Sprinkler — Reliable Broadcast for Geographically Dispersed Datacenters

Haoyan Geng and Robbert van Renesse

Cornell University, Ithaca, New York, USA

Abstract. This paper describes and evaluates Sprinkler, a reliable high-throughput broadcast facility for geographically dispersed datacenters. For scaling cloud services, datacenters use caching throughout their infrastructure. Sprinkler can be used to broadcast update events that invalidate cache entries. The number of recipients can scale to many thousands in such scenarios. The Sprinkler infrastructure consists of two layers: one layer to disseminate events among datacenters, and a second layer to disseminate events among machines within a datacenter. A novel garbage collection interface is introduced to save storage space and network bandwidth. The first layer is evaluated using an implementation deployed on Emulab. For the second layer, involving thousands of nodes, we use a discrete event simulation. The effect of garbage collection is analyzed using simulation. The evaluation shows that Sprinkler can disseminate millions of events per second throughout a large cloud infrastructure, and garbage collection is effective in workloads like cache invalidation.

Keywords: Broadcast, performance, fault tolerance, garbage collection.

1 Introduction

Today's large scale web applications such as Facebook, Amazon, eBay, Google+, and so on, rely heavily on caching for providing low latency responses to client queries. Enterprise data is stored in reliable but slow back-end databases. In order to be able to keep up with load and provide low latency responses, client query results are computed and opportunistically cached in memory on many thousands of machines throughout the organization's various datacenters [21]. But when a database is updated, all affected cache entries have to be invalidated. Until this is completed, inconsistent data can be exposed to clients. Since the databases cannot keep track of where these cache entries are, it is necessary to multicast an invalidation notification to all machines that may have cached query results. The rate of such invalidations can reach hundreds of thousands per second. If any invalidation gets lost, inconsistencies exposed to clients may be long-term. Other important uses of reliable high-throughput broadcast throughout a geoplex of datacenters include disseminating events in multi-player games and stock updates in financial trading.

Much work has been done on publish-subscribe and broadcast mechanisms (see Section 6). Pub-sub services focus on support for high throughput in the face of many topics or even content-based filtering, but reliability is often a secondary issue and slow subscribers may not see all updates. Some recent systems [4,20]

D. Eyers and K. Schwan (Eds.): Middleware 2013, LNCS 8275, pp. 247–266, 2013.

do provide high reliability and many topics, but the number of subscribers per topic is assumed to be small (such as a collection of logging servers). Group communication systems focus on high reliability, but such systems may stall in the face of slow group members and, partly for that reason, assume that group membership is small.

This paper describes Sprinkler, a high-throughput broadcast facility that is scalable in the number of recipients while providing reliable delivery. Sprinkler achieves its objectives through a novel broadcast API that includes support for garbage collection and through a careful implementation that is cognizant of the physical networking infrastructure.

Garbage collection both reduces load and makes it easier for clients or datacenters to recover from an outage. For example, if there are two updates or invalidations to the same key, then the first update is obsolete and it is no longer necessary to try and deliver it to clients. Similarly, if a temporary key is deleted, all outstanding updates can be garbage collected. As we show in Section 5, in applications where there are many updates to a small set of popular keys, and where there is significant use of temporary keys, such garbage collection can significantly reduce the demands on the broadcast service.

Sprinkler is designed for a system consisting of a small and mostly static number of datacenters each containing a large and dynamic set of machines. Consequently, Sprinkler uses two protocols: reliable multi-hop broadcast between datacenters, followed by reliable broadcast within a datacenter. Each datacenter deploys a replicated *proxy* to participate in the first protocol. While the details are different, both protocols depend on each peer periodically notifying its neighbors about its state (*i.e.*, gossip [13]).

To evaluate Sprinkler and find suitable values for certain configuration parameters, we conducted throughput, latency, and fault tolerance experiments. We first evaluated an incomplete prototype implementation of Sprinkler. Using Emulab [1] we were able to emulate realistic deployment scenarios and see what broadcast throughput is possible through a small number of datacenters. As a datacenter may contain thousands or tens of thousands of clients, we evaluated a complete implementation of the protocol through simulation, calibrated using measurements from experiments on the prototype implementation. We also quantified the effectiveness of garbage collection by conducting a simulation study on savings in storage space and network bandwidth using a workload mimicking cache invalidation in Facebook [21]. As a result of these experiments, we believe that Sprinkler is capable of disseminating millions of events per second throughout a large cloud infrastructure even in the face of failures.

The scientific contributions of this paper can be summarized as follows:

- the design and implementation of Sprinkler, a reliable high-throughput broadcast facility that scales in the number of recipients;
- a novel garbage collection interface that allows publishers to specify which messages are obsolete and do not need to be delivered;
- an evaluation of the throughput, latency, fault tolerance, and garbage collection of Sprinkler.

This paper is organized as follows. We start by giving an overview of the Sprinkler interface, as well as of the environment in which Sprinkler is intended to be deployed, in Section 2. Section 3 provides details of the various protocols that make up Sprinkler. Section 4 briefly describes the current implementation of Sprinkler. Evaluation of Sprinkler is presented in Section 5. Section 6 describes background and related work in the area of publish-subscribe and broadcast facilities. Section 7 concludes and presents areas for future work.

2 System Overview

2.1 Sprinkler Interface

Sprinkler has the following simple interface:

- `client.getEvent()` → *event*
- `client.publish`(*event*)

Each event e belongs to a *stream*, $e.stream$. There are three types of events: *data events*, *garbage collection events*, and *tombstone events*. Data events are simple byte arrays. A garbage collection event is like a data event, but also contains a predicate $P(e)$ on events e—if $P(e)$ holds, then the application considers e obsolete. A tombstone event is a placeholder for a sequence of events all of which are garbage collected. An event is considered *published* once the corresponding `client.publish`(*event*) interface returns. We consider each event that is published unique. An event e is considered *delivered* to a particular client when `client.getEvent()` → e' returns and either $e' = e$ or e' is a tombstone event for e.

The interfaces satisfy the following: Sprinkler only delivers data and garbage collection events that are published, or tombstone events for events that were published and garbage collected. Published events for the same stream s are ordered by a relation $\prec_s$, and events for s are delivered to each client in that order. If the same client publishes e and e' for stream s in that order, then $e \prec_s e'$.

For each event e that is published for stream s, each client is either delivered e or tombstone event for e followed by a matching garbage collection event g. A garbage collection event g containing predicate $g.P$ matches e if $g.P(e) \wedge e \prec_s g$ holds—that is, a garbage collection event cannot match a future event. A garbage collection event g can match another garbage collection event g'. In that case we require (of the application programmer) that $\forall e : e \prec_s g' \Rightarrow (g'.P(e) \Rightarrow g.P(e))$. For example, if g' matches all events (prior to g') that are red, then g also matches all events (prior to g') that are red. The intention is to ensure that garbage collection is final and cannot be undone.

A tombstone event matches a sequence of events that have been garbage collected. For each event being garbage collected, at least one tombstone event matching it is generated. A tombstone event t can also be garbage collected by another tombstone event t' that contains all events in t. For example, two consecutive tombstone events as well as two overlapping tombstone events can be replaced by a single tombstone event. However, tombstone events cannot contain "holes" (missing events in a consecutive sequences of events).

These properties hold even in the case of client crashes, except that events are no longer delivered to clients that have crashed. Sprinkler is not designed to deal with Byzantine failures. Note that the Sprinkler interface requires that all events are delivered to each correct client, and events can only be garbage collected if matched by a garbage collection event. Trivial implementations that deliver no events or garbage collect all events are thereby prevented.

2.2 Implementation Overview

Sprinkler is intended for an environment consisting of a relatively small and static number of datacenters, which we call *regions*, each containing a large and dynamic number of clients. A stream belongs to a region—we support only a small number of streams per region. A typical stream is "key invalidation" and a corresponding event contains the (hash of the) key that is being invalidated. The key's master copy is stored in the stream's region, as only the key's master copy broadcasts invalidation messages.

The rate at which events get published may be high, so high throughput is required. Low latency is desirable as well, although the environment is asynchronous and thus we cannot guarantee bounds on delivery latencies.

Each region runs a service, called a *proxy*, each in charge of a small number of streams. The proxy may be replicated for fault-tolerance. Sprinkler clients connect to the local proxy. When a client publishes an event, it connects to the proxy that manages the stream for the event. (Typically a client only publishes events to streams that are local to the client's region.) The proxy assigns a per-stream sequence number to the event and disseminates the event among the other proxies through the *proxy-level protocol* (PLP). Each proxy that receives the event stores the event locally and disseminates the event among the local clients through the *region-level protocol* (RLP). The details of the two protocols are described in the next section, and more on the implementation follows in Section 4.

3 Details of the Protocols

3.1 Proxy-Level Protocol (PLP)

Figure 1 contains a state-transition specification for proxies. The state of a proxy p is contained in the following variables:

- $pxID_p$ contains a unique immutable identifier for p;
- $streams_p$ contains the set of streams that p is responsible for;
- $Hist_p$ contains the events received by p and that are not yet garbage collected. $Hist_p$ is empty initially;
- cnt_p^s: an event counter for each stream s, initially 0;
- $expects_p^s$: for each stream s, a tuple consisting of a proxy identifier, a counter, and a timestamp.

Events are uniquely identified by the tuple $(e.type, e.stream, e.seq, e.range)$. Here $e.type$ is one of DATA, GC, or TOMBSTONE; $e.stream$ is the stream of the event,

specification `Proxy-Level-Protocol`:
  state:
    $pxID_p$: **unique id of proxy** p
    $streams_p$: **set of stream ids managed by** p
    $Hist_p$: **set of events that proxy** p **stores**
    cnt_p^s: **counters for each stream** s
    $expects_p^s$: **(proxy, counter, time)**
  initially:
    $\forall p$:
      $Hist_p := \emptyset$
      $\forall p' : p' \neq p \Rightarrow$
        $pxID_p \neq pxID_{p'}$
        $streams_p \cap streams_{p'} = \emptyset$
      $\forall s$:
        $cnt_p^s = 0$
        $expects_p^s = (\bot, 0, 0)$
  transition `addLocalEvent`(p, e):
    precondition:
      $e.stream \in streams_p \wedge e.seq = \bot$
    action:
      $cnt_p^{e.stream} := cnt_p^{e.stream} + 1$;
      $e.seq := cnt_p^{e.stream}$;
      $Hist_p := filter(Hist_p \cup \{e\})$;
  transition `addRemoteEvent`(p, e):
    precondition:
      $e.stream \notin streams_p \wedge e.seq > cnt_p^{e.stream}$
    action:
      $cnt_p^{e.stream} := e.seq$;
      $Hist_p := filter(Hist_p \cup \{e\})$;
  transition `rcvAdvertisement`(p, p', cnt, T):
    precondition:
      TRUE
    action:
      $\forall s \notin streams_p$:
        if $cnt^s > expects_p^s.seq +$
            $C/(T - expects_p^s.time)$ then
          if $expects_p^s.source \neq p'.pxID$ then
            if $expects_p^s.source \neq \bot$ then
              Unsubscribe$(expects_p^s.source, T)$
            Subscribe$(p'.pxID, s, cnt^s)$
          $expects_p^s = (p'.pxID, cnt^s, T)$

Fig. 1. Specification of a proxy. $filter(H)$ is a function on histories that replaced all events from H that are matched by a garbage collection event in H with a tombstone event.

and $e.seq$ is the sequence number of the event. For tombstone events, $e.range$ is the number of garbage-collected events represented by the tombstone—and $e.seq$ is the sequence number of the last such event. For non-tombstone events, $e.range = 1$.

Transition `addLocalEvent`(p, e) is performed when proxy p receives an event from a client that is trying to publish the event. The proxy only accepts the event if it manages the stream of the event, and in that case assigns a sequence number to the event. Finally, e is added to the history and a filter is applied to replace garbage collected events by tombstone events and to aggregate consecutive and overlapping tombstone events into single tombstone events.

Events for a stream s are ordered by their sequence number, that is, $e \prec_s e'$ iff $e.stream = e'.stream = s \wedge e.seq < e'.seq$.

Proxies forward events to one another over FIFO channels. Performing transition `addRemoteEvent`(p, e) adds an event to p's history for a stream that is not managed by p but by some other proxy. The transition applies the same filter to replace events that are garbage collected by tombstone events, and also updates $cnt_p^{e.stream}$ to keep track of the maximum sequence number seen for $e.stream$.

It is an invariant that $Hist_p$ does not contain any events e for which $e.seq > cnt_p^{e.stream}$, as is clear from the specification. We note without proof that it is also invariant that $Hist_p$ contains all published events with $e.seq \leq cnt_p^{e.stream}$, or matching tombstone event and garbage collection events.

A simple way for events to propagate between proxies would be to have each proxy broadcast its events to the other proxies. However, such an approach may not work if certain datacenters can no longer communicate directly. To address this, the way a proxy receives events from another proxy is through a subscription mechanism. For each non-local stream, a proxy subscribes to events from at most one other proxy, which does not have to be the owner of that stream. Periodically, each proxy p' broadcasts *advertisements* containing $cnt_{p'}$ to the other proxies, notifying them of its progress on each stream.

Proxy p maintains for each stream s a variable $expects_p^s$, containing a tuple consisting of a proxy identifier, a sequence number, and a timestamp. If p is not subscribed for the stream as is initially the case, then the tuple is $(\perp, 0, 0)$. If p is subscribed to receiving events from p', then $expects_p^s$ contains the proxy identifier of p', and the sequence number and time that p received in the latest advertisement from p'.

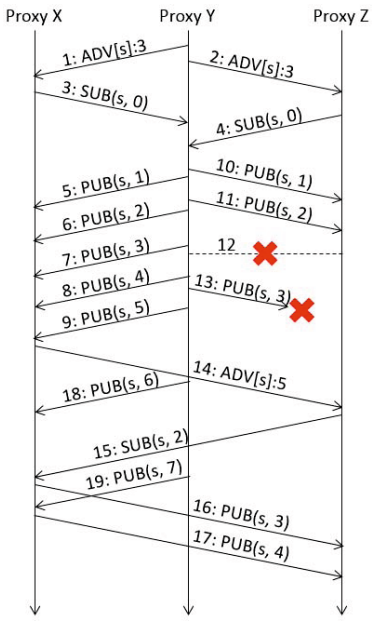


Fig. 2. Space-time diagram for subscription change when the link between two datacenters goes down

Transition `rcvAdvertisement`$(p, p', cnt_{p'}, T)$ shows what happens when proxy p receives an advertisement from p' at time T. For each non-local stream, p checks to see if the advertisement is further advanced than the last advertisement that it got for the same stream and by how much. C is a configuration variable. If set to 0, proxies tend to switch between subscriptions too aggressively. We divide C by the time expired since the last advertisement so that a proxy does not indefinitely wait for a proxy that may have crashed. When switching from one proxy to another p', p specifies to p' how far it got so that p' knows which event to send to p first.

Figure 2 illustrates how the subscription pattern changes adaptively in the presence of network outage. The figure shows three proxies X, Y, and Z, and messages flowing between them. For convenient reference, all messages are numbered. Initially, proxies can communicate with each other directly. Proxy Y is in charge of stream s, and events up to 3 have already been published. Proxies X and Z do not yet store any events for s, and are not subscribed to any source.

Messages 1 and 2 are advertisement messages for s, in which the count for s is 3. (Actual advertisement messages also include counters for other streams, but only s is shown for brevity.) Proxies X and Z send messages 3 and 4 to subscribe to stream s from proxy Y. In response, proxy Y starts sending events for s to proxies X and Z, shown as messages 5 through 11. The network between proxies Y and Z goes down at the broken line that is labeled 12, and subsequent events published by proxy Y cannot get through to Z (message 13). Message 14 is an advertisement message sent from proxy X to proxy Z for stream s, which contains a larger sequence number for stream s than the most recent advertisement message that proxy Z received from Y. So proxy Z changes its subscription to proxy X using message 15, and consequently starts receiving events from proxy X (messages 16 and 17). Meanwhile, proxy X continues to receive events for s directly from proxy Y, as illustrated by messages 18 and 19.

3.2 Region-Level Protocol (RLP)

The Region-Level Protocol delivers events from a proxy to all the clients within the region of the proxy. Reliability and throughput are key requirements: all events should be delivered to each correct client at high rate as long as it does not crash. Compared to the Proxy-Level Protocol, there are the following important differences: First, there are only a few number of proxies and the set of proxies is more or less static, while there are many clients in a region (on the order of thousands typically) and clients come and go as a function of reconfigurations for a variety of reasons. Second, proxies are dedicated, high-end, and homogeneous machines with resources chosen for the task they are designed for, while clients have other tasks and only limited resources for event dissemination. Third, proxies may be replicated for fault tolerance of event dissemination, but clients cannot be.

The Region-Level Protocol (RLP) consists of two sub-protocols: a gossip-based membership protocol based on [5], combined with a peer-to-peer event dissemination protocol loosely based on Chainsaw [23]. The membership protocol provides each client with a *view* that consists of a small random subset of the

other clients in the same region. The views are updated frequently through gossip. At any particular time, the clients and their views induce a directed graph over which clients notify their progress to their neighbors and request missing events, similar to the Proxy-Level Protocol. However, unlike proxies, clients do not keep track of old events for long because they have only limited capacity. But clients that cannot retrieve events from their neighbors can always fall back onto their local proxy, a luxury proxies do not possess.

We describe the two protocols in more detail below.

Membership Protocol. In the membership protocol, each client maintains a *local view*, which is a subset of other clients that has to grow logarithmically with the total number of clients. In the current implementation, the maximum view size V is configured and should be chosen large enough to prevent partitioning [5]: selecting a large view size increases overhead but makes partitions in the graph less likely and reduces the diameter of the graph and consequently event dissemination latency. Typically, V is on the order of 10 to 20 clients.

We call the members of the view the client's *neighbors*. A client periodically updates its local view by periodically gossiping with its neighbors. When a client c receives a view from its neighbor c', c computes the union of its own view and the view of c', and then randomly removes members from the new view until it has the required size. However, it makes sure that c' is in the new view. This last constraint, called *reinforcement* [5], is subtle but turns out to be important—without it the induced graph is likely to become star-like rather than to converge to a random graph. [5] shows that with reinforcement the protocol maintains a well-connected graph of clients with $O(\log N)$ diameter, where N is the total number of clients. Clients that have crashed or have been configured to no longer participate in the protocol automatically disappear from views of other clients because they do not reinforce themselves.

The Sprinkler membership protocol deviates from [5] in only minor ways. The local proxy is one of the clients that is gossiping. While partitioning in this graph is rare, it can happen. For this reason, each client occasionally gossips with its local proxy even if the proxy is not in its view. This causes partitions to fix themselves automatically. As shown in [5], partitions tend to be small in size: on the order of two to three clients. Therefore, if the view of a Sprinkler client is smaller than V, the client adds the local proxy to its view automatically. Such small partitions thus join the larger graph immediately. New clients start with a view consisting of only the local proxy.

Data Dissemination Protocol. Figure 3 presents a state-transition diagram for the data dissemination protocol. The state of a client c is contained in the following variables:

- $Recv_c$ contains the events delivered to c and that are not yet discarded. $Recv_c$ is empty initially. If c is a proxy, then $Recv_c = Hist_c$;
- cnt_c^s: is the sequence number for the last event that c received for stream s, initially -1;

specification `Region-Level-Protocol`:
  state:
    $Recv_c$: **set of events that node** c **stores**
    cnt_c^s: **counters for delivered events for stream** s
    $next_c^s$: **counters for requests for stream** s
  initially:
    $Recv_c := \emptyset$
    $\forall t$:
      $cnt_c^s = -1$
      $next_c^s = 0$
  transition `deliverEvent`(c, e):
    precondition:
      $e.seq - e.range \leq cnt_c^{e.stream} < e.seq$
    action:
      $Recv_c := filter(Recv_c \cup \{e\})$
      $cnt_c^{e.stream} := e.seq$
      `sendNotify`$(e.stream, cnt_c^{e.stream})$
  transition `receiveNotify`(c, c', s, cnt):
    precondition:
      $next_c^s \leq cnt$
    action:
      `sendRequest`$(c', s, next_c^s, cnt)$
      $next_c^s := cnt + 1$
  transition `receiveRequest`(c, c', s, nxt, cnt):
    precondition:
      TRUE
    action:
      $E := \{e \in Recv_c \mid e.stream = s \wedge$
        $\exists s \in (e.seq - e.range, e.seq] : nxt \leq s \leq cnt\}$
      `sendEvents`(c', cnt_c^s, E)
  transition `discardEvent`(c, e):
    precondition:
      $e \in Recv_c$
    action:
      $Recv_c := Recv_c - \{e\}$
  transition `requestFromProxy`(c, p, s):
    precondition:
      $cnt_c^s < next_c^s - 1$
      $s \in p.streams$
    action:
      `sendRequest`$(p, s, cnt_c^s + 1, next_c^s - 1)$

Fig. 3. Specification of a client for data dissemination

– $next_c^s$: is the sequence number of the next event that c wants to request for stream s, initially 0.

Performing transition `deliverEvent`(c, e) delivers an event to client c. If c is a proxy, this corresponds to c receiving the event in an `addLocalEvent`(c, e) or `addRemoteEvent`(c, e) transition. Otherwise c is an ordinary client that received the event either from the proxy or from a peer client in its region. Event e is delivered only if its sequence number is directly after the maximum sequence number delivered to c. When delivered, e is added to $Recv_c$ and $cnt_c^{e.stream}$ is updated. Finally, client c broadcasts a `NOTIFY` message its current neighbors (determined by the membership protocol), notifying them of its progress with respect to $e.stream$.

Transition `receiveNotify`(c, c', s, cnt) shows what happens when client c receives a `NOTIFY` message from client c' for stream s. If the sequence number in the `NOTIFY` message exceeds the events that client c has already requested, then c sends a `REQUEST` message to c' for the missing events.

Transition `receiveRequest`(c, c', s, nxt, cnt) is performed when client c receives a request from client c' for stream s. The client responds with an `EVENTS` message containing all events between nxt and cnt (possibly a sequence with holes, or even an empty sequence). The message also contains cnt_c^s so the recipient can detect what events exactly are missing from $Recv_c$.

Non-proxy clients may have limited space to store events. The Sprinkler specification gives clients the option of not keeping all events. In our implementation each client c has only limited capacity in $Recv_c$ and replace the oldest events with the newest events. Transition `discardEvent`(c, e) happens when client c removes event e from $Recv_c$.

Because clients do not keep all events, clients sometimes need to request missing events from the local proxy. In transition `requestFromProxy`(c, p, e), client c sends a `REQUEST` to the client's local proxy p. The client only sends requests for events that it previously requested from other clients.

Shuffling. In the protocol described above, a client c broadcasts a `NOTIFY` message to all its neighbors, each neighbor immediately sends `REQUEST` message to c, and c immediately responds with the requested events. Depending on the view size of c (bounded by V), this could create a large load on c.

In order to deal with this imbalance, each client only broadcasts the `NOTIFY` message to a subset of its neighbors of size F (for Fanout). This subset is of configurable size, and is changed periodically, something we call a *shuffle*. In the limit $F = 1$, but as we shall see in evaluation studies, a slightly larger subset has benefits for performance. We provide a simulation-based analysis on the effect of choosing different values for F and the *shuffle time*.

3.3 Fault Tolerance of a Proxy

So far we have described a proxy as if it were a single process, and as such it would be a single point of failure, depriving clients in its region from receiving events.

The Sprinkler proxy is replicated using Chain Replication [27]. To tolerate f failures in a region, there are $f + 1$ proxy replicas configured in a chain. Clients submit events by sending them to the head of the chain. The events are forwarded along the replicas in the chain, each replica storing the events in its copy of *Hist*. The tail of the chain communicates with the head replicas of its peer proxies, and also participates in the local RLP.

The chain is under the management of a local configuration service. In case a replica fails, it is removed from the chain. If the removed replica is not the tail, the impact is minimal—the predecessor of the replica may have to retransmit missing events to its new successor. It the head is removed, peer proxies and clients that try to publish events have to be notified. If it is the tail that is removed, a new tail ensues that has to set up new connections with the head nodes of its peer proxies. Both endpoints on each new connection exchange advertisements to allow the proxies to recover. A beneficial feature of Chain Replication is that the new tail is guaranteed to have all events that the old tail stored, and thus no events can get lost until all replicas in the chain fail and lose their state.

Sprinkler allows recovery of a crashed replica, as well as adding a replica with no initial state. The replica to the end of the chain, beyond the current tail, and will start receiving the events that it missed. Once the new tail is caught up, the old tail gives up its function and passes a token to the new replica. The replica then sets up new connections as described above.

3.4 Garbage Collection

In typical settings, Sprinkler broadcasts each event to thousands of hosts. All the events that are not garbage collected are stored at each of the proxies. In an environment with high load, the amount of data needs to be stored and transferred is huge. Efficient garbage collection would save critical storage space and network bandwidth. In this section, we give two examples of garbage collection policy.

One possible approach for garbage collection is to keep only the most recent events, and discard old events once they meet certain "age" criteria. An example is to keep only the most recent N events. In this case, each data event is also a garbage collection event: an event at index i collects all events with indices less that $i - N$. Another example is to discard all the events that are older than a certain period of time, say, k days. If this policy is enforced in a daily basis, the system generates one garbage collection event each day that collects all the events that are more than k days old. Such approach is useful if there is time bound on the usefulness of the data. LinkedIn uses such approach in processing log data with Kafka [20].

Another class of policy is key-based. In applications like cache invalidation, each data event states that the cache entry for a specific key is no longer valid. For any two events invalidates the same key, the later event implies the earlier one. From a client's perspective, if the later event is delivered, there is no need for the earlier one. So in this case, each data event is also a garbage collection event that collects all previous events on the same key.

The effectiveness of the key-based policy depends on actual workload. We show in section 5 that it is effective under our synthesized workload that shares similar properties to that of a popular, real web service.

4 Implementation

We have implemented a limited prototype of Sprinkler in the C programming language. We also have implemented a discrete event simulator of the full Sprinkler protocol described in this paper.

Nodes in the Sprinkler prototype communicate by exchanging messages across TCP connections. Each message starts with a header followed by an optional payload that contains the application data if needed. Batching of multiple events within a message is extensively used to optimize throughput.

We also have an initial implementation of the client library, except that we do not yet provide a comprehensive evaluation of it. Instead, in our evaluation, each client is configured to just receive events from proxies. Each proxy process also acts as client and is running both the Proxy-Level Protocol and the Region-Level Protocol. Proxy replication has only been implemented in the simulator.

5 Evaluation

In this section we evaluate the throughput and latency provided by Sprinkler for various scenarios, determine good values for parameters such as the fanout F, and investigate the efficacy of fault tolerance mechanisms within Sprinkler.

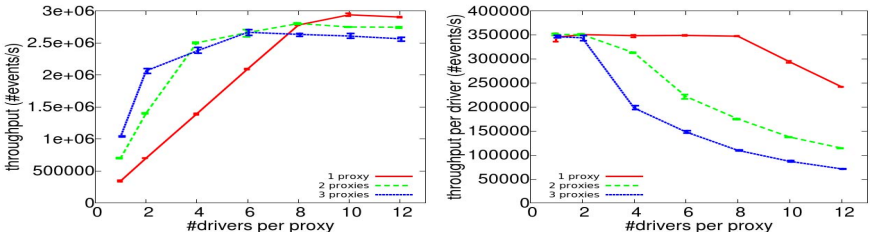
The performance of Sprinkler depends on both the Proxy-Level Protocol and the Region-Level Protocol. Given the small number of regions in a typical cloud infrastructure, we can use a prototype implementation of proxies to evaluate the Proxy-Level Protocol. However, since each region may have many thousands of clients, we evaluate the Region-Level Protocol using discrete event simulation. We use experimental measurements of the prototype implementation to calibrate the simulation of a large number of clients. For these measurements, each proxy is configured with a static view of clients.

5.1 Throughput of Proxy-Level Protocol

We tested the Proxy-Level protocol on an Emulab cluster[1]. Each node in the cluster is equipped with an AMD 1.6 GHz Opteron 242 processor and 16 GB of RAM. Nodes are connected to a single gigabit Ethernet switch.

We set up experiments with one, two, or three proxies. In these experiments, each proxy can communicate directly with each other proxy. The maximum view size V and the *fanout* F are both set to 3, and as described above, the local view of proxies do not change over time. Consequently there is no shuffling present in these experiments.

[1] We used the Marmot cluster of the PRObE project [3].



(a) Aggregate throughput (b) Throughput per driver

Fig. 4. Throughput as a function of the number of drivers per proxy

Some clients are used to publish events, and we call those clients *drivers*. Drivers do not receive any events—they just send events to proxies. Consequently, drivers do not run the Region-Level Protocol. Each driver invokes `publish`() in a closed loop with no wait time between invocations. The size of each event is fixed at 10 bytes, large enough to contain the hash of a key to be invalidated, say. Each published event is a garbage collection event: an event at index i specifies that all events with indices less than $i - 100,000,000$ can be garbage collected. We control the load on Sprinkler by varying the number of drivers attached to a proxy. In our experiments, each process, whether proxy, client, or driver, runs on a separate machine.

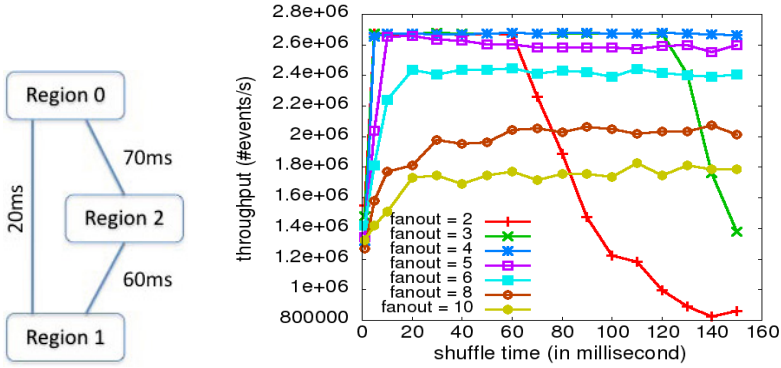
Figure 4(a) shows the throughput as a function of the number of drivers per proxy. Each data point shows an average over five experiments, as well as minima and maxima. The graph has three lines, one for each scenario. As the number of drivers increases, the throughput increases until the traffic load saturates the system. Peak throughput decreases slightly as the number of proxies increases because of the overhead of forwarding events between proxies. Figure 4(b) shows the throughput per driver for the same experiment.

5.2 Simulation Study

In the next experiment, we evaluate the performance of the complete Sprinkler protocol using discrete time simulation. The basic settings are as follows: There are three regions connected by 10 Gbps links (the bandwidth that is provided by the National Lambda Rail, a transcontinental fiber-optic network). Figure 5(a) shows the latencies between the three regions, chosen to reflect typical latencies for datacenters located on the west coast and the east coast of the United States.

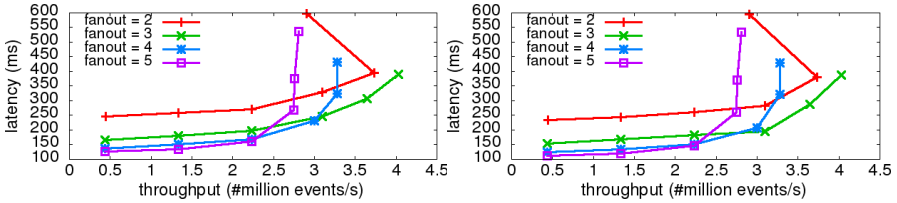
Within a region, processes communicate over 1 Gbps networks and one-way latencies are 1ms. Each region has 1000 clients and a proxy that has three replicas. Each client (as well as the tail server of the proxy) maintains a maximum view size of 20 peers.

In the 3-region prototype experiment of the previous section, the throughput peaks between 2.6-2.7 million events per second. We send 864k events per second to each proxy at a fixed rate, for a total of 2.592 million events per second, approximately matching the maximum throughput of the prototype.



(a) Topology (b) Throughput

Fig. 5. (a) The experimental topology used to simulate throughput and latency. (b) Average throughput as a function of shuffle time.

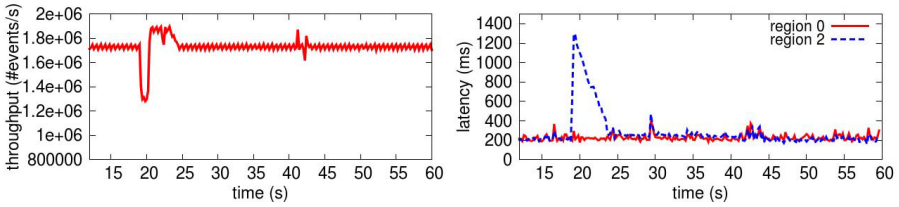


(a) Origin: region 0 (b) Origin: region 1

Fig. 6. Average latency and throughput observed as load is increased. The figures shows events added by clients in two different regions. The third region is similar.

Figure 5(b) shows the average throughput the simulator achieves varying the fanout F and the shuffle time. We do not show variance for clarity—it is small in our simulations. Each data point is the average throughput. To remove bias, measurements do not start until 300ms into an experiment, at which point the subscriptions are established and the throughput has stabilized. As can be seen, a consequence of this is that the throughput is slightly higher than the load added to the system, as the proxies catch up to deliver old events. Eventually, the throughput matches the load imposed on the system. The best throughput is achieved for a fanout of 4. For larger fanouts, the outbound bandwidth of a client gets exhausted for a relatively small number of events. Such clients cannot forward other events and start dropping events from their *Recv* buffer. This in turn results in an increase of requests made for missing events to the proxy, competing with bandwidth for normal traffic.

A similar effect happens when the fanout is small, but the shuffle time is long. Decreasing the shuffle time allows a client to forward events to more neighbors, effectively reducing the diameter of the forwarding graph, in turn reducing event loss in clients and the load on proxies.



(a) Average throughput (b) Average latency

Fig. 7. Performance over time with inter-region link failure

Figure 6 shows latency and throughput as load is increased, for various values for the fanout F. The latency of an event is the time from the event arriving at the local proxy until it is delivered to all clients. The shuffle time in these experiments is fixed at 30ms. In each line there are 6 data points, corresponding to increasing the load. At the first (leftmost) data point, 1/6 of the maximum throughput of the prototype implementation is introduced, that is, 1/6th of 2.592 million events per second. At the next data points we add 1/3rd of the load successively. Consequently, at the last data point we introduce $1/6 + 5 \times 1/3 = 11/6$ of the maximum load achieved on the prototype implementation. The halfway point on the line corresponds to the maximum load. Note that in some experiments the system becomes overloaded and cannot keep up with the load. We show results for events originating from different regions separately.

As shown in the figure, throughput gradually goes up until the system saturates. Before saturation, latency of events disseminated to 3000 clients is generally below 300ms.

5.3 Impact of Inter-Region Link Failure

Inter-region link failure blocks one region from communicating directly with another. The Proxy-Level Protocol supports indirect routing through other regions, and thus as long as there is transitive connectivity events should continue flowing to all clients. To evaluate this, we set up a four-region network, with inter-region latencies as shown in Figure 8. Latencies are chosen based on typical numbers for cross-country datacenter deployment. At the start of the experiment, all regions can directly communicate. After time $t = 19$, the link between regions 2 and 3 is taken out, and restored at time $t = 41$.

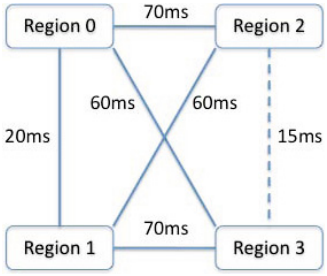


Fig. 8. The experimental topology used to simulate the impact of inter-region link failure, with the failed link (15ms) on the right side

Figure 7a shows throughput as a function of time. The network outage results in the brief drop in throughput at time $t = 19$, caused by the interruption of events flowing between regions 2 and 3. The throughput recovers shortly after time $t = 20$, after the new advertisement messages from regions 0 and 1 arrive and regions 2 and 3 update their subscriptions accordingly. The throughput

increases for about five seconds before returning to normal, as regions 2 and 3 catch up. Note the slight glitch after time $t = 41$ when the link is restored and regions 2 and 3 resume sending their events directly to one another.

For the same experiment, Figure 7b shows the latencies over time for events added from regions 0 and 2. Note that only the latter is directly connected to the failed link. Before the outage, the latencies of events added from the two regions are similar, both around 200ms. Latencies of events from region 2 significantly increase at the time the link is taken out, because those events cannot reach the proxy in region 3 until the subscription changes. The latencies of events from region 2 recover after the new subscription. Latencies are slightly higher than before because the path to region 3 has greater latency. After the link is restored, latencies drop to the original level after a short period of adjustment.

5.4 Effectiveness of Garbage Collection

In this section, we first describe the workload we use to evaluate the efficacy of garbage collection and show the workload is realistic. Next we show simulation results of the effectiveness of garbage collection.

Workload Description. In our model, there are two kinds of updates: 1) keys are updated with new values, and 2) new keys are added.

Each update event invalidates a random key from the current set of keys. The probability for a key to be selected follows a Zipf distribution. We assume that inter-arrival times of key update events are Poisson distributed.

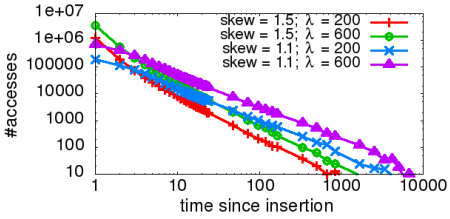
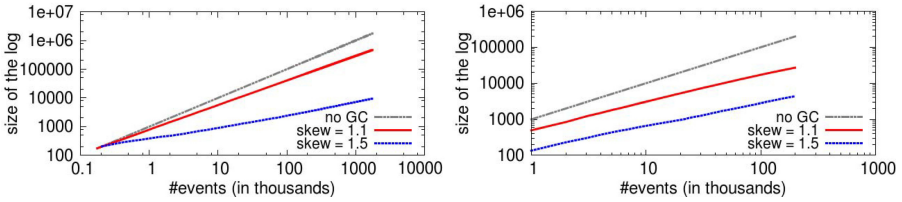


Fig. 9. Number of invalidations to keys with varying time since their addition. *skew* is the parameter for the Zeta distribution, while λ is the parameter for the Poisson distribution of inter-arrival times of invalidation events.

New keys are added to the set at random. In our model, the initial popularity of new keys also follows a heavy-tailed distribution. We choose a Zeta distribution (Zipf distribution over an infinite set) because for simulation purposes it is easy to scale up to a large number of keys. A single parameter, *skew*, determines the shape of a Zeta distribution. For simplicity, we assume that the Zeta distribution has the same skew as the Zipfian popularity distribution of keys. We also assume that inter-arrival times for new key events are Poisson distributed—we fix the parameter λ at 10 for these experiments.

Figure 9 shows the aggregated number of invalidation events on keys as a function of age. A data point at coordinates (x, y) shows that during the experiment, y invalidations are made to any object that have been inserted x ticks ago when such an invalidation was generated. The figure shows that new objects tend to attract more updates than old objects, because over time new objects come in, making old ones gradually less popular. popularity decrease also roughly follows



(a) The Zeta model: infinite key set (b) The Zipf model: finite key set

Fig. 10. Number of events stored at each proxy as a function of total number of events generated on log-log scale

a Zipf-like distribution. The slope of the line is steeper with larger *skew*, since the most popular objects get invalidated with higher probability. For the same skew, the plotted line is higher for larger λ as more invalidation events are generated.

In private communication with an engineer at a popular social network based web service provider, we confirmed that our workload, in particular the line with $skew = 1.1, \lambda = 200$, exhibits properties similar to their real workload.

Evaluation Results. We evaluated the performance of garbage collection with the workload described above. The metric is the amount of storage needed at each proxy. Note that since events of the same stream arrive in FIFO order at each proxy, there is no need to store tombstone events explicitly. A tombstone event is always followed by a corresponding data event.

Figure 10a shows the number of events to be stored at each proxy as new events are generated. Garbage collection saves roughly three-fourths of the space needed to store events. The effect becomes more significant if the workload is more heavily skewed.

In a real-world application, the number of keys are generally bounded. Figure 10b shows the same experiment on a slight variant of the above model: the set of possible keys is finite, and a Zipf distribution over the finite set is used to model the popularity. In this experiment, there are $100,000$ keys in the set initially. For each chosen skew value, we study two cases: a) the set of keys is fixed over time; and b) new keys are inserted into the system with a $1:10$ insertion/invalidation ratio. The result shows that garbage collection saves roughly 85% of the space with a skew of 1.1, and more with a higher skew. Only the results for the case of fixed set of keys are shown in the figure, since the addition of new keys makes little difference to the results.

6 Related Work

Sprinkler provides roughly similar functionality to topic-based publish-subscribe systems such as Information Bus [22] (TIBCO), iBus [6], JMS Queue [12], WebSphere MQ [2], and so on. The main focus of such systems is to support high throughput, but, unlike Sprinkler, slow subscribers or subscribers that join late may not receive all updates. Topic-based pub-sub systems are closely related to group communication systems [24], as topics can be viewed as groups [14].

Examples of group communication systems such as ISIS [8] focus on reliability, but throughput is limited by the slowest group member.

Apache HedWig [4] is a recent publish-subscribe system that is designed to distribute data across the Internet. Like Sprinkler, HedWig provides reliable delivery. However, HedWig is intended for a large number of topics with a small number of subscribers per topic (no more than about 10). In contract, Sprinkler can support only a small number of streams but can scale to hundreds of thousands of subscribers per stream. HedWig uses a separate coordination service (ZooKeeper [17]) to keep its metadata. To provide high throughput in the face of slow subscribers, HedWig logs all events to disk before delivery.

LinkedIn's Kafka [20] is another recent publish-subscribe system that provides high throughput and persistent on-disk storage of large amounts of data. Messages are guaranteed to be delivered and in order within a so-called partition (a sharding unit within a topic). Like HedWig, Kafka relies on ZooKeeper to maintain group membership and subscription relationships, but unlike Sprinkler does not deal with deployments in geographically dispersed locations.

Sprinkler is strongly inspired by gossip protocols. Proxies as well as clients gossip their state to their peers, and Sprinkler's per-region membership protocol is gossip-based as well. First introduced in [13], gossip has received considerable research. The first gossip protocols assumed all participants to gossip all their state with all other participants, providing strong reliability properties but limiting scalability drastically. Bimodal multicast [7] is an IP-multicast protocol that provides reliability with high probability through such a gossip mechanism. However, both IP multicast and uniform gossip limits its scalability. To obtain good scalability it is necessary to gossip in a more restricted manner.

Another gossip-based option is to provide each member with a small and dynamic view consisting of a random subset of peers, inducing a random graph that is connected with high probability [15,16,25,5]. SelectCast [9] (based on Astrolabe [26]) is a publish-subscribe protocol that builds a tree-structured overlay on participants using gossip. The overlay is then used to disseminate events. Sprinkler's membership protocol is entirely based upon [5]. Our Region-Level Protocol is influenced by the Chainsaw protocol [23]. While Chainsaw is intended for streaming video and can afford to lose video frames, RLP provides provides reliable delivery of (usually) small events.

Early large scale multicast protocols such as [18] build network overlays, but only provide best effort service. Multicast protocols such as SCRIBE [10], SplitStream [11] and Bullet [19] use Distributed Hash Tables to build tree-based overlays. Such protocols, besides providing only best effort service, tend to suffer from relatively high "stretch" as messages are forwarded pseudo-randomly through the overlay.

7 Conclusion and Future Work

We have described the design, implementation, and initial evaluation of Sprinkler, a high-throughput reliable broadcast facility that scales in the number of recipients. Prior approaches either assume a small number of recipients per

topic or drop events to slow recipients or temporarily disconnected recipients. In order to reach our objectives, we have added a garbage collection facility that replaces application-specified obsolete events with tombstone events. Such tombstone events can be readily aggregated. Garbage collection is particularly effective in the face of updates to keys that are skewed by popularity, or in the face of keys that are used temporarily for intermediate results. Combined with a careful design that separates inter-datacenter forwarding from intra-datacenter forwarding and specializes each case, we have shown that Sprinkler can provide high throughput in the face of millions of recipients.

At the time of this writing, we only have an initial implementation and evaluation of Sprinkler. Garbage collection events currently support predicates that remove events prior to a certain sequence number, or all previous events for the same key. We want to support a richer language for predicates, but have to ensure that Sprinkler proxy CPUs do not get overloaded by evaluation of predicates. We are working on a design of a predicate evaluation language as well as an index for events that allow fast identification of events that match a predicate.

Proxies have the option to maintain all events in memory, or to sync events onto disk to make them persistent. For this paper, we only implemented and evaluated the first option. While keeping everything in memory works well if garbage collection is sufficiently effective and replication prevents data loss, we want to evaluate the performance of storing events on disk. Most access will be sequential writing, and modern disks spin at an impressive 15,000rpm. As disks are cheap, we can deploy multiple disks in parallel to further increase bandwidth. Also SSDs are becoming increasingly cost effective. Cache controllers with battery-backed caches mask the latency of disks—they can complete writes even as the main CPU has crashed. We thus do not expect massive slowdown in the face of disk logging of events.

Acknowledgements. We are grateful for the anonymous reviews and the partial funding by grants from DARPA, AFOSR, NSF, ARPAe, iAd, Amazon.com and Microsoft Corporation.

References

1. Emulab, `http://www.emulab.net`
2. IBM WebSphere MQ, `http://www.ibm.com/WebSphere-MQ`
3. PRObE: Parallel Reconf. Observational Env., `http://newmexicoconsortium.org/probe`
4. Apache HedWig (2010), `https://cwiki.apache.org/ZOOKEEPER/hedwig.html`
5. Allavena, A., Demers, A., Hopcroft, J.E.: Correctness of a gossip based membership protocol. In: Proc. of the 24th ACM Symp. on Principles of Distributed Computing, pp. 292–301 (2005)
6. Altherr, M., Erzberger, M., Maffeis, S.: iBus - a software bus middleware for the Java platform. In: Proceedings of the Workshop on Reliable Middleware Systems, pp. 43–53 (1999)
7. Birman, K., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., Minsky, Y.: Bimodal Multicast. ACM Transactions on Computer Systems 17(2), 41–88 (1999)
8. Birman, K.P., Joseph, T.A.: Exploiting virtual synchrony in distributed systems. In: Proc. of the 11th ACM Symp. on Operating Systems Principles (1987)

9. Bozdog, A., van Renesse, R., Dumitriu, D.: SelectCast – a scalable and self-repairing multicast overlay routing facility. In: First ACM Workshop on Survivable and Self-Regenerative Systems, Fairfax, VA (October 2003)
10. Castro, M., Druschel, P., Kermarrec, A., Rowstron, A.: SCRIBE: A large-scale and decentralized application-level multicast infrastructure. IEEE Journal on Selected Areas in Communications (JSAC) 20(8) (2002)
11. Castro, M., Druschel, P., Kermarrec, A.-M., Nandi, A., Rowstron, A., Singh, A.: SplitStream: high-bandwidth multicast in cooperative environments. In: Proc. of the 19th ACM Symp. on Operating Systems Principles, pp. 298–313 (2003)
12. Curry, E.: Message-Oriented Middleware. In: Mahmoud, Q.H. (ed.) Middleware for Communications, Chichester, UK. John Wiley and Sons, Ltd. (2005)
13. Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. In: Proc. of the 6th ACM Symp. on Principles of Distributed Computing, pp. 1–12 (1987)
14. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.-M.: The many faces of publish/subscribe. ACM Comput. Surv. 35(2), 114–131 (2003)
15. Eugster, P.T., Guerraoui, R., Handurukande, S.B., Kouznetsov, P., Kermarrec, A.-M.: Lightweight probabilistic broadcast. ACM Trans. Comput. Syst. 21(4), 341–374 (2003)
16. Ganesh, A.J., Kermarrec, A.-M., Massoulié, L.: Peer-to-peer membership management for gossip-based protocols. IEEE Trans. Comput. 52(2), 139–149 (2003)
17. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: wait-free coordination for internet-scale systems. In: Proceedings of the 2010 USENIX Annual Technical Conference, pp. 145–158 (2010)
18. Jannotti, J., Gifford, D., Johnson, K., Kaashoek, M., O'Toole, J.W.: Overcast: Reliable multicasting with an overlay network. In: Proc. of the 4th Symp. on Operating Systems Design and Implementation (October 2000)
19. Kostić, D., Rodriguez, A., Albrecht, J., Vahdat, A.: Bullet: high bandwidth data dissemination using an overlay mesh. In: Proc. of the 19th ACM Symp. on Operating Systems Principles, pp. 282–297 (2003)
20. Kreps, J., Narkhede, N., Rao, J.: Kafka: a distributed messaging system for log processing. In: 6th International Workshop on Networking Meets Databases, NetDB 2011 (2011)
21. Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H.C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., Venkataramani, V.: Scaling Memcache at Facebook. In: Proc. of the 10th Symp. on Networked Systems Design and Implementation, Lombard, IL (April 2013)
22. Oki, B.M., Pfluegl, M., Siegel, A., Skeen, D.: The Information Bus—an architecture for extensible distributed systems. In: Proc. of the 14th ACM Symp. on Operating Systems Principles, Asheville, NC, pp. 58–68 (December 1993)
23. Pai, V., Kumar, K., Tamilmani, K., Sambamurthy, V., Mohr, A.E.: Chainsaw: Eliminating trees from overlay multicast. In: van Renesse, R. (ed.) IPTPS 2005. LNCS, vol. 3640, pp. 127–140. Springer, Heidelberg (2005)
24. Powell, D.: Group communication. Commun. ACM 39(4), 50–53 (1996)
25. Shen, K.: Structure management for scalable overlay service construction. In: Proc. of the 1st Symp. on Networked Systems Design and Impl., pp. 281–294 (2004)
26. Van Renesse, R., Birman, K.P., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Trans. Comput. Syst. 21(2), 164–206 (2003)
27. van Renesse, R., Schneider, F.B.: Chain Replication for supporting high throughput and availability. In: Proc. of the 6th Symp. on Operating Systems Design and Implementation (December 2004)