*, where each site consists of many machines connected by a local-area network with low latency and high bandwidth. Sites are connected to each other via wide-area network links which have lower bandwidth and higher latencies than the links within a site. Machines or processes are subject to crash failures; we do not consider Byzantine failures. Network partitions may prevent communication across sites. Such partitions are rare, be-

[1] For this reason, cloud services such as Google AppEngine, Amazon SimpleDB, and others support both weak and strong consistency.

Mbps	K$/month	Mbps	K$/month	Mbps	K$/month
2	3.3	8	11.6	40	31.2
4	6.4	10	13.7	50	37.3
5	7.8	20	21.4	60	43.4
6	9.1	30	25.1	100	67.9

Figure 1: Sample cost to connect sites using MPLS VPNs, as a function of the contracted maximum bandwidth [5].

cause the cross-site links are either provided by ISPs that promise very high availability, or by private leased lines that also provide very high availability. Nevertheless, partitions may occur; we represent them as larger network delays that last until the partition is healed. The system is subject to disasters that may destroy an entire site or disconnect the site from the other sites. We represent such a disaster as a crash of many or all the machines in the affected site.

Processes have access to synchronized clocks, which are used as counters. These clocks can be realized with GPS sensors, radio signals, or protocols such as NTP. Alternatively, it is possible to replace the use of synchronized clocks in Vivace with a distributed protocol that provides a global counter. However, doing so would reduce performance of Vivace due to the need of a network round-trip to obtain a counter value.

The network is subject to intermittent congestion across sites, because the cross-site bandwidth is provisioned based on typical or average usage, not peak usage. This is so due to cost considerations. For example, small or medium data centers may use MPLS VPNs[2], which can provide a fixed contracted bandwidth priced accordingly (Figure 1). Large data centers might use private leased lines or other solutions, but the bandwidth could still be much smaller than needed at peak times, causing congestion. For example, in Amazon EC2, simple measurements of network latencies across locations show that congestion occurs often, causing the round-trip latencies of messages sent over TCP to grow from hundreds of milliseconds to several seconds or more [36].

We assume the ability to prioritize messages in the network, so that they are transmitted ahead of other messages. We evaluate whether this assumption holds in Section 6.2. Support for prioritization is required only at the network edge of a site—say, at the router that connects the site to the external network—not in the external network itself, because congestion tends to occur at the site edge.

We wish a distributed key-value storage system that replicates data across sites and provides strong consistency, defined precisely by linearizability [33]. Roughly speaking, linearizability requires each storage operation—such as a read or write—to appear to take effect sequentially at an instantaneous point in time.

The key-value storage system should support two types of data objects: (1) RW objects, which provide read-write storage, and (2) RMW objects, which provide state machines; RMW stands for read-modify-write, which are operations that can modify the object's state based on its previous state. Objects of either type are identified by a *key*. A RW object has two operations, *write*(*value*) and *read*(), which stores a new value and retrieves the current value of the object, respectively. The size of object keys tend to be small (bytes), whereas the values stored in an object can be much larger (KBs).

RMW objects have an operation *execute*(*command*), which applies the command. RMW objects are state machines [50]—which are implemented by protocols such as Paxos—and the command can be an arbitrary deterministic procedure that, based on the current state of the object, modifies the state of the object and returns a result. We consider a slightly weaker type of state machine, where if many concurrent commands are executed on the same RMW object, the system is allowed to abort the execution of the commands, returning a special value $\perp$. Aborted operations may or may not take effect; if an aborted operation does not take effect, the user can reissue the operation after a while. By using known techniques, such as exponential random back-off or a leader election service, it is possible to guarantee that the operation takes effect exactly once [11].

3 Design and algorithms

We now explain the design of Vivace, with a focus on the replication algorithms that it uses.

3.1 Architecture

Vivace has a standard architecture for a key-value storage system, which we now briefly describe. There is a set of storage servers, which store the state of objects, and a set of client machines, which run applications. Applications interact with Vivace via a client library.

Each object has a *type*, which is RW or RMW (Section 2), and a *replica set*, which indicates the storage servers and sites where the object is replicated. In addition, for each site, each object has a set of local storage servers, called the *local replica set* (or simply *local set*) of the object, which we explain in Section 3.2. Our algorithms can make progress despite the crash of any minority of replicas in the replica set, plus any minority of replicas in the local set. Replica sets and local sets can be provisioned accordingly. For example, in our evaluation we provisioned three replicas for every replica set and local set, so the system tolerates one failure in each set.

[2] MPLS VPN is a technology offered by ISPs to connect together sites [1].

The type, replica set, and local set comprise the *metadata* of an object. To reduce the overhead of storing metadata, objects are grouped into containers, and all objects in the container share the same metadata. The container of an object is fixed and the container id is a part of the object's key. A directory service stores the metadata for each container. Clients consult the directory service rarely, since they cache the metadata. The directory service is itself implemented using Vivace's replication algorithms, except the metadata for the directory objects is fixed: the type is a RW object, and the replica set is a fixed set of DNS names.

3.2 Algorithm for RW objects

Vivace's algorithm for RW objects is based on the *ABD* algorithm by Attiya, Bar-Noy, and Dolev [16]. It is a simple algorithm that provides linearizable read and write operations that always succeed when a majority of replicas are up. Moreover, the algorithm ensures safety and progress in a completely asynchronous system. In Section 3.3, we present a more complex algorithm that implements a state machine for RMW objects. That algorithm requires some partial synchrony to ensure progress—as with any other state machine algorithm.

We now briefly describe the ABD algorithm. To write value v to an object, the client obtains a new timestamp ts, asks for the replicas to store (v,ts), and waits for a majority of acknowledgments[3]. To read the latest value, the client asks the replicas to send their current pairs of (v,ts). The client waits for a majority of replies, and picks the reply (v',mts) with the largest timestamp mts. The client then executes a write-back phase, in which it asks replicas to store (v',mts) and waits for a majority of acknowledgments. This write-back phase is needed to provide linearizability: it ensures that a subsequent read operation sees v' or a more recent value.

If the replicas are in different sites, the ABD algorithm sends and receives messages across sites before the operation can complete. If remote network paths are congested, this remote communication can take a long time. We propose to avoid these congestion delays, by having the client use prioritized messages that are transmitted ahead of other messages, thereby bypassing the congestion. Prioritized messages must be small, otherwise these messages themselves will congest the network. To obtain small messages, we modify the ABD algorithm by breaking up its messages into two parts: critical fields—such as timestamps, statuses, and acks—that must be sent immediately, and the other fields. We restructure the ABD algorithm to operate correctly when the messages are broken up, and then we use prioritized messages for sending the critical fields. The challenge in doing so is threefold. First, we must still continue to provide linearizability (strong consistency) when the messages have been split; in fact, we define linearizability as the correctness criteria for the new algorithm. The difficulty here is that a message that is split in two parts may be interleaved with the split messages of other clients, creating concurrency problems. To address such problems, the new algorithm includes some additional phases of communication and coordination. Second, we must find a very small amount of critical information to prioritize, otherwise the prioritized data will congest the network; we later analyze and evaluate the new algorithm to show that the prioritized fields we chose indeed comprise a small proportion of the total data sent. Third, we must not impose significant extra overhead in the new algorithm, otherwise it will perform worse than the original algorithm when the network is not congested; in particular, the new algorithm has extra phases of communication; we later evaluate this overhead and show that it is very small and worth the benefit, because the extra communication occurs in the local area network.

We now describe the algorithm in more detail. To read a value, the client uses a small prioritized message to ask replicas to send their current timestamp. Replicas reply with another small prioritized message. Once the client has a majority of replies, it computes the highest timestamp mts that it received. It then asks replicas to send the data associated with timestamp mts. The reply is a large non-prioritized message with data but, in the common case, a replica in the local site has the data, so this replica responds quickly without remote communication. Thus, the client can read the value without being affected by the congestion on the remote path. The client then performs a fast write-back phase, by sending a small prioritized message with only the highest timestamp, not the data value.

To write a value v, the client obtains a new timestamp ts. We want to avoid sending v to remote sites in the critical path. The client stores (v,ts) at temporary replicas located in the same site as the client; the set of temporary replicas is called the *local replica set*, and each replica is called a *local replica*. The client also stores the timestamp ts at the (normal) replicas—which are typically at remote sites—using small prioritized messages; these messages carry only the timestamp ts, not the data value, and they bypass congestion on the remote paths. Once the client receives enough acknowledgments, the operation completes. Meanwhile, in the background, each local replica propagates the data value to the (normal) replicas. These larger messages do not delay the client, even if there is congestion, because they are not in the critical path. Furthermore, the larger messages are not prioritized.

[3] In the algorithm in [16], there are no synchronized clocks, so there is an extra round of communication to obtain a new timestamp. Here we obtain the timestamp from the synchronized clocks.

Algorithm 1 Vivace algorithm for RW objects

```
function read(key):
    acks ← sendwait($*\langle$R-TS, key$\rangle$, nodes[key], $\lceil (n+1)/2 \rceil$)
    mts ← $\max_{1 \le i \le \lceil (n+1)/2 \rceil}$ acks[$i$].msg.ts
    data ← sendwait($*\langle$R-DATA, key, mts$\rangle$, nodes[key], 1)
    sendwait($*\langle$W-TS, key, data[0].ts$\rangle$, nodes[key], $\lceil (n+1)/2 \rceil$)
    return  data[0].val

function write(key, val):
    ts ← clock()
    sendwait($\langle$W-LOCAL, key, ts, val$\rangle$, local_nodes[key], $\lceil (n+1)/2 \rceil$)
    sendwait($*\langle$W-TS, key, ts$\rangle$, nodes[key], $\lceil (n+1)/2 \rceil$)

function sendwait(msg, nodes, num_acks):
    send msg to nodes
    wait for num_acks replies
    return the replies in an array acks[0..num_acks $-$ 1]

upon receive $\langle$R-TS, key$\rangle$:
    return  $*\langle$ACK-R-TS, ts[key]$\rangle$

upon receive $\langle$R-DATA, key, ts$\rangle$:
    wait until ts[key] $\ge$ ts and val[key] $\ne \perp$
    return  $\langle$ACK-R-DATA, ts[key], val[key]$\rangle$

upon receive $\langle$W-TS, key, ts$\rangle$:
    if ts $>$ ts[key] then
        ts[key] ← ts
        if remote_buf[key][ts] exists then
            val[key] ← remote_buf[key][ts]
            delete remote_buf[key][$x$] for all $x \le$ ts
        else val[key] ← $\perp$
    return  $*\langle$ACK-W-TS$\rangle$

upon receive $\langle$W-LOCAL, key, ts, val$\rangle$:
    local_buf[key][ts] ← val
    async
        sendwait($\langle$W-REMOTE, key, ts, val$\rangle$, nodes[key], $\lceil (n+1)/2 \rceil$)
        delete local_buf[key][ts]
    return  $*\langle$ACK-W-LOCAL$\rangle$

upon receive $\langle$W-REMOTE, key, ts, val$\rangle$:
    if ts $=$ ts[key] then val[key] ← val
    else if ts $>$ ts[key] then remote_buf[key][ts] ← val
    return  $\langle$ACK-W-REMOTE$\rangle$
```

Detailed pseudocode. The pseudocode is given by Algorithm 1. We denote by *read*(*key*) and *write*(*key*, *v*) the operations to read and write an object with the given *key*; n is the number of replicas for the object, and f is a fault-tolerance parameter indicating the maximum number of replicas that may crash. The algorithm requires that $f<n/2$, that is, only a minority of replicas may crash. The replica set of an object with a given *key* is denoted *nodes*[*key*], while the local replica set at a given *site* is denoted *local_nodes*[*key*, *site*]. We omit *site* from *local_nodes*[*key*, *site*] when the site is local (where the client is). That is, *local_nodes*[*key*] refers to the local replica set at the client's site. A prioritized message m is denoted $*\langle m \rangle$, and a normal message m is denoted $\langle m \rangle$.

The communication in the algorithm occurs via a simple function *sendwait*, which sends a given message *msg* to a set of *nodes* and waits for *num_acks* replies. The replies are returned in an array.

To write a value, the client obtains a new timestamp and executes two phases. In the first phase, the client sends the value and timestamp to the local replicas, and waits for an acknowledgment from a majority. In the second phase, the client sends just the timestamp to the replicas, using prioritized messages. When the client receives acknowledgments from a majority, it completes the write operation. Meanwhile, the local replicas propagate the value to the (regular) replicas. The client need not wait for the propagation to complete, because a majority of the local replicas already store the data and a majority of the replicas store the timestamp: if another client in a different site were to execute a read operation, it would observe the timestamp from at least one replica and know what value it needs to wait for.

To read a value, the client executes three phases. In the first phase, the client retrieves the timestamp from a majority of the replicas, and picks the highest timestamp *mts*. In the second phase, the client asks the replicas to send the data associated with this timestamp, if they have it. A replica replies only if it has a timestamp at least as large as the requested timestamp. This phase completes when the client obtains its first reply. In the common case, this reply arrives quickly from a replica in the same site as the client. Once the second phase has completed, the client knows the value that it must return for the read operation. In the third phase, the client writes back the timestamp to the replicas using prioritized messages. When the client receives a majority of acknowledgments, it completes the read operation.

Note that the client returns from a read operation without having to write back the value v. This is possible because the client that originally sent timestamp *mts* to the replicas did so only after it had stored v at a majority of local replicas. When the read operation returns, a majority of replicas has seen the timestamp *mts* of v, but not necessarily v itself. However, we are guaranteed that a majority of replicas subsequently receive v from the local replicas.

3.3 Algorithm for RMW objects

We now present Vivace's algorithm for state machines, to implement RMW objects. We apply the same principles as in Section 3.2: the basic idea is to break-up the protocol messages into critical and non-critical fields, restructuring the algorithm so that, in the critical path, remote communication involves only the critical fields. The non-critical fields are stored at a majority of local replicas and later propagated to the (regular) replicas in the background.

Our starting point is an algorithm for RMW objects similar to the algorithms in [27, 21, 11], which we now briefly describe—we later explain how we apply the above principles to this algorithm. The base RMW al-

Algorithm 2 Algorithm for RMW objects in a LAN

```
function execute(key, command):
  ots ← clock()
  acks ← sendwait(⟨OR, key, ots⟩, nodes[key], $\lceil (n+1)/2 \rceil$)
  if acks = ⊥ then return ⊥
  mts ← $\max_{1 \le i \le \lceil (n+1)/2 \rceil}$ acks[i].msg.ts
  mval ← acks[i].msg.val where acks[i].msg.ts = mts
  ⟨val, r⟩ ← apply(mval, command)
  w_acks ← sendwait(⟨OW, key, ots, val⟩, nodes[key], $\lceil (n+1)/2 \rceil$)
  if w_acks = ⊥ then return ⊥
  else return r

function sendwait(msg, nodes, num_acks):
  send msg to nodes
  wait for num_acks replies
  if any reply has status = false then return ⊥
  else return the replies in an array acks[0..num_acks − 1]

upon receive ⟨OR, key, ots⟩:
  if ots > ots[key] then
    ots[key] ← ots
    return ⟨ACK-OR, true, ts[key], val[key]⟩
  else return ⟨ACK-OR, false⟩
upon receive ⟨OW, key, ots, val⟩:
  if ots ≥ ots[key] then
    ots[key] ← ots
    ts[key] ← ots
    val[key] ← val
    return ⟨ACK-OW, true⟩
  else return ⟨ACK-OW, false⟩
```

gorithm is not new; we explain it here for completeness.

To execute a command, a client first obtains a new timestamp *ts*, and sends it to the replicas. Each replica stores the timestamp as a tentative ordering timestamp and subsequently rejects smaller timestamps. The replica replies with its current value and associated timestamp. The client waits for a majority of responses, and picks the value v with the highest timestamp. It applies the command to the value v to obtain a new value v' and a response r. The client then asks the replicas to store v' with the new timestamp *ts*. Each replica accepts the request if it has never seen a higher timestamp; otherwise, the replica returns an error to the client. The client waits for a majority of responses, and if any of them is an error, the client aborts by returning ⊥; otherwise, if no responses were an error, the client returns r.

The pseudocode is given by Algorithm 2. The first message sent by a client has a tag *OR*, which asks each replica to store a tentative ordering timestamp and reply with its current value and timestamp. The second message sent by a client has a tag *OW*, which asks each replica to store the new value and timestamp. These messages can have data values that are large, and they are sent in the critical path.

We now explain how to send just small messages in the critical path, so that we can prioritize these messages and make the algorithm go faster when there is congestion. The *OR* message in Algorithm 2 itself does not have a data value, but its response carries data. We modify the algorithm so that the response no longer carries any data, just a timestamp. The client then picks the largest received timestamp *mts* and must now retrieve the value v associated with it, so that it can apply the command to v. To do so, the client sends a separate *OR-DATA*

Algorithm 3 Vivace algorithm for RMW objects

```
function execute(key, command):
  ots ← clock()
  acks ← sendwait(∗⟨OR-TS, key, ots⟩, nodes[key], $\lceil (n+1)/2 \rceil$)
  if acks = ⊥ then return ⊥
  mts ← $\max_{1 \le i \le \lceil (n+1)/2 \rceil}$ acks[i].msg.ts
  data ← sendwait(∗⟨OR-DATA, key, mts⟩, nodes, 1)
  if data = ⊥ then return ⊥
  ⟨val, r⟩ ← apply(data[0].val, command)
  sendwait(⟨W-LOCAL, key, ots, val⟩, local_nodes[key], $\lceil (n+1)/2 \rceil$)
  w_acks ← sendwait(∗⟨OW-TS, key, ots⟩, nodes[key], $\lceil (n+1)/2 \rceil$)
  if w_acks = ⊥ then
    return ⊥
  else
    return r

function sendwait(msg, nodes, num_acks):
  send msg to nodes
  wait for num_acks replies
  if any reply has status = false then return ⊥
  else return the replies in an array acks[0..num_acks − 1]

upon receive ⟨OR-TS, key, ots⟩:
  if ots > ots[key] then
    ots[key] ← ots
    return ∗⟨ACK-OR-TS, true, ts[key]⟩
  else
    return ∗⟨ACK-OR-TS, false⟩
upon receive ⟨OR-DATA, key, ts⟩:
  wait until ts[key] ≥ ts and val[key] ≠ ⊥
  if ts[key] = ts then
    return ⟨ACK-OR-DATA, true, val[key]⟩
  else
    return ⟨ACK-OR-DATA, false⟩
upon receive ⟨OW-TS, key, ots⟩:
  if ots ≥ ots[key] then
    ots[key] ← ots
    ts[key] ← ots
    if remote_buf[key][ots] exists then
      val[key] ← remote_buf[key][ots]
      delete remote_buf[key][x] for all x ≤ ots
    else
      val[key] ← ⊥
    return ∗⟨ACK-OW-TS, true⟩
  else
    return ∗⟨ACK-OW-TS, false⟩
upon receive ⟨W-LOCAL, key, ts, val⟩:
  local_buf[key][ts] ← val
  async
    sendwait(⟨W-REMOTE, key, ts, val⟩, nodes[key], $\lceil (n+1)/2 \rceil$)
    delete local_buf[key][ts]
  return ∗⟨ACK-W-LOCAL⟩

upon receive ⟨W-REMOTE, key, ts, val⟩:
  if ts = ts[key] then
    val[key] ← val
  else if ts > ts[key] then
    remote_buf[key][ts] ← val
  return ⟨ACK-W-REMOTE⟩
```

message to all replicas asking specifically to retrieve the value associated with *mts*. In the common case, a replica at the local site has the appropriate value and replies to the client quickly. The client can now proceed as before, by applying the command to obtain a new value v' and a response r that it will return to the caller when it is finished.

The *OW* message in Algorithm 2 carries the new value v', so we must change it. The client uses the local replica set as in the write function of Algorithm 1. The client proceeds in two phases: it first sends v' to the local replicas in a *W-LOCAL* message and waits for a majority of replies. In the background, the local replicas send the data to the (regular) replicas. The client then sends just the new timestamp *ts* to the replicas, knowing that they will eventually receive the value from the local replicas. Algorithm 3 presents the pseudocode with these ideas.

3.4 Optimizations

We now describe some optimizations to Algorithms 1 and 3, to further reduce the latency and bandwidth of operations.

3.4.1 Read optimizations

We present two optimizations for the three-phase read operation of Algorithm 1. The first optimization reduces the number of phases, while the second removes the need for a client to communicate with a majority of replicas.

Avoiding or parallelizing the write-back phase. Recall that Algorithm 1 has a write-back phase, which propagates the largest timestamp *mts* to a majority of replicas. This phase can be avoided in some common cases. In the original ABD algorithm, if (C1) the client receives the largest timestamp *mts* from a majority of replicas in the first phase, it can skip the write-back phase. Similarly, we can skip this phase in Algorithm 1, provided that the same condition (C1) is met and another condition is also met: (C2) the timestamp of the data received in phase two is also *mts*. (Condition (C2) is not needed in the original ABD algorithm because the data and timestamps are together.) With this optimization, in the common case when there are no failures or concurrent writes, a read completes in two phases. It is also possible to read in two phases when (C1) is not met, but (C2) is. To do this, we add a parallel write-back in the second phase, triggered when the client observes that (C1) does not hold. The client proactively writes back *mts* in phase two, in parallel with the request for data. When a data reply is received, the client checks (C2)—it compares the data timestamp with *mts*—and if it holds the read operation can complete in two phases. There are two rare corner cases, when (C2) does not hold. In that case, if (C1) was met in phase one, the read remains as in Algorithm 1; if neither (C1) nor (C2) hold, then the read can use the parallel write-back in phase two.

Reading data from fewer replicas. This optimization reduces bandwidth usage, by having the client send *R-DATA* requests to only some replicas. After the first read phase, the client considers the set of replicas for which it received *ACK-R-TS* containing the large timestamp *mts*. If a local replica exists in this set, the client sends a single *R-DATA* request to that replica. If not, the client sends the request to a subset of replicas, based on some policy. For example, the client can keep a history of recent latencies at remote links, and send a single request to the replica with the lowest latency. The policy used affects performance, but policy choice is orthogonal to the algorithm.

3.4.2 Role change optimizations

In these optimizations, the role played at a node is moved to another node: the first optimization moves the execution of commands from the client to a replica, and the second moves the client role from outside to inside a replica.

Executing commands at replicas. In the execute operation of Algorithm 3, the client reads the current value of the object from the replicas, applies the command to obtain the new value, and writes that value to the replicas. Doing so involves transferring the value from the replicas to the client and back to the replicas. If the value is large, but the command is small, it is more efficient for the client to send the command to the replicas and thereby avoid transferring the value back and forth. To do this, the client first sends the *OR-TS* message and finds the largest timestamp *mts*—this determines the state on which the command should be applied. Then, rather than retrieving the data, the client sends the command to the replicas and the timestamp *mts*. The replicas apply the command to the value with timestamp *mts* (they may have to wait until they obtain that value, but they eventually do), or they reject the command if they see a larger timestamp. If a replica applies the command, it stores the new value in a temporary buffer together with the new timestamp. The client waits for a majority of responses, and if none of these are rejects, the client can send *OW-TS* messages as before. A replica then retrieves the value from its temporary buffer. This optimization reduces the bandwidth consumption of remote links when the command is smaller than the data value.

Delegating to a replica. In Vivace, the client library does not directly execute Algorithms 1 and 3. Rather, the library contacts one of the replicas, which then executes the algorithms on behalf of the client. Doing so is a common technique that saves bandwidth of the client, at the expense of the added latency of a local round-trip. It also makes it possible to modify the algorithms without changing the client library.

Message type	normal max delay	priority max delay
Local, within site	δ	δ
Remote, across sites	D	d

Figure 2: One-way delay parameters for latency analysis.

4 Analysis

We now analyze the algorithms of Section 3.

4.1 Latency

We first consider latency, when a majority of the replicas are on sites different from the client's. (If a majority of the replicas are in the client's site, clients complete their operation locally.) One-way message delays are represented by a few parameters, depending on whether the delay is within or across sites, and whether the message is normal or prioritized, as shown in Figure 2. Within a site, normal and prioritized messages have the same delay δ, due to lack of congestion. The delay for messages sent to remote sites are represented as D when sent normally, and d when prioritized. All the delay parameters incorporate the time to send, transmit, receive, and process a message.

Figure 3 summarizes the results. An execution of an operation can have different latencies, depending on the set of live replicas and the object state at those replicas. We analyze two cases: common and worst. The common case represents a situation with no failures, so that there is a live replica at the local site (the site where the client is). The worst case occurs when (a) all replicas at the local site are failed, and (b) the latest value is not yet stored at any replicas in the replica set; it is stored only at temporary, local set replicas, in a site remote to the client.

We first consider writes of RW objects. The ABD algorithm has a round-trip between the client and replicas (latency: $2D$). In the new Algorithm 1, this round-trip is replaced by two phases: the first one has a round-trip with local replicas (2δ), and the second one has a prioritized round-trip to remote sites ($2d$). By adding these delays, we obtain the total write latencies in Figure 3.

The read operation of ABD has two phases, each with a round-trip ($2D + 2D$). Algorithm 1 has three phases. The first phase has a prioritized round-trip ($2d$). The second phase depends on whether there is a replica at the client's site. If there is, the phase has a local round-trip (2δ); otherwise, there are three message delays: (1) the client sends a prioritized request to the replicas (d); (2) the replicas may not have the most recent value and must wait for it from a remote temporary replica (D); and (3) a remote replica sends the value to the client (D). The final write-back phase has a prioritized round-trip to remote replicas ($2d$). By adding these delays, we obtain the total of read latencies in Figure 3.

Algorithm	Operation	Common Case	Worst Case
ABD Algorithm (prior work)	read write	$4D$ $2D$	$4D$ $2D$
Algorithm 1 (new)	read write	$2\delta+4d$ $2\delta+2d$	$5d+2D$ $2\delta+2d$
Algorithm 2 (prior work)	execute	$4D$	$4D$
Algorithm 3 (new)	execute	$4\delta+4d$	$2\delta+5d+2D$

Figure 3: Message delays: common and worst cases.

From Figure 3, we can see that reads and writes of RW objects are faster with the new Algorithm 1 than with ABD, because typically $\delta \ll d \ll D$. Also note that, in the common case, the latency of Algorithm 1 is independent of D, which is not true for ABD.

We now consider the execute operation for RMW objects. Algorithm 2 has two remote round-trips ($2D+2D$). Algorithm 3 has four phases. The first and fourth phases each have a prioritized round-trip ($2d + 2d$). Without failures, the second and third phases have a local round-trip ($2\delta+2\delta$). When replicas at the local site are not live, the read phase is prolonged, as in the read operation of Algorithm 1, which we analyzed above (it takes $d + 2D$ instead of 2δ). By adding these delays, we obtain the total of execute latencies in Figure 3. We can see that the new Algorithm 3 is significantly better than Algorithm 2 and that, in the common case, the latency of Algorithm 3 does not depend on D.

4.2 Size of prioritized and normal messages

Prioritized messages should be a small fraction of the traffic, otherwise they become useless. We now analyze whether that is the case with the new algorithms. Prioritized messages in Algorithms 1 and 3 have up to four pieces of information: message type, key, timestamp, and an accept bit indicating if the request is accepted or rejected. The message type and accept bit are stored in one byte. The timestamp is eight bytes, which is large enough so that it does not wrap around. The key has variable length, but is typically small, say 16 bytes. Adding up, the size of a prioritized message is up to 25 bytes. Each data message (which is not prioritized) has the preceding information and a value with several KBs, which is orders of magnitude larger than a prioritized message.

This difference in size is advantageous. Let k be the factor by which normal messages are larger than prioritized messages, and B be the bandwidth of a remote link. Then clients can issue storage operations with peaks of throughput equal to $P = k \times B$ without affecting the performance of the system: at the peak throughput, the entire remote link bandwidth is consumed by prioritized messages, and their message delays are still d, so Algorithms 1 and 3 perform as expected. Since k can be large (with data values of size 1 KB, $k \approx 40$), the benefit is significant.

4.3 Fault tolerance

The new Algorithms 1 and 3 are fault tolerant: they tolerate up to f replica crashes. Even if a site disaster destroys more than f replicas in a site, the algorithms safeguard most of the data: only data written in a small window of vulnerability is lost (data held by the temporary local replicas but not yet propagated remotely). Furthermore, the algorithms allow administrators to identify the lost data quickly: these are the keys for which the remote replicas store a timestamp but not the data itself.

5 Implementation

Vivace consists of 6000 lines of Java. Clients and servers communicate using TCP. The system can be configured to use any of the four algorithms of Section 3: the ABD algorithm, Algorithm 1, Algorithm 2, and Algorithm 3. We implemented the optimizations, in Section 3.4 of avoiding the write-back phase and delegation to a replica, for the experiments in Section 6.4. We did not implement the directory service: currently, the metadata for containers is kept in a static configuration file. This does not affect our performance evaluation, because metadata lookups are rare due to caching.

6 Evaluation

We now evaluate Vivace. After describing the experimental setup (Section 6.1), we validate the assumption we made in Vivace that prioritized messages are feasible and effective (Section 6.2). We then consider the performance of the new algorithms of Vivace (Section 6.3). Finally, we demonstrate the benefits of Vivace in a real web application (Section 6.4).

6.1 Experimental setup

The experimental setup consists of machines in Amazon's EC2 and a private local cluster in Urbana-Champaign. In EC2, we use extra large virtual machine instances with 4 CPU cores and 15 GB of memory, in two locations, Virginia and Ireland. We use the private cluster for experiments that require changing the configuration of the network. The local cluster has three PCs with 2.4 GHz Pentium 4 CPU, 1 GB of memory, and an Intel PRO/100 NIC. The cluster is connected to the Internet via a Cisco Catalyst 3550 router, which has 48 100 Mbps ports. The median round-trip latencies were the following (in ms):

	Local cluster	EC2 Virginia	EC2 Ireland
Local cluster	<1	23	109
EC2 Virginia		<1	93
EC2 Ireland			<1

6.2 Message prioritization schemes

Vivace assumes the existence of an effective mechanism to prioritize messages in the network. In this section, we examine whether this assumption holds. We consider network-based and host-based schemes to achieve prioritized messages, and evaluate their overhead and effectiveness in the presence of congestion. The network-based scheme relies on prioritization support by network devices, while the host-based scheme implements prioritization in software using a dedicated server. Both schemes can be set up within a site, without assumptions on the external network that connects sites.[4]

For each scheme, we answer four questions: What is required to use it? How to set it up? What is the overhead? How effective is it?

Network-based solution

What is required? The scheme requires devices and appropriate protocols that support prioritization of messages. Prioritized messages are available at several levels:

Layer	Device	Mechanism
IP	IP router	RFC 2474 (DiffServ)
MPLS VPN	Edge router	RFC 2474 (DiffServ)
Ethernet	Switch	IEEE 802.1p

One way to connect sites is via a private leased line, using a modem and an IP router or switch at each end of the line. An alternative cost-effective scheme is to use VPNs, such as MPLS VPNs.

With private leased lines, prioritization is possible via IP or Ethernet solutions. IP solutions were first available using the Type of Service (ToS) bits in the IP header, which were later superseded by the 6-bit DSCP field in IP DiffServ. The DSCP field defines a traffic class for each IP packet, which can be used to prioritize traffic at each network hop. Ethernet solutions are based on the IEEE 802.1p standard, which uses a 3-bit PCP field on an Ethernet frame. Such solutions are available even on commodity low-end switches, where a use case is to prioritize video or real-time gaming traffic in a home network connected to a broadband modem.

With MPLS VPNs, prioritization is available via IP prioritization at the edge routers [1].

How to set it up? The simpler and lower-end devices are configured via web-based user interfaces. Higher-end routers and switches have configuration interfaces based on command lines. We set up traffic prioritization in the Cisco Catalyst 3550 router by configuring it to classify packets based on DSCP bits at ingress and place them accordingly into egress priority queues [8, Chapter 28].

[4] Another scheme is TCP Nice [53], which de-prioritizes traffic. We can conceptually prioritize messages by de-prioritizing all others using TCP Nice, but doing so requires de-prioritizing traffic outside our system. This could be hard and it fails if other systems use UDP.

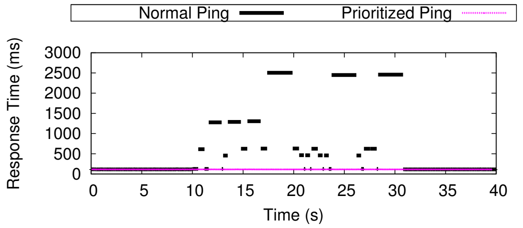


Figure 4: Round-trip delay for regular and prioritized messages using a router with IP DiffServ support.

What is the overhead? We evaluate the overhead of the scheme, by configuring the Cisco Catalyst 3550 router in the local cluster. We measured the round-trip latency of null requests sent from the cluster to EC2 (Ireland), with DSCP bit prioritization enabled and disabled. We found no detectable differences, indicating that the overhead of the prioritization mechanism in the IP router is small.

How effective is it? We perform a simple experiment: two clients in the private cluster periodically measure the round-trip time of their established TCP connection to a machine outside the cluster. One client was configured to set the DSCP field to a priority value, while the other used the default DSCP field. 10 seconds into the experiment, two machines in the cluster generate congestion by running iperf, which sends UDP traffic at a rate of 60 Mbps each to two machines. The congestion continues for 20 seconds and then stops. Figure 4 shows the round-trip latency observed by both clients. We can see that prioritized messages are effective: their latency is unaffected by the congestion. In contrast, congestion causes regular messages to be dropped, which triggers TCP retransmissions—sometimes multiple times—causing large delays due to the TCP back-off mechanism.

Host-based solution

What is required? This scheme requires one or more proxy machines to handle traffic between sites. Messages targeted to machines outside the local site are first sent to a local proxy machine. The proxy forwards the message to a proxy machine in the remote site, which finally forwards the message to the destination. In each proxy, packets are prioritized by placing them into queues according to their DSCP bits. The proxy is a dedicated Linux instance, running SOCKS Dante server processes [4] and using outbound priority queues configured with the HTB packet scheduler in the *tc* tool [7].

How to set it up? To reduce internal traffic at a site, the proxy is placed close to the site's external network connection. Machines are configured to use the proxy, using the SOCKS protocol.

What is the overhead? We measure the round-trip latency of null requests sent between the Virginia and Ireland EC2 locations, with and without the proxy, to determine the extra latency added by the proxy. There was no congestion in the network. Without the proxy, the average round-trip latency is 93 ms, while with the proxy, it is 99 ms. The overhead of the proxy is dwarfed by the much larger network latencies across sites.

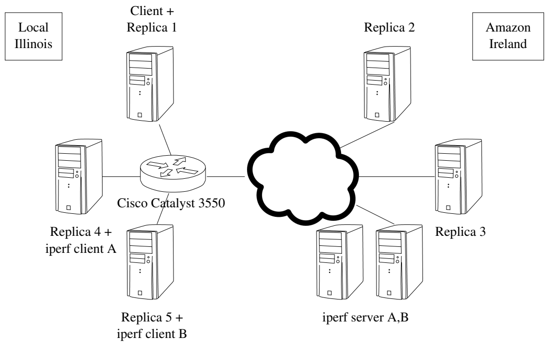


Figure 5: Network setup.

How effective is it? To evaluate the scheme, we place four machines at each of two EC2 locations (Virginia, Ireland). One machine runs the proxy, one runs two clients, and two run iperf to generate congestion. The experiment is the same as for the network-based solution, except that it uses an extra machine for the proxy, and it uses machines in EC2 only. The results are also similar and demonstrate that the prioritized messages are quite effective (not shown).

Applicability of each solution

The host and network-based solutions are applicable to a small or medium private data center connected by leased lines or MPLS VPNs. Larger data centers or the cloud have larger external bandwidths, and so the host-based scheme creates a bandwidth bottleneck at the proxy, making the network-based scheme a better alternative.

6.3 Storage algorithms

In the next experiments, we consider the performance of the new storage algorithms in Vivace. To do so, we configure Vivace to use either prior algorithms (ABD and Algorithm 2) or the new algorithms (Algorithm 1 and 3). The goal is to understand what are the overheads and benefits of the new algorithms.

The experimental setup consists of five server processes placed in two sites—the local cluster and EC2 Ireland—as shown in Figure 5. Replicas 1, 2, 3 are used as the replica set, with replica 1 placed locally and replicas 2 and 3 placed in Ireland. The local replicas consist of replicas 1, 4, 5, which are in the local cluster. The client is co-located with replica 1. We use the network-based prioritization scheme available in the IP router.

In each experiment, the client issues a series of operations of a given type on a 1 KB object for 20 seconds, and we measure the latency of those operations. For the RW algorithms (ABD and Algorithm 1), the operation types are *read* or *write*; for the RMW algorithms (Algorithms 2

Algorithm	Operation	Min latency	Max latency
ABD Algorithm (prior work)	read write	221 111	228 120
Algorithm 1 (new)	read write	221 113	231 121
Algorithm 2 (prior work)	execute	221	231
Algorithm 3 (new)	execute	222	231

Figure 6: Round-trip latency with no congestion (in ms).

and 3), the operation type is *execute*.

6.3.1 Overhead under no congestion

The new algorithms are designed by deconstructing some existing algorithms to prioritize critical fields in their messages. This deconstruction increases the number of communication phases of the algorithms, and raises the question of whether they would perform worse than prior algorithms. To evaluate this point, we run an experiment where there is no congestion in the network and we compare performance of the different algorithms.

The results are shown in Figure 6. We find that the algorithms perform similarly, which indicates that the overhead of the extra phases in the new algorithms are not significant. This result confirms the analysis in Section 4.1 when $\delta \ll d$.

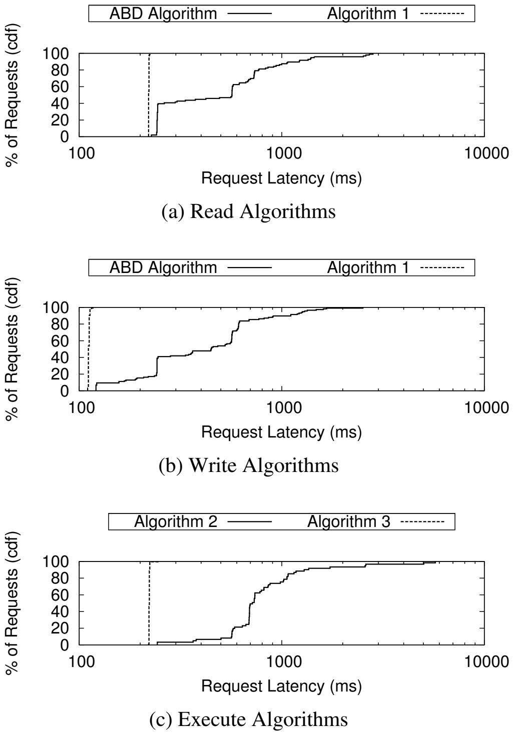
6.3.2 Benefit under congestion

In the next experiment, we evaluate the performance of the algorithms under network congestion to understand the benefits of prioritizing critical messages in the new algorithms.

The results are shown in Figure 7. Note that the x-axis has a logarithmic scale. As can be seen, the new algorithms perform significantly better than the equivalent prior algorithms. The continuous congestion causes large latencies in the execution of the prior algorithms. The difference in median latency is over 300ms for all operations. Perhaps more significantly, the differences in the higher percentiles are large. The difference at the 90th percentile for read and write operations is nearly 1s, while for the execute operation it is over 1s. This is particularly relevant because online services tend to have stringent latency requirements at high percentiles: for instance, in Amazon's platform, the latency requirements are measured at the 99.9th percentile [24]. With the use of priority messages, the new algorithms are better suited for satisfying such requirements.

6.4 Effect on a web application

We now consider how the new algorithms in Vivace can benefit a real web application. We use Vivace as the storage system for a Twitter-clone called Twissandra [2], which is originally designed to use the Cassandra [3] storage system. We replace Cassandra with Vivace, to obtain a system that employs the new algorithms with prioritization. More precisely, Twissandra uses Cassandra to store account settings, tweets, follower lists, and timelines. In the modified Twissandra, we store account settings and tweets in Vivace RW objects, and we store follower lists and timelines in Vivace RMW objects.



(a) Read Algorithms

(b) Write Algorithms

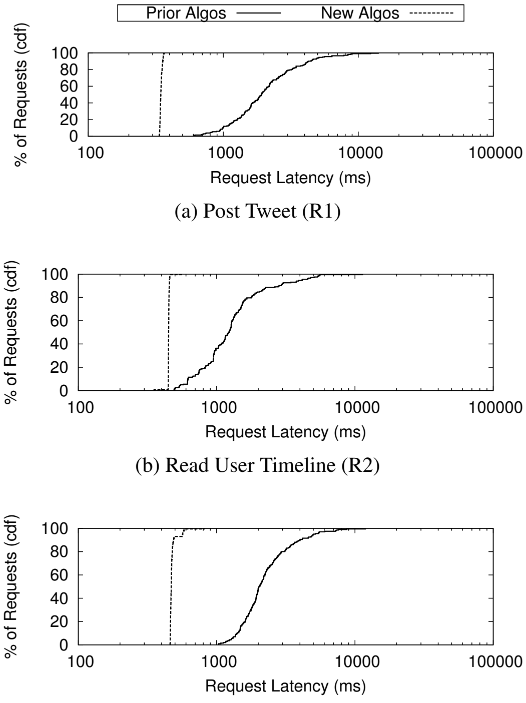
(c) Execute Algorithms

Figure 7: Latency of Vivace under congestion.

We evaluate the benefit of Vivace's algorithms, by measuring the latency of common user operations in Twissandra. As in Section 6.3.2, we configure Vivace to use the prior and new algorithms, and compare the difference in performance.

We run the experiments with two sites—the local cluster and EC2 Ireland—using the IP router to provide network-based prioritization. A load generator issues a sequence of 200 requests to Twissandra of a given type, one request at a time. We consider three request types: (R1) post a new tweet, (R2) read the timeline of a single user, and (R3) read the timeline of a user's friends. Each of these application requests result in multiple Vivace requests. We measure the latency it takes to process each request while the network is congested with background traffic generated by two machines running iperf (as in other experiments).

The results are shown in Figure 8. As can be seen, when Vivace is configured to use the new algorithms, the system is much more resilient to congestion. With the prior algorithms, latency for user operations often grew well above 1 second, with a median latency of 2.0s, 1.2s, 2.0s, and maximum latency of 14.1s, 11.3s, 11.9s, for



(a) Post Tweet (R1)

(b) Read User Timeline (R2)

(c) Read Friends Timeline (R3)

Figure 8: Latency of Twissandra-Vivace under congestion.

requests of types R1, R2, R3, respectively. In contrast, with the new algorithms, the median latency was 0.3s, 0.4s, 0.5s, and maximum latency was 0.4s, 0.6s, 0.8s, for the same request types, respectively—showing a significant improvement of using the new algorithms of Vivace under congestion.

7 Related Work

There has been a lot of work on distributed storage systems in the context of local-area networks (e.g., [34, 28, 48, 55, 26, 14, 20, 6, 31, 12, 39, 52, 41]). In contrast, we are interested in a geo-distributed data center setting, which comprises of multiple local-area networks connected by long-distance links. There is also work on distributed file systems in the wide area [54, 44, 49], which is a setting that in some ways resembles geo-distributed systems. These distributed file systems provide a weak form of consistency that requires conflict resolution [44] or other techniques to handle concurrent writes at different locations [54, 49]. In contrast, we provide strong consistency (linearizability), and our focus is on performing well despite congestion of remote links. Dynamo [24] is a key-value storage system where replicas can be located at multiple data centers, but it provides only relaxed consistency and applications need to deal with conflicts. Cassandra [3] is a storage system that combines the design of Dynamo [24] with the data model of BigTable [20]; like Dynamo, Cassandra provides relaxed consistency. PNUTS [23] is a storage system for geo-distributed data centers, but it too resorts to the technique of providing relaxed consistency to address performance problems. COPS [40] is a key-value storage system for geo-distributed data centers, which provides a consistency condition that is stronger than eventual consistency but weaker than strong consistency, because it allows stale reads.

There has been theoretical work on algorithms for read-write atomic objects or arbitrary objects (e.g., [29, 13, 30, 47, 43]). There has also been practical work on Byzantine fault tolerance (e.g., [19, 9, 35, 22, 32]) which considers the problem of implementing a service or state machine that can tolerate Byzantine failures. These algorithms were not designed with the geo-distributed data center setting in mind. In this setting they would see long latencies under congestion, because processes send large messages across long-distance links in the critical path.

In the context of wide-area networks, there have been proposals for more efficient protocols for implementing state machines. The Steward system [15] builds Byzantine fault tolerant state machines for a system with multiple local-area networks connected by wide-area links. They use a hierarchical approach to reduce the message complexity across the wide-area. Mencius [42] is another system that builds a state machine over the wide-area. The use of a multi-leader protocol and skipping allows the system to balance message load according to network conditions. Hybrid Paxos [25] is another way to reduce message complexity. It relies on the knowledge of non-conflicting commands as specified by the application [46, 10, 38]: for instance, commands known to be commutative need not be ordered across replicas, allowing for faster processing. Steward, Mencius, and Hybrid Paxos can reduce or amortize the bandwidth consumed by messages across remote links. Yet, the systems can experience high latency if congestion on these links reduces the available bandwidth to below the message load. Our work addresses this problem by deconstructing algorithms and prioritizing a small amount of critical information needed in the critical path. As long as there is enough bandwidth available for the small fraction of load produced by prioritized messages, bursts of congestion will not slow down our algorithms.

PRACTI [18] separates data from control information to provide partial replication with flexible consistency and data propagation. Vivace also separates certain critical fields in messages, but this is done differently from PRACTI for many reasons. First, Vivace has a different purpose, namely, to improve performance under congestion. Second, Vivace uses different algorithms, namely, algorithms based on majority quorum systems, while PRACTI is based on the log exchange protocol of Bayou [51]. Third, Vivace provides a different storage service to clients, namely, a state machine [50], while PRACTI provides a more limited read-write service.

Megastore is a storage system that replicates data synchronously across multiple sites. Unlike Vivace, Megastore supports some types of transactions, but it has no mechanisms to cope with cross-site congestion.

Other related work includes peer-to-peer storage systems, which provide weak consistency guarantees rather than linearizability.

8 Conclusion

In this paper, we presented Vivace, a distributed key-value storage system that replicates data synchronously across many sites, while being able to cope with congestion of the links connecting those sites. Vivace relies on two novel algorithms that can overcome congestion by prioritizing a small amount of critical information. We believe that the volume of data across data centers will increase in the future, as more web applications become globalized, which will worsen the problem of congestion across sites. But even if that does not happen, Vivace will still be useful, by allowing remote links to be provisioned less aggressively. More broadly, we believe that geo-distributed systems that make judicious use of prioritized messages will become more relevant, not just for storage systems as we considered here, but also in a wider context.

References

[1] `http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6557/prod_white_paper0900aecd803e55d7.pdf` as of Oct 2011.

[2] `https://github.com/twissandra/twissandra`, as of Oct 2011.

[3] `http://cassandra.apache.org`, as of Oct 2011.

[4] Dante – proxy communication solution. `http://www.inet.no/dante/`.

[5] Global MPLS VPN pricing guide. `http://shop2.sprint.com/assets/pdfs/en/solutions/worldwide/taiwan_global_mpls_vpn.pdf` as of Oct 2011.

[6] The Hadoop distributed file system: Architecture and design. `http://hadoop.apache.org/core/docs/current/hdfs_design.html`.

[7] HTB Linux queuing discipline manual—user guide. `http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm`.

[8] *Catalyst 3550 Multilayer Switch Software Configuration Guide, Cisco IOS Release 12.1(13)EA1*. Mar. 2003.

[9] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *SOSP*, Oct. 2005.

[10] M. K. Aguilera, C. Delporte-gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. In *DISC*, Oct. 2000.

[11] M. K. Aguilera, S. Frolund, V. Hadzilacos, S. L. Horn, and S. Toueg. Abortable and query-abortable objects and their efficient implementation. In *PODC*, 2007.

[12] M. K. Aguilera, W. Golab, and M. A. Shah. A practical scalable distributed B-tree. *VLDB*, 1(1), Aug. 2008.

[13] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. In *PODC*, Aug. 2009.

[14] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *TOCS*, 27(3), Nov. 2009.

[15] Y. Amir, C. Danilov, J. Kirsch, J. Lane, D. Dolev, C. Nita-Rotaru, J. Olsen, and D. Zage. Scaling Byzantine fault-tolerant replication to wide area networks. In *DSN*, 2006.

[16] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *JACM*, 42(1), Jan. 1995.

[17] J. Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *Conference on Innovative Data Systems Research*, Jan. 2011.

[18] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *NSDI*, 2006. Extended version available at `http://www.cs.utexas.edu/users/dahlin/papers/PRACTI-2005-10-extended.pdf`.

[19] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *OSDI*, Feb. 1999.

[20] F. Chang et al. Bigtable: A distributed storage system for structured data. In *OSDI*, Nov. 2006.

[21] G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. *Distributed Computing*, 18(1), July 2005.

[22] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI*, Apr. 2009.

[23] B. F. Cooper et al. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, Aug. 2008.

[24] G. DeCandia et al. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, 2007.

[25] D. Dobre, M. Majuntke, M. Serafini, and N. Suri. HP: Hybrid paxos for WANs. In *European Dependable Computing Conference*, Apr. 2010.

[26] J. R. Douceur and J. Howell. Distributed directory service in the Farsite file system. In *OSDI*, Nov. 2006.

[27] E. Gafni and L. Lamport. Disk paxos. *Distributed Computing*, 16(1), Feb. 2003.

[28] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, Oct. 2003.

[29] S. Gilbert, N. Lynch, and A. Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *DSN*, June 2003.

[30] S. Gilbert, N. A. Lynch, and A. A. Shvartsman. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4), Dec. 2010.

[31] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable distributed data structures for internet service construction. In *OSDI*, 2000.

[32] J. Hendricks. Efficient Byzantine fault tolerance for scalable storage and services. Technical Report CMU-CS-09-146, Carnegie Mellon University, School of Computer Science, July 2009.

[33] M. P. Herlihy and J. M. Wing. Acm toplas. *ACM Trans. Program. Lang. Syst.*, 12, July 1990.

[34] J. H. Howard et al. Scale and performance in a distributed file system. *TOCS*, 6(1), Feb. 1988.

[35] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. In *SOSP*, Oct. 2007.

[36] T. Kraska, G. Pang, M. J. Franklin, and S. Madden. MDCC: Multi-Data Center Consistency. 1203.6049, Mar. 2012.

[37] L. Lamport. The part-time parliament. *TOCS*, 16(2), May 1998.

[38] L. Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, Mar. 2005.

[39] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *ASPLOS*, Oct. 1996.

[40] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. Don't settle for eventual: Stronger consistency for wide-area storage with COPS. In *SOSP*, Oct. 2011.

[41] P. Mahajan et al. Depot: Cloud storage with minimal trust. In *OSDI*, 2010.

[42] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *OSDI*, 2008.

[43] J.-P. Martin and L. Alvisi. A framework for dynamic Byzantine storage. In *DSN*, June 2004.

[44] L. B. Mummert, M. R. Eblig, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *SOSP*, Dec. 1995.

[45] J. Nielsen. *Designing Web Usability: The Practice of Simplicity*. New Riders Publishing, 1999.

[46] F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2), Apr. 2002.

[47] R. Rodrigues and B. Liskov. Rosebud: A scalable Byzantine-fault-tolerant storage architecture. Technical Report TR/932, MIT LCS, Dec. 2003.

[48] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. FAB: building distributed enterprise disk arrays from commodity components. In *ASPLOS*, Oct. 2004.

[49] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mhalingam. Taming aggressive replication in the Pangaea wide-area file system. In *OSDI*, Dec. 2002.

[50] F. B. Schneider. Implementing fault-tolerant services using the state machine approach : A tutorial. *ACM Computing Surveys*, 22(4), Dec. 1990.

[51] D. B. Terry et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, Dec. 1995.

[52] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *SOSP*, Oct. 1997.

[53] A. Venkataramani, R. Kokku, and M. Dahlin. TCP nice: A mechanism for background transfers. In *OSDI*, Dec. 2002.

[54] R. Y. Wang and T. E. Anderson. xFS: A wide area mass storage file system. In *Workshop on Workstation Operating Systems*, Oct. 1993.

[55] S. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, Nov. 2006.