Taming uncertainty in distributed systems with help from the network

Joshua B. Leners[*‡] Trinabh Gupta[*‡] Marcos K. Aguilera[†] Michael Walfish[‡]

[*]The University of Texas at Austin [†]VMware Research Group [‡]NYU

Abstract

Network and process failures cause complexity in distributed applications. When a remote process does not respond, the application cannot tell if the process or network have failed, or if they are just slow. Without this information, applications can lose availability or correctness. To address this problem, we propose Albatross, a service that quickly reports to applications the current status of a remote process—whether it is working and reachable, or not. Albatross is targeted at data centers equipped with software defined networks (SDNs), allowing it to discover and *enforce* network partitions: Albatross borrows the old observation that it can be better to cause a problem than to live with uncertainty, and applies this idea to networks. When enforcing partitions, Albatross avoids disruption by disconnecting only individual processes (not entire hosts), and by allowing them to reconnect if the application chooses. We show that, under Albatross, distributed applications can bypass the complexity caused by network failures and that they become more available.

1 Introduction

In a distributed application, if a process stops responding, the rest of the application cannot be sure what is going on. Has the process crashed? Is there a network failure? Maybe there are no failures, and the issue is that the process is slow or the network is congested?

Unfortunately, if the application guesses incorrectly, it ends up with problems. These include split-brain scenarios (an incorrect guess that there was a problem can cause multiple instantiations of a process), consistency violations (if clients access the wrong server), and loss of availability (if the process incorrectly guesses that there are no problems).

This *uncertainty* about whether a failure exists is a perennial source of complexity in distributed systems (§2.2). Indeed, many applications devote substantial code to the problem (e.g., they implement fault-tolerant protocols that are impervious to uncertainty [12, 39, 48]). Other applications avoid the problem by leveraging *membership services* that track working processes. However, these services have various limitations. They take tens of seconds[1] or longer to react to failure [13, 36] (and faster reaction times would cause collateral damage [13, §2.8]), they sometimes halt working machines to eliminate uncertainty [26, 51],[2] or they cannot handle network problems [50].

In this paper, we propose a new membership service, called *Albatross*. We target data center networks and assume the presence of Software Defined Networks (SDNs). Albatross is a new design point. To our knowledge, it is the first membership service that achieves the following combination: (1) it addresses common network failures (as discussed below, Albatross cannot address *all* network failures: doing so would violate known impossibility results), (2) it is quick (it answers in less than a second), which improves availability, and (3) it avoids interfering with working processes and machines, which also improves availability. Albatross is a complete system and, as such, handles process failures too, but we will not focus on this aspect, since it is well-studied [9, 15, 26, 34, 50, 51, 67].

Albatross is based on two high-level insights. First, the ability to monitor and configure network elements makes it possible to provide (1)–(3), for reasons that will be explained in the coming pages. Second, SDNs provide a standard interface to the required network functionality.

1.1 Components and requirements

Albatross consists of a *host module* (installed on the hosts of applications that use the service) and a few replicated servers, called *managers*. The host module communicates with the managers, and exposes an interface that a process can query to learn the failure status of remote processes. Using SDN functionality, the managers receive notifications about the state of the network, determine which processes are reachable, and enforce their determinations by installing *drop rules* on switches. Albatross adopts several requirements:

(1) Provide guarantees that are well-defined and useful to applications. Ideally, Albatross would be an oracle that answers any query with perfect information, immediately. But this ideal is impossible: Albatross may be ignorant of a process's true status, and Albatross cannot provide information to processes that it cannot reach. Thus, Albatross relaxes the strength and scope of its guarantees. First, rather than promise perfect information, Albatross provides *definitive* reports, which guarantee the failure status of a remote process. To provide this guarantee, Albatross sometimes *inter-*

EuroSys '15, April 21–24, 2015, Bordeaux, France.
ACM 978-1-4503-3238-5/15/04.
http://dx.doi.org/10.1145/2741948.2741976

[1]Tens of seconds can be an expensive outage: some Web properties make tens or even hundreds of thousands of dollars per minute.

[2]This technique is called STONITH (for Shoot The Other Node In The Head).

feres with processes (as noted in the next requirement), which amounts to applying an old technique (STONITH [26, 50, 51]) to a new context (SDN-enabled networks). Second, Albatross provides *asymmetric* guarantees: it categorizes processes as *excluded* or *non-excluded* and promises definitive answers only to non-excluded processes. Third, Albatross allows reports to be delayed in favor of being definitive, but it strives to be quick (sub-second detection time). To our knowledge, this combination is new, and we find that it is strong enough to be useful to applications.

(2) Limit interference. Ideally, Albatross would only report events, not influence them. On the other hand, a convenient way to provide definitive responses is to interfere [26, 50, 51]. To balance these concerns, Albatross does several things. First, it inspects the state of the network rather than relying on coarse inferences from timeouts. Second, it manipulates that state at fine grain; its technique here is to embed a name space for applications in source MAC addresses, which enables switches to block the traffic of individual processes. The result is to avoid unwarranted interference with processes that use Albatross, and to eliminate interference with processes that do not use Albatross. Finally, Albatross allows disconnected processes to later reconnect.

(3) Use few resources. Switches have limited space for drop rules [35]. To work within this constraint, Albatross names processes according to their starting time and enclosing application; this naming scheme allows drop rules to be aggregated when failures affect many processes. Furthermore, Albatross must eventually garbage collect unused drop rules and other resources. To this end, Albatross introduces a reference counting scheme that handles distributed references and tolerates faults.

(4) Tolerate failures within Albatross itself. Albatross is itself a distributed system and, as such, is subject to the very failures that it wishes to detect. To be useful, Albatross must function under reasonable and common failures. (As an analogy, a fire alarm must function under usual types of fires.) Albatross responds in various ways. First, Albatross is replicated using Paxos [46]. Second, Paxos requires that a majority of the servers are responsive and mutually connected, which Albatross achieves by carefully placing servers (§5.3). Third, Albatross has a principled design, to avoid architecture flaws that lead to spurious failures.

Albatross cannot survive and report every conceivable network and process problem: doing so would mean tolerating arbitrary network partitions, which is impossible [29]. Albatross's limitations are, first, that it cannot survive failures that disconnect or kill a majority of its managers (as explained above). Second, to obtain service from Albatross, a host must be able to reach a majority of Albatross managers. The upshot is that Albatross does not work under catastrophic events (e.g., failure of power and backup); however, such failures take the data center offline anyway. Albatross does tolerate failures that affect only a restricted part of the network—which are the common case, according to a study of the failure events in the data centers of a large production service (§2.1).

1.2 Performance, results, and contributions

In evaluating Albatross (§7), we find that it has low cost: it requires little state in the network (fewer than 5 rules per switch to enforce disconnection), and uses little CPU and memory. Yet, it detects network failures an order of magnitude more quickly than the ZooKeeper membership service [36] (we will refer to this membership service as simply "ZooKeeper"). This gain owes to the design of Albatross: if ZooKeeper were to lower its timeouts to gain the same speed, its servers would be overwhelmed (§7.2).

Furthermore, we demonstrate that Albatross's guarantees are useful to applications. We show that integrating RAMCloud [59] with Albatross prevents clients from communicating with servers that have been declared failed; this eliminates a consistency bug in RAMCloud. We also show that Albatross can be used in place of existing membership services in distributed algorithms, despite its different guarantees.

The concrete contributions of this paper are as follows:

- Albatross, a service that provides a new combination: it reports network (and process) failures definitively, quickly, and with little interference (§3).
- A formalization of Albatross's guarantees, so that applications can reason about reports (§4).
- The design and implementation of Albatross; the design carefully composes novel techniques (a process naming scheme, a way of dropping processes' traffic at fine grain, a readmission protocol, algorithms for garbage collection) with existing ones (SDNs, Paxos [46], Falcon spies [50]) to produce a coherent system (§5–§6).
- An experimental evaluation (§7) of the implementation.

There is one more contribution of this paper, and it is conceptual. Whereas the purpose of SDNs was originally simplifying network management, this paper identifies a different use of SDNs: enhancing classical distributed systems. This connection had not been observed before, and we think that it may be more widely applicable.

2 Further motivation

2.1 What network partitions happen in data centers?

We analyzed a year-long trace of the failures that occurred in several data centers of a company with a strong Internet presence, using similar methods to Gill et al. [30]. Events in the failure trace were generated by in-device monitoring and were collected in a central repository using monitoring protocols (such as SNMP) or manual intervention. Events are tagged with metadata, including what type of device failed, and whether the failure was masked by redundancy; we used these tags to determine which events created partitions.

We found that larger data centers (more than a thousand net-

work elements) had about 12 partitions per month, of which about half disconnected an entire rack and half disconnected a single host. The partitions in the larger data centers never disconnected more than a single rack (owing to path redundancy), but we found that smaller data centers (fewer than 600 network elements) experienced multi-rack partitions. With this information in mind, we focus attention on partitions in a single data center that affect a subset of the network.

2.2 The whys, whats, and hows of membership services

One way to detect failures is by using end-to-end timeouts [9, 15, 34, 67]. However, timeouts are troublesome. First, a poor choice can compromise availability: if the timeout is too long, the system is forced to wait, and if the timeout is too short, the system wastes time responding to non-failures. Worse, the variability of response times means that there may be no good choice of timeout [74]. A second problem with timeouts is their inherent *uncertainty*: a timeout does not imply a failure.

To address this uncertainty directly, distributed systems typically use one or more of the following three techniques. The first technique uses a majority of processes to obtain agreement among correct processes [11, 16, 39, 46, 47, 58]. This technique is useful in building high-performance systems [12, 48], but it imposes particular structures on application developers. For example, applications must be built using the replicated state machine approach [45, 64] or using group communication primitives [11, 16].

A second technique is to use a mechanism like leases [13, 32, 36] or watchdogs [26] to ensure that suspected processes kill themselves in effect or fact. This technique assumes that the system has bounded timeliness. Also, short timeouts (as needed for fast detection) consume many resources (§7.2). Furthermore, with leases in particular, short timeouts can trigger complicated corner cases (§7.1).

The third technique is to kill processes that are suspected of having failed [50, 51]. A problem here is that killing tends to be coarse-grained (e.g., shutting down a machine). Furthermore, this technique can founder under network partitions, as the command to kill may not get through.

The aforementioned techniques bring complexity. Many services have been built that place these techniques behind a layer of abstraction that hides the complexity, allowing developers of distributed applications to interact with a clean abstraction. We refer to these services as *membership services*.[3] Membership services provide *definitive reports* about which processes are working, in the sense that processes can safely treat such reports as ground truth.

An alternative to end-to-end timeouts is to use information local to a system's components to detect failures [49, 50]. This approach allows a membership service, in the common case, to react to failures as they happen without waiting for an *end-to-end* timeout to expire. However, prior attempts at building membership services this way are incomplete: they do not handle network problems [50], or they give ambiguous reports [49]. (Section 9 elaborates.)

2.3 How does Albatross fit?

Albatross uses local information as hints but with a new mechanism for making reports definitive: before reporting a suspected process as not working, Albatross modifies the network to prevent the suspected process from sending messages to working processes. Albatross uses SDNs for this purpose. For its own fault-tolerance, Albatross uses majority-based techniques internally, thus following the tradition of implementing the complex technique once (§2.2).

As a concrete example, consider an application that uses primary-backup replication [4];[4] we use primary-backup as a running example for its conceptual simplicity, though membership services enable other fault-tolerance techniques (recovering from snapshots, raising an alarm, switching to a safe state, etc). In this application, the primary receives a request, replicates that request at the backup, and only then executes the request and responds to the requesting node. This setup provides fault-tolerance safely, via the invariant that replication happens before responding to the requestor.

However, for availability, the application needs a way to make progress if the primary or backup fails. Here is where the membership service enters. A standard choice would be ZooKeeper [36], which uses leases, and is used by production data center applications (e.g., [55]). The idea is that each replica acquires a lease at ZooKeeper named by its role ("primary" or "backup") and then monitors the other's lease. The backup learns that the primary is no longer working when the "primary" lease expires, and the backup can safely take over by acquiring the "primary" lease.

What if, as an alternative, the primary-backup application uses Albatross? Then, when Albatross suspects a partition (e.g., because a switch reports a link as down), it can install rules to stop the primary from using the network and then report the problem to the backup. The backup can then take over immediately—without having to wait for a lease to expire—because it knows that Albatross is preventing the primary from using the network.

3 Overview of Albatross

Albatross is a service that a process of a distributed application can query to learn about the status of a remote process. The status can be "disconnected" or "connected"; roughly, "disconnected" means crashed or partitioned, and "connected" means alive and reachable. If Albatross reports a process as "disconnected", it is safe to assume that process cannot affect the world. For the rest of this paper, a process refers to an

[3] This name is inspired by group membership services [11] but expanded in scope to include services such as ZooKeeper [36] and Chubby [13].

[4] More generally, membership services enable algorithms for consensus and atomic broadcast that can tolerate f failures using only $f+1$ processes [14] (versus $2f+1$).

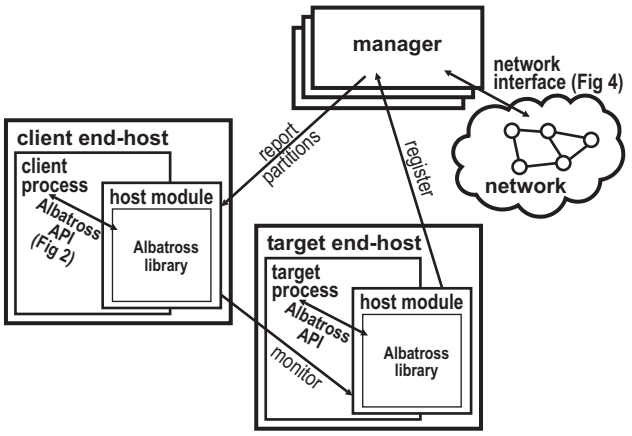


Figure 1—High-level view of Albatross. The host module provides the API through which applications use Albatross; the host module helps detect crashes of local processes. The manager is replicated at dedicated servers and coordinates Albatross's response to network and host failures. The manager interacts with the network through an abstract interface, and notifies clients about partitioned processes.

Function	Description
becomeAlbatrossProcess(appid)	register target process of app
handle = init((IP, proto, port), cb)	monitor a target, given by (IP, proto, port). callback cb is invoked when the target fails or is partitioned
query(handle)	return the state of target
startTimer(handle, timeout)	start timeout on target
stopTimer(handle)	cancel timeout on target
ackDisconnect()	acknowledge disconnection

Figure 2—The Albatross API.

operating system process.

Environmental assumptions. We target data centers with a single administrative domain. We presume the ability to configure network switches (e.g., via SDNs). We assume that minor modifications to end-host software are acceptable.

Components. Figure 1 depicts the components of Albatross. We survey them briefly below. (Section 5 gives details).

The *manager* detects, enforces, and reports network failures. Detecting and enforcing happens via a *network interface* that abstracts SDN-like features. Reporting happens by calling back client processes that have registered for notifications. The manager is a single logical entity that is replicated over several servers, using state machine replication. A *host module* detects and reports local process failures (to remote host modules); like the manager, this component uses callbacks for reporting. Much of the logic for detecting local process failures is borrowed [49, 50]. The host module also implements the Albatross API, described immediately below.

Albatross API. Figure 2 shows the Albatross API; Figure 3 gives an example use of the API and explains what causes communication among the components in Figure 1. A monitoring process is known as a *client*; a monitored process is called a *target*. To request monitoring, a client calls init(); this call generates a message to the target's host. Albatross returns notifications about the target via a client-supplied callback function or in response to query().

Action	Resulting communication
1. target calls becomesAlbatrossProcess()	target host module sends "register" RPC to the manager
2. client calls init(...), gets handle	client host module sends "monitor" RPC to the target's host module
3. client calls query(handle), gets "connected"	none
4. target crashes	target host module or manager sends RPC to the client's host module; host module invokes client callback (if any)
5. client calls query(handle), gets "disconnected"	none
6. target recovers, calls ackDisconnect()	target host module sends "deregister" RPC to the manager (not shown)
7. client calls init(...), gets new handle	client host module sends "monitor" RPC to the target's host module
8. client calls query(handle), gets "connected"	none

Figure 3—Example sequence of actions using the Albatross API and the resulting communication by Albatross.

The API serves three other purposes. First, processes register as targets with Albatross, by invoking becomeAlbatrossProcess(appid). This call may generate a message to the manager (the manager tracks applications). The specified appid should uniquely identify the application and should be used by all processes of the application.

Second, clients use the API to set an end-to-end timeout that serves as a *backstop* when Albatross cannot otherwise detect a problem. Specifically, Albatross expects a client process to call startTimer() when it is waiting for a message from a target process and to call stopTimer() when it receives the expected message. If the timer fires, Albatross disconnects the target and reports "disconnected".

Third, disconnected targets can reconnect, by calling ackDisconnect() (possibly after rolling back state), at which point monitoring clients must call init() again.

Informal contract. Albatross covers all host failures and common network failures. Its reports are definitive but asymmetric: it excludes some processes and promises definitive reports (about whether a process is excluded) only to non-excluded processes. Intuitively, the non-excluded processes are the ones that a majority of Albatross manager replicas can reach. These guarantees are formalized in Section 4.

In addition, Albatross provides fast (sub-second) detection time, which it achieves through its overall architecture: visibility into the network (which provides timely information), callbacks (which enable low latency without the overhead of

frequent polling), etc. Of course, one way to provide speed is to indiscriminately disconnect processes at any suspicion of a problem, but Albatross also limits interference, using the techniques summarized in Section 1.1.(2).

Rationale. Reporting *all* network failures is impossible [27]. Similarly, providing definitive, symmetric reports seems infeasible: how can a system give a report to a node that it cannot reach? Of course, just because a contract is feasible does not mean that it is useful to an application (Albatross could promise to return the string "elephant" always, which is feasible to implement but useless). Fortunately, Albatross's guarantees are useful to applications (§7.1), though there are some small corner cases, covered in the next section.

4 Albatross's contract

This section precisely describes Albatross's guarantees. We will define a set of *excluded* processes, and the guarantees will be asymmetric: processes outside the excluded set receive assurances that processes inside the set do not. The high-level concepts of exclusion and asymmetric guarantees have appeared before [10, 11, 16] but not, to our knowledge, in our specific context, namely failure reports [14].

Albatross's guarantees refer to a notion of *time*, which is a *logical* time at the Albatross manager; we are not assuming that entities in Albatross have synchronized clocks. We say that a *process p cannot reach process q at time t* if a message sent by p at time t would fail to be delivered to q (because, for example, q crashes before the packet arrives or there are no routes to q, or the routes to q disappear as the packet is traveling, etc.). Observe that this definition of "reachable" collapses a message's future and fate into a label associated with the sending time (t). We say that *processes p and q are partitioned at time t* (or p is partitioned from q at time t) if either p cannot reach q or q cannot reach p at time t.

The guarantees of Albatross are relative to a monotonically increasing set E of excluded processes. Intuitively, these are the processes that Albatross disconnects from the rest of the system (and the outside world). We denote by E_t the membership of E at time t. Albatross ensures the following:

- *(Exclusion Monotonicity) Processes are excluded permanently.* More precisely, if $t \leq t'$ then $E_t \subseteq E_{t'}$.
- *(Isolation) Non-excluded processes do not receive messages from excluded processes.* More precisely, if $p \in E_t$, $q \notin E_t$, and p sends a message to q at time t, then q never receives that message. In particular, if q receives a message from p, that message must have been sent before time t.

Exclusions are permanent, but in practice an application may wish to reconnect the process. This is allowed and modeled by having the process assume a new id.

The next property states that a process is indeed excluded if something bad happens to it:

- *(Exclusion Completeness) If a process has a problem for sufficiently long, then it is eventually excluded.* If q has crashed, or q is permanently partitioned from a process that is never excluded, then $q \in E_t$ for some t.

The above property does not guarantee immediate exclusion when the problem occurs, because the system may take some time to detect the problem; in practice, it is desirable that this delay be as small as possible. Also, exclusion is not guaranteed if q is partitioned temporarily, because the partition can heal before Albatross notices it. Similarly, exclusion is not guaranteed if q is partitioned from a process r that later gets excluded, because the exclusion of r may happen before Albatross notices the partition between q and r.

The final property states that queries by a non-excluded process return "disconnected" or "connected" according to whether the remote process is excluded.

- *(Correspondence) If a process is excluded then eventually a query about it by a non-excluded process always returns "disconnected". Moreover, a query about a process by a non-excluded process returns "disconnected" only if the process is excluded.* More precisely, if $q \in E_t$ then there is a time t_q such that, for all $t' > t_q$, a query about q by $p \notin E_{t'}$ returns "disconnected". If $p \notin E_t$ and a query about q by p returns "disconnected" at time t, then $q \in E_t$.

All properties above are conditional; Albatross provides them if the application follows the expectations in Section 3 (processes register, set backstop timeouts, etc.), and if a majority of Albatross managers remains alive and mutually connected (per the fault-tolerance discussions in Sections 1.1 and 5.3).

Consequences of the guarantees, and an example

- Albatross may return incorrect answers to queries done *by* excluded processes. This asymmetry is acceptable: to the *non-excluded* part of the system—which includes the outside world—these processes are as good as dead.
- Messages sent by an excluded process *before* Albatross reports a partition may still be received by a non-excluded process *after* Albatross's report. However, all of the non-excluded processes know that these messages causally precede Albatross's report because of the Isolation property, and can act accordingly (e.g., by dropping stale messages).
- Excluded processes may continue to interact with, and affect, *each other*. Thus, prior to reconnecting, excluded processes must rollback their state to some checkpoint that causally precedes [45] Albatross's "disconnected" report. By rolling back their state, excluded processes accept their effective deaths, and can be safely reintegrated using standard catch-up techniques (e.g., replay).
- It is possible for a crashed process to be temporarily reported as "connected"; the Completeness and Correspondence properties together imply that if a process has crashed or been partitioned, Albatross *eventually* reports it as "disconnected".

We now revisit the primary-backup example from Section 2.3, focusing on corner cases. First, consider the case that the backup receives a request from the primary *after* it hears

that the primary is "disconnected". The backup can safely discard this message because it knows that *the primary could not have responded to the requesting node (causally) before it was excluded* (and if it responds to the requesting node causally after exclusion, then the requesting node is also excluded, by Isolation). The italicized phrase holds because during the period when the primary was *not* excluded, it would have correctly observed the backup as "connected" (by Correspondence), and thus waited for an acknowledgment from the backup before responding to the requesting node.

Second, suppose a backup "takes over" for a non-excluded primary (a potential split-brain scenario). A correct backup will take over only if it hears that the primary is "disconnected"; since the primary is not in fact excluded, then the backup must be (by Correspondence). Thus, the "take-over" by the backup is something akin to a delusion (experienced by the backup and perhaps other excluded hosts).

What about reconnection? An excluded replica may eventually learn that it is excluded, for example, by querying its own state or receiving a "you are disconnected" message from the other replica. Then, the replica must determine a checkpoint from before it was excluded and rollback to it before reconnecting (via ackDisconnect()) and then replaying. For an excluded backup, a suitable checkpoint would be the one prior to the last request received from the primary.

5 Detailed design

This section describes Albatross's design, bottom up; we begin with the scheme by which processes are named and end with the core logic that enforces partitions and rehabilitates processes. Section 6 describes notable implementation details.

5.1 Names and identifiers

Under Albatross, each target process receives a *process id (pid)* when it registers (§3) with the host module. This pid uniquely identifies the process in terms of its host, application, and birth period. A pid contains the following fields:

- A *host id* (for example, an IP address);
- An *application id* (appid), which is programmer-supplied and unique to applications within the given network (§3);
- A *local id*, which differentiates multiple processes of the same application on the same host; and
- An *epoch number*, which identifies the epoch in which the process registered.

Epochs are determined by the manager; an epoch corresponds to a view of the network's topology and partitions.

Pids are carried in packets. Albatross uses the fields of a pid to create partitions (by filtering traffic). The choice of field depends on the desired granularity of a partition. For example, Albatross uses epochs when it needs to create a partition affecting an entire rack of end-hosts. We describe the interface for enforcing partitions next.

Primitive	Description
CUT-APP(switch, appid, port)	drop incoming traffic of application appid entering port of a switch
CUT-EPOCH(switch, epoch, port)	drop incoming traffic of epoch entering port of a switch
BLOCK(switch, pid)	drop all incoming traffic of process pid at a switch
SUBSCRIBE(destination)	request topology information and failure events to be sent to a destination

Figure 4—Network interface used by Albatross.

5.2 Network interface

As noted earlier, Albatross relies on SDN-like functionality from the network (though SDNs per se are not required to implement Albatross, as discussed in Section 8). Here, we describe the functionality in terms of an abstract interface, depicted in Figure 4. (Section 6.2 describes an implementation of this interface, using OpenFlow and NOX [33].)

CUT-APP and CUT-EPOCH tell a switch to block incoming traffic that (a) enters the given port and (b) matches the given appid or epoch (§5.1). BLOCK tells a switch to block traffic belonging to a given process id on all ports. Albatross also requires the ability to undo CUT-APP, CUT-EPOCH, and BLOCK (not shown in Figure 4). SUBSCRIBE tells switches where to send information about network topology and failure events. Events of interest are *link failure*, indicating that a link is deemed down; and *end-host failure*, indicating that a host connected to a port is deemed down. The intent is that these events are sent to the manager, described next.

5.3 Manager

Albatross's manager coordinates the response to most failures.

Network failures. At a high level, the manager tracks the network topology; when the topology experiences a partition, the manager chooses a main partition, asks switches at the edge of the main partition to block the traffic of Albatross applications coming from outside the partition, and then calls back clients to notify them about which processes have been disconnected. This procedure does not affect applications that do not use Albatross; it also does not affect applications that use Albatross but are launched after the failure is resolved. (One can think of this approach, loosely, as virtualizing partitions, in that different applications see different views of the network topology.)

In more detail, the manager runs the logic in Figure 5. The manager maintains a model of the current network topology. To that end, the manager, on starting up, requests notifications about topology changes, using SUBSCRIBE (our implementation assumes that the manager also begins with a correctly configured base topology; a less lazy implementation could build the topology as switches join). When the manager re-

```
at startup call SUBSCRIBE(self)

function handle_failure_event(link):
  remove link from topology
  if topology has a new partition:
    enforced := false
    pick candidate excluded set P
    while not enforced:
      enforced := enforce_partition(P)
    report_partition(P)

function enforce_partition(P):
  for each switch.port connecting to P:
    try:
      if switch.port connects to an end-host:
        for each appid in activeAppid running at end-host:
          call CUT-APP(switch, appid, switch.port)
      else: // switch.port connects to another switch
        for each epoch in activeEpochs:
          call CUT-EPOCH(switch, epoch, switch.port)
        currentEpoch := get_inactive_epoch()
        activeEpochs.insert(currentEpoch)
    except call failure:
      add switch to P
      return false
  return true

function report_partition(P):
  broadcast list of hosts in P and activeEpochs
```

Figure 5—Logic for detecting, enforcing, and reporting partitions.

ceives an *end-host failure* or *link failure* event, it updates its model. If the model has a partition, the manager chooses an *excluded set P* of switches and hosts. *P* is chosen to be all switches and hosts outside the largest strongly connected component that is reachable by a majority of manager replicas. Ties are broken arbitrarily.

Before Albatross reports a problem to clients, the manager enforces the partition: it invokes CUT-APP or CUT-EPOCH for every switch port bordering *P*. The choice of primitive carries a trade-off. On the one hand, if the port bordering *P* connects to an end-host, then the manager uses CUT-APP: each end-host has a small set of applications, so enforcing a partition requires few per-application rules. On the other hand, if the port bordering *P* connects to another switch, then using CUT-APP would require a rule for each application whose traffic is carried by the switch—potentially hundreds of rules. Instead, the manager handles this case by excluding at coarser granularity: it uses CUT-EPOCH, which tells that switch, and only that switch, to exclude all applications of active epochs. While CUT-EPOCH compromises on surgical disconnection, we note, first, that applications that begin in a new epoch are not affected, since the use of CUT-EPOCH induces a change of epoch. Second, CUT-EPOCH is invoked only when a switch fails or is partitioned; the common case, end-host failures, is handled with CUT-APP.

If the call to CUT-APP or CUT-EPOCH fails, the manager adds the switch to *P* and continues.[5] If the manager cannot use the network interface to install rules at *any* switch, then Albatross may be unable to report some failures, but this case is rare and means that the whole network is likely unusable (§1.1).

[5]This failure can be detected via timeouts. These timeouts differ from *end-to-end* timeouts (§2.2) because, first, they are monitoring a constrained component, and, second, SDN management traffic can be prioritized, which would make message latencies predictable and thus avoid spurious timeouts.

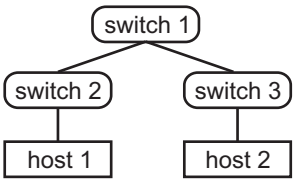
```
function handle_backstop_timeout(client, pid):
 for switch in topology such that switch is connected to host of pid:
  try: call BLOCK(switch, pid)
  except call failure:
   for link in switch:
    handle_failure_event(link)
 reply_to_client(client)
```

Figure 6—Logic to handle client backstop timeout on target *pid*.

When the procedure finishes, the manager broadcasts the list of hosts in *P* and the affected epochs. This information is received by the Albatross host modules, which mark the processes in *P* as down. The broadcast packets might be dropped; in that case, Albatross can still detect failures using the client's backstop timeout (see below)—albeit more slowly.

Host and process failures. Albatross treats a *host failure* as a one-host network failure, using the mechanism described immediately above. *Process* crashes, however, are handled differently; they separate into two cases. The first case is that a backstop timeout (§3) fires; this event causes the monitoring process to request help from the manager, which disconnects the target process, using BLOCK. Figure 6 shows the detailed logic. The second case is that a module running on the remote host is aware of a process crash; this case does not involve the manager at all and is covered in Section 6.3.

Example. Consider the example network below:



host 1 appids: 1, 2
host 2 appids: 2, 3
active epochs in network: 1, 3
appids of other hosts at switch 2: 4–47
appids of other hosts at switch 3: 13–63

If switch 3 reports an *end-host failure* event to the manager, then the manager will install at switch 3 a CUT-APP rule for appids 2 and 3. If switch 1 reports a *link-failure* event for its link to switch 3, or the manager suspects that switch 3 has failed (e.g., because switch 3 failed to install a CUT-APP rule), then the manager will install at switch 1 a CUT-EPOCH rule for epochs 1 and 3 and choose an inactive epoch as the current epoch. The manager use CUT-EPOCH instead of CUT-APP because CUT-EPOCH requires two invocations, whereas

CUT-APP would require fifty-three (the fifty-one ids at switch 3 plus appids 2 and 3); this choice is important because, as we will explain in Section 6.2, each invocation consumes scarce resources at the switch. This example ignores how failures might affect the manager; we describe the manager's fault tolerance next.

Fault tolerance. Recall that the manager is replicated for fault-tolerance (§3), following the state machine replication approach [45, 64], which is a majority-based technique for fault-tolerance (see Section 2.2). If manager replicas are placed at diverse parts of the network (such as different racks), then under common network partitions (described in Section 2.1), the majority of servers remains with the majority of network elements.[6]

Revisiting the guarantees. The formal guarantees (§4) reference a set E. This set is never materialized explicitly. Instead, recall that the manager maintains a set P, which is a set of (a) blocked processes, together with (b) a set of blocked applications, switches, and hosts (keyed by epoch). Because appids and epoch ids imply a set of processes, P implicitly represents the membership of E.

Albatross provides Completeness through the backstop timeout; if all else fails, a client will eventually request that the manager block the target (Figure 6). Albatross guarantees Isolation by configuring network switches to drop the traffic of excluded processes (Figure 5). Albatross guarantees Monotonicity because it never unblocks processes; it does recycle identifiers, however, as described in the next section. The first part of Correspondence is provided, also, by the backstop timeout; if a report from the manager is dropped, the client will eventually timeout and request blockage of the target (forcing the manager to retry its message). The second half of Correspondence is provided by the sequencing of events; the manager reports partitions only after enforcing them.

Albatross provides speed by reacting to failure events as opposed to end-to-end timeouts, in the common case. It limits interference by inspecting network state, by using surgical rules, and by allowing reconnection (described next).

5.4 Reconnecting processes and recycling identifiers

We now describe how Albatross reconnects processes and recycles epochs and pids. Although epochs change infrequently, they are important to recycle since, in our implementation (§6.1), there are only a handful of them, network-wide. Pids are scarce because the local id field—which identifies a process within a given application on a given host—is small.

Reconnecting processes. When a process tries to reconnect by calling ackDisconnect(), its host module gives it a new pid (§3,§4). Although Albatross's contract allows the host module to return any unallocated pid, in the interest of progress, the host module first checks that the new pid is not being blocked by any switch. To this end, before allocating a new pid, the host module asks the manager (a) what is the current epoch, and (b) which of the host's applications have been excluded (via CUT-APP). If no applications have been excluded, the host module returns a new pid with the current epoch and appid. If applications have been excluded, the host module locally blocks the processes belonging to those applications using a packet filter, asks the manager to undo the blanket exclusion (meaning, undo the CUT-APP calls at the edge switch), and only then returns the new pid. The order of these steps is important to upholding the Isolation property (§4); if the CUT-APP rules are undone before the excluded processes are blocked by their host module locally, then the excluded processes could affect non-excluded processes.

Garbage collecting epochs. Recall that when the manager enforces a partition of more than one end-host, it must activate a previously inactive epoch number. To allow the manager to track which epochs are active, host modules inform the manager which epochs they are using (by attaching a list to their normal messages to the manager). When the manager sees that an active epoch is not used by any host module, it undoes the CUT-EPOCH for the epoch, and marks it inactive.

Garbage collecting pids. A host module must be careful about when it reuses the pid of a process that has exited or has acknowledged a disconnection. Suppose, for example, that a host module were to give to a new process the pid of a process that had recently terminated; later, a third process could time out on the original terminated process, and have the manager enforce a partition using that pid, which would disrupt the new process. To avoid this and similar scenarios, Albatross includes the following counting scheme.

Each pid has a counter that is physically stored at the host module that allocated that pid (the local host); the counter tracks references to that pid held by other hosts. The local host module increments (or decrements) the counter when it hears that a remote process has started (or stopped) monitoring the associated process. A pid can be reused when these conditions all hold: (1) the pid of the process has reference count zero, (2) the local process has crashed or acknowledged the disconnection, and (3) the manager is not blocking the process's pid (with BLOCK).

The challenge in keeping the counter accurate is that there can be failures, both of clients referencing the pid and the host module storing the counter. To handle both cases, the local host module tracks, in a persistent write-ahead log, which clients have references; periodically, the local host module queries remote host modules to confirm that clients referencing its allocated pids are still running.

6 Selected implementation details

6.1 Packet marking

Figure 7 depicts the format of a pid. It consists of a 4-byte host identifier (currently, the host's IP address), together with 16 per-process bits. The per-process bits are the process's

[6] Even if the manager-majority partition holds a minority of processes, the minority can keep operating, under "$f+1$" algorithms (see footnote 4).

host id (IP addr) (32 bits)	epoch (3 bits)	appid (10 bits)	local id (3 bits)

Figure 7—Format of a Albatross process id (pid). Pids are six bytes; a process's pid appears in the source MAC address field of packets originated by the process. The number of epoch bits is small, but epochs are recycled (§5.4). The local id disambiguates multiple processes of the same application on the same host.

epoch number, the appid, and the local id.

Under Albatross, a process's pid appears in the source MAC address field of the packets that it originates. If a packet is sent by a process that is not using Albatross (including packets of ICMP, ARP, etc.), the bottom 16 bits are set to 0. Only the source MAC fields are used this way; the destination MAC field uses the usual MACs, obtained from ARP. This scheme assumes a scalable layer-two network in the data center (e.g., SEATTLE [41]).

The scheme has three features. First, it is easy to identify the traffic of applications that use Albatross—by observing a non-zero value in the bottom 16 bits. Second, blocking the traffic of a process at a switch requires a single rule (to match the source MAC); likewise, bit fields within the source MAC can be used to block the traffic of an entire application or epoch with one rule. Third, once the rule is installed, it need not be updated based on how and where the process sends data. By contrast, a scheme that blocked based on source TCP or UDP ports would require one rule per port used by the process, and updates in response to port changes. We discuss how this scheme affects existing Layer 2 protocols in Section 8.

6.2 Network interface implementation

Our implementation of Albatross assumes a network with OpenFlow switches and a NOX controller [33]. Given this environment, one can implement the network interface (Figure 4, §5.2) as follows. The CUT-APP(switch, appid, port) and CUT-EPOCH(switch, epoch, port) primitives direct the NOX controller to install an OpenFlow drop rule that matches on the appid or epoch bits of the pid; similarly, the BLOCK(switch, pid) primitive results in the installation of an OpenFlow drop rule that matches the entire pid.

SUBSCRIBE(destination) is implemented by augmenting the NOX controller to forward topology changes and failure events to the destination (which is the Albatross manager). Additionally, the destination needs to receive the link and end-host failure events (§5.2). *Link failure* events correspond to port- or link-down status events, and OpenFlow switches (by nature) notify the controller of such events. The controller simply forwards these notifications to the destination.

The more difficult case is *end-host failure* events. These are not directly supported by OpenFlow, so our implementation must synthesize them. Our solution leverages *SDN rule timeouts*, as follows. Each host module sends a special heartbeat packet to its switch every $T_{heartbeat}$ time units. On the first heartbeat, the switch sends an *unknown packet* event to the SDN controller. The SDN controller then configures the switch to (a) drop these heartbeat packets, and (b) send a timeout notification if the rule is not used for $T_{net\text{-}check}$ time units. If the controller receives such a notification, it sends an end-host failure event to the destination (the manager). Our implementation sets $T_{heartbeat}$ to 10 ms and $T_{net\text{-}check}$ to 1 s (the smallest OpenFlow timeout), which provide reasonably fast detection while tolerating dropped or delayed heartbeats.

6.3 Detecting process crashes

As noted in Section 5.3, process crash detection involves an additional module. This module mostly reuses prior work; we cover it for completeness. The core logic is a modified Falcon spy [50]. A Falcon spy uses local information (e.g., an OS's process table) to detect process crashes and report them to clients monitoring the crashed (target) processes. In particular, the Falcon spy inspects a process's internal state by invoking an application-specific function over IPC; this mechanism detects problems that may be hidden externally, such as deadlock. The modification from Falcon is that, instead of terminating unresponsive processes, the spy drops their traffic locally with a packet filter (using iptables). This modification reduces the impact of a false suspicion.

6.4 Miscellaneous implementation details

- Each host module caches the status of monitored target processes: when a client's host module receives a notification from a target's host module or from the manager, the client's module invokes the relevant callback function (Figure 2) and stores the "disconnected" for future queries.
- Albatross's manager is separate from the SDN controller. The manager's solution to replication makes use of a library [54]. (§8 discusses SDN controller fault-tolerance.)
- A final detail is interprocess communication (IPC). Albatross must enforce Isolation even when processes are on the same host. Thus, Albatross requires that all IPC be sent through the host's top-of-rack switch. If this requirement is burdensome (e.g., if processes use IPC extensively), two local processes can share the same Albatross pid, the tradeoff being that Albatross treats such processes as a unit.

7 Evaluation of Albatross

Using empirics and experiments, our evaluation reprises the arguments in Sections 2.2 and 2.3. First, we explain the benefits of a membership service that provides definitive reports, showing in the process that Albatross's contract is sufficient to derive these benefits (§7.1). Second, we investigate how well Albatross does the job of providing this contract (§7.2).

All experiments run on a prototype network (with 13 switches connected in a complete ternary tree) implemented using QEMU/KVM [62] virtual machines (version 1.0.1) and CPqD OpenFlow 1.2 software switches [60]. The hypervisor is a 64-core Dell PowerEdge R815 with AMD Opteron Processors and 128 GB of memory, running Linux (kernel ver-

sion 3.7.10-gentoo-r1). The network controller is NOX [33], modified to work with OpenFlow 1.2 [56].

7.1 What are the benefits of Albatross's contract?

On the one hand, the fact that membership services simplify the design of the distributed applications that use them has long been established: the fail-stop model (which assumes that all processes can detect all crashes correctly) is known to enable "easier" algorithms than the crash model. As just one example, Chain Replication [69] (a form of primary-backup) is simpler than Viewstamped Replication [57], Paxos-based replication [46], and Raft [58].

On the other hand, Albatross's contract (§4), with its asymmetric guarantees, is not precisely the fail-stop model. Thus, this section investigates whether Albatross's contract is sufficient to provide the same qualitative benefits. We do this by illustrating what can go wrong without a membership service; demonstrating that Albatross's guarantees are sufficient to simplify distributed algorithms; and describing the subtle relationships among Albatross, ZooKeeper (as an alternate membership service), and majority-based agreement.

Without a membership service, what can go wrong? We use RAMCloud [59] as a short case study. RAMCloud is a storage system that keeps data in memory at a set of *master servers*. These servers also process client requests to read and write data. For durability, a master server writes copies of data on the disks of multiple *backup servers*. A *coordinator* manages the configuration of the servers (which servers are masters for what data, etc.). To avoid losing writes or reading stale data, RAMCloud must guarantee that exactly one master server is responsible for a piece of data. One way to do this would be to use a membership service, but RAMCloud instead[7] uses several mechanisms internally: short timeouts, self-killing, propagation of crash information, and coordination among backups. These mechanisms must be orchestrated carefully to handle corner cases.

We first determine if RAMCloud ever returns stale (incorrect) data. We inject network failures at times carefully chosen to trigger the following corner case: a master is transiently disconnected from the coordinator, causing the coordinator to initiate the master's recovery. We find that RAMCloud can indeed return incorrect data; this bug was observed empirically and confirmed by the RAMCloud developers. Specifically, RAMCloud detects failures using a short timeout of hundreds of milliseconds; if the coordinator times out on a master, the coordinator starts data recovery, which is very fast. Because the timeout is short and recovery is fast, the entire process may complete before the old master realizes that it was replaced, resulting in two masters: a split-brain scenario. Intuitively, the issue is that RAMCloud does not make its suspicion of failure definitive (e.g., by waiting for the old master to shut down) before acting on that suspicion.

[7] RAMCloud uses ZooKeeper but only for coordinator failures [59, §3.10].

Algorithm	**Aab (§7.1)**	**Zab [39]**
Description length	half page	~3 pages
# phases	2	3
# roundtrips on recovery	2	3
# message types	3	9
# timestamps/counters	1	2
At most one leader?	yes	no
failures (f) tolerated relative to total (n)	$f<n$	$f<n/2$

Figure 8—Comparison of atomic broadcast with and without definitive reports. Aab uses Albatross (and would be similar if it used any other membership service), and Zab [39] uses majority-based agreement; both algorithms are described in the text.

We replaced RAMCloud's failure detector with Albatross (65 lines of C++) and found that RAMCloud then worked correctly: when the master is reported as "disconnected", it is excluded and cannot serve clients, by Correspondence and Isolation (§4). This benefit is not unique to Albatross; other membership services would eliminate this error too.

How do Albatross's guarantees simplify algorithm design? As another case study, we examine *atomic broadcast*: it is a building block of many distributed systems, and it has solutions with and without definitive reports [21]. We specifically compare (a) *Zab* [39], a protocol that uses majority-based agreement (as opposed to definitive reports), and (b) *Aab*, a protocol that uses Albatross.

Zab: atomic broadcast without definitive reports. Zab [39] takes a standard approach, which we briefly summarize here. A leader orders messages. Because partitions can result in multiple leaders (one leader becomes disconnected, another leader is elected, and the original reconnects), the protocol relies on a majority (quorum) of processes to approve leader actions. As a result, if two leaders try to act, only one succeeds in getting approval from a majority.

Aab: atomic broadcast under Albatross. Under Albatross, processes can select a unique leader by picking the smallest process id among processes that Albatross considers to be "connected". This scheme works because, if there could be two non-excluded leaders at the same time, let p be the one with higher id; then p considers the other leader as "disconnected", otherwise it would not have picked itself as leader. Thus, by Correspondence (§4), the other leader is excluded—contradiction. Thus, we have essentially unique leaders. We say "essentially" because there could be many self-styled leaders; however, all but one will be excluded.

Given (essentially) unique leaders, we can implement atomic broadcast using a sequencer-based algorithm [21], adapted to use Albatross. The algorithm proceeds in periods; each period has a unique leader (chosen as described above). In each period, a process that wants to broadcast a message sends it to the leader and waits for an acknowledgment; if the leader changes, the process resends to the new leader (in a new period). The leader handles each period in two phases, recovery and order. In the recovery phase, the leader completes the broadcast of pending messages from prior periods (if any).

where injected?	what failure is injected?	what does the failure model?
network	link failure	network partition
network	switch failure	network partition
network	misconfiguration that causes a partition	operator error
network	host floods UDP traffic	sudden traffic spike
network	dropped OpenFlow messages	problems in the SDN
network	spurious failure event	link flapping
end-host	process crash (segfault)	problem in the application
end-host	host crash (kernel panic)	machine crash or reboot
Albatross	crash of host module	bug in host module
Albatross	crash of leader in manager	bug in manager

Figure 9—Panel of synthetic failures. We inject failures in the network, at the end hosts, and into Albatross itself.

In the order phase, the leader serves as a sequencer: it gets a new message to broadcast, assigns it a sequence number, and sends it to processes for delivery. Processes then deliver the messages in sequence number order.

Comparison. Figure 8 compares the two algorithms. Aab has a smaller description, fewer phases, fewer round-trips, fewer message types, and fewer counters for ordering messages. Moreover, it tolerates the failure of all but one process; Zab, by contrast, tolerates the failure of fewer than half of the processes. (Equivalently, to tolerate f failures, the Albatross-based Aab requires $f + 1$ processes, whereas Zab requires $2f+1$ processes.) The fundamental source of these differences is that Zab is built on majority-based agreement, which brings complexity, as noted earlier (§2.2).

Albatross vs. ZooKeeper vs. consensus vs. atomic broadcast. The preceding comparison immediately raises a question. Namely, *Albatross also uses majority-based techniques internally*—in fact, the consensus-based algorithm for replicating the manager (§3, §5.3) has the same qualitative complexity as Zab. So why is this fact omitted in the Aab-vs-Zab comparison? Because under Albatross, the complexity is localized to the manager, and handled once; the clients of Albatross are not exposed to the complexity, and the additional resource cost is amortized over all clients of Albatross.

But can't the same kind of amortization work for Zab? Yes and no. On the one hand, ZooKeeper's lease server abstraction is built on Zab (Zab stands for "ZooKeeper atomic broadcast"; Zab is used to order commands to a replicated lease server state machine), and the intent is that many different applications can be clients of ZooKeeper's lease server. On the other hand, ZooKeeper cannot achieve the same performance under the same number of clients as Albatross. The reason is that short leases require frequent polling, which can overwhelm a server with many clients; this is demonstrated in Section 7.2.

Can ZooKeeper be modified to use Albatross? Yes. ZooKeeper is built on an atomic broadcast interface, which is implemented by Zab (as noted above). We could replace Zab with Aab. However, the resulting system would inherit the disadvantages of leases (§2.2, §2.3).

failure type	action taken by Albatross
link failure	the network interface reports link failures (§5.2); the mgr. detects a partition, enforces an excluded set, and reports it to clients (Figure 5)
switch failure	ditto
network misconfiguration	detected by client timeout (§3) and enforced by the manager's backstop logic (Figure 6)
network flooding	no failure is detected
dropped OpenFlow messages	detected by an OpenFlow timeout in the SDN controller; the controller treats this as a switch failure and reports it to the manager as multiple link failure events (Figure 5)
spurious failure event	handled as a link failure (see above)
host crash	detected with an end-host failure event (§5.2), and enforced by the manager (Figure 5)
process crash	Falcon spy [50] detects and reports failure (§6.3)
crash of host module	detected by backstop timeout (§3) and enforced by the manager (§5.3)
crash of manager replica + partition	replication library (§6.4) elects a new leader (§5.3), then the manager handles the failure as above

Figure 10—Albatross's reaction to the failure panel (Figure 9). Albatross detects all failures save network flooding (a non-failure), and its enforcement actions affect only applications that use it.

Can Albatross be built atop ZooKeeper? Yes. Albatross could replicate its manager using ZooKeeper's lease servers (or Zab directly). This represents an alternative instantiation of Albatross; it is essentially equivalent to the one covered in the rest of this paper.

7.2 Is Albatross a good membership service?

We now experimentally investigate the qualities of Albatross: (a) how it responds to failures, (b) how its timeliness compares with two baseline mechanisms, (c) how well it limits interference, and (d) what system resources it uses. The experiments use a panel of synthetic failures, depicted in Figure 9. These failures model problems in the network, at end-hosts, and in Albatross itself; link and switch failures are derived from our failure analysis (§2.1). While deploying Albatross on physical hardware and measuring its response to failures in the wild would be better than a synthetic evaluation, this is beyond our scope, as we currently seek a more basic understanding of how Albatross performs. Thus, this evaluation should be read as suggestive rather than conclusive.

How does Albatross respond to failures? We run an experiment where a *client* process monitors a remote *target* process; we inject a failure of some chosen type, affecting the target process, and we record Albatross's response. We repeat the experiment 25 times for each failure type.

We find that Albatross reacts the same way in the 25 repetitions for a given failure type; the reactions for each failure type are in Figure 10.

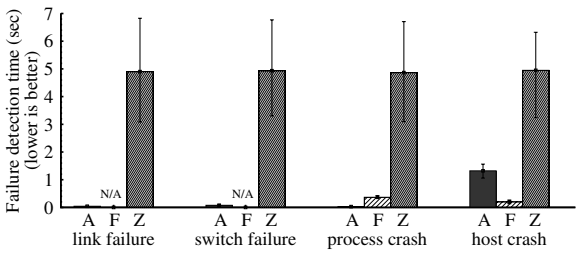


Figure 11—Detection time and coverage of Albatross (**A**), compared to Falcon [50] (**F**) and ZooKeeper [36] (**Z**). Error bars are minimum and maximum observed detection times. Two of Falcon's bars are labeled N/A because it does not detect link or switch failures. Albatross detects failures quickly by using information at end-hosts and in the network. ZooKeeper's detection time reflects its timeout (4s); a shorter one causes overload (see text).

How does Albatross compare with baseline mechanisms? We take as baselines (1) ZooKeeper [36], and (2) Falcon [50], both of which are described in Section 2.[8] For each failure that Albatross detects (without using backstop timeouts), we repeat the aforementioned experiments 100 times and measure the *detection time* from when the failure occurs to when it is reported to the application. We experiment with Albatross and with ZooKeeper. For Falcon, we report published results [50] (because two of Falcon's spies are incompatible with the testbed used for Albatross).

Figure 11 shows the results. Network problems are detected by Albatross quickly, usually in less than a second. The specifics of these numbers depend on the implementation of our testbed's switches, which are software; deploying Albatross on real hardware may have different performance characteristics, though we expect the order of magnitude will be similar.

Process failures are detected by Falcon and Albatross quickly; Albatross is faster than Falcon here (even though Albatross uses a Falcon spy to detect these failures) because Falcon's published results include a delay for confirming that a process has left the process table whereas Albatross needs only to install an iptables rule (§6.3).

On host failures (e.g., kernel panics), Albatross takes 1s longer than Falcon; the difference is that Albatross detects host failures using OpenFlow rule timeouts (§6.2), which have a minimum duration of 1s. Unlike Albatross, Falcon cannot detect switch or link failures [50].

ZooKeeper's detection speed reflects its timeout, which we configure to be 4 seconds, as suggested in its tutorial [1]. This choice is not arbitrary: if one lowers ZooKeeper's timeout to match Albatross's detection speed, ZooKeeper would be overloaded by keep-alives. To establish this, we experiment with ZooKeeper. We find, first, that ZooKeeper can monitor 1500 targets, each using a 4s timeout on their leases. But when we reduce the timeout to 500 ms, ZooKeeper drops the connections of about 70 targets, even though the network is not saturated. We believe this effect is similar to Burrows's observations [13]: timeouts shorter than 12s overwhelmed Chubby's servers in Google's clusters (which monitors many more targets). In contrast, we find that Albatross's manager can monitor over 1500 targets. Essentially, ZooKeeper polls clients with ping messages whereas Albatross watches for the *causes* of dropped pings (crashes, partitions, etc.), and can thus react quickly.

[8] We do not use Pigeon [49] as a baseline because it lacks definitive reports for network failures and because it targets Layer 3 networks.

max additional rules installed at a switch	
rules installed	1 rule
CPU *usage per component (§5)*	
host module	1.8 %
manager	0.03 %
bandwidth used	
at each end-host	61.0 kBps
at manager	6.9 kBps

Figure 12—Summary of Albatross's costs under link failure. Albatross uses few resources. Scalability is discussed in the text.

How well does Albatross limit interference? We evaluate whether some common network behaviors might cause Albatross to disconnect processes without cause. We inject two non-failures into our testbed: (a) heavy traffic (modeling congestion) and (b) a spurious link failure event (modeling link flapping), for a link whose removal splits the network. We observe that Albatross does not disconnect processes under heavy traffic. Albatross does not detect a problem because the duration of the spike in traffic is less than the client process's end-to-end timeout. Albatross does disconnect under the spurious failure. While this behavior is not ideal, it is not disastrous because, first, a known down link may be better than persistent link flapping; second, Albatross does not interfere if there are alternate paths or the link is not used by Albatross processes; and third, applications can reconnect (§5.4). Reconnection takes about one second in our experiments.

What are Albatross's costs? We measure Albatross's resource cost for detecting and enforcing a partition for a single application. Figure 12 shows the results. As expected, Albatross installs one rule per application id before reporting the target process as "disconnected".

We must also consider what happens when there are more applications and hosts. In general, the number of rules grows with the number of disconnected processes; for example, a switch with 40 disconnected end-hosts, each with 20 distinct applications, would have 800 rules. On the one hand, numbers like these are acceptable: the HP ProCurve J9451A switch, for example, has capacity of 1500 OpenFlow rules [35]. On the other hand, the linear in-network costs could become undesirable. In that case, Albatross could reduce the number of rules that it uses; on links that connect to end-hosts, it could block at the granularity of epochs instead of appids (§5.3), at the cost of possibly blocking additional processes.

Albatross uses few resources at the manager replicas in terms of CPU and network bandwidth. Albatross's cost at

end-hosts is higher, as the host module generates heartbeat packets (§6.2). However, the effect is local: these packets are dropped by a host's switch before entering the network.

Albatross is implemented with 4044 lines of C++ code.

8 Discussion and future work

Does Albatross have all the features of existing membership services? While Albatross implements the basics of a membership service, there are other features of existing membership services that Albatross does not implement, including metadata storage [13, 36], message ordering [11, 16], and access control [36]. However, Albatross can be used within existing services to help reduce both failure detection time (§7.2) and implementation complexity (§7.1).

Does Albatross require SDNs? While the current implementation of Albatross uses OpenFlow, Albatross requires relatively few things from the network: the ability to receive failure events and to block traffic based on packet fields. These requirements are made explicit by the network interface (§5.2), and Albatross can work in any network where this interface can be implemented.

Is the SDN controller a single point of failure? This issue is mostly orthogonal to Albatross. Albatross currently uses NOX, which is centralized and thus a single point of failure. However, Albatross could instead use recent fault-tolerant controllers (see Section 9).

Must Albatross repurpose the source MAC field? Albatross's embedding of process identifiers in packets' source MAC field (§6.1) is not fundamental. Albatross could use other space in packets: MPLS labels, a shim layer for Albatross, bits in an RPC header, etc. The only requirement is that switches can filter packets based on these fields.

How does Albatross's MAC rewriting scheme affect existing Layer 2 protocols? Under existing Layer 2 protocols, such as IEEE 802.1d [37], switches will use the source MAC addresses of incoming packets to learn the mapping between MAC addresses and output ports, for future forwarding decisions. Since Albatross's MAC-rewriting scheme creates source addresses that will never be used as destination addresses (§6.1), a Layer 2 protocol deployed alongside Albatross should be modified to never learn from these packets. Fortunately, Albatross works in the context of SDNs, so many Layer 2 protocol changes would require only software changes at the SDN controller.

Can Albatross work across data centers? One challenge is that wide-area delays will worsen detection time when the monitoring and target processes are far apart. Another challenge is finding reasonable guarantees for Albatross to provide when the data centers are partitioned. Addressing these challenges is future work.

Can Albatross work with virtual machine migration? Albatross assumes that processes and end-hosts remain stationary, which conflicts with virtual machine migration [17]. This issue is surmountable, if the manager and migration mechanism collaborate to migrate filter rules. This, too, is future work.

Does Albatross consider network policy? Albatross models only the *physical* network topology (§5.3). Yet policies (e.g., ACLs) can constrain communication. The Albatross manager might thus be unable to detect unreachability: it might think a path exists, when in reality it is prohibited. This problem would be handled by the client's backstop timeout (§3, §5.3). Using policy information in Albatross is future work.

Can cooperating applications have inconsistent views of the network? As mentioned in Section 5.3, one can think of Albatross as virtualizing partitions. A natural question is whether the discrepancies in the views of applications create problems when applications communicate. The answer is no. Albatross guarantees that, if a process is partitioned away, it is partitioned for all applications.

What are the security implications of Albatross? Processes can block any Albatross-enabled process by starting and never canceling an end-to-end timeout (Figure 2). Adding access control to Albatross's API is future work.

Does Albatross contradict the end-to-end argument? No, because Albatross's guarantees are (a) about the state of the network and (b) require help from the network, two cases that fall outside the end-to-end argument's jurisdiction.

9 Related work

Albatross leverages software-defined networking [33, 61] (as described earlier). Albatross also builds on the distributed systems literature, borrows from the networking literature, and relates to work at their intersection.

Distributed systems. Earlier (§2.2), we walked through distributed systems solutions that relate to Albatross generally. Here, we delve into closely related approaches.

Albatross can be seen as a type of failure detector, a service that indicates the operational status of remote processes. This service has a well-developed theory (e.g., [14]), including some extensions for network partitions [3, 5], and a well-developed practice [9, 15, 34, 67], based on timeouts (§2.2). Unlike Albatross, this work does not leverage information and mechanisms in the network.

Falcon [50] and Pigeon [49] are failure detectors that, like Albatross, rely on local (or inside) information. Albatross has some debts to Falcon, most notably the spy module to detect process crashes (§6.3). A minor difference is that Albatross better limits interference (Falcon kills end-hosts, whereas Albatross merely disconnects processes). The major innovations over Falcon are: Albatross's handling of network failures (Falcon handles only process failures; it hangs if there are partitions), its precisely articulated contract, its design (process naming, fine-grained dropping using SDNs, etc.), and using SDNs to enhance classical distributed systems.

Like Albatross, Pigeon [49] is designed to handle network failures; unlike Albatross, Pigeon does not leverage SDNs. Another major difference is that Pigeon does not report failures definitively: under network problems, its interface pro-

vides only hints (§2.2). We built a membership service atop Pigeon, in the form of a lease server [49, §5.2]. Under host failures, inside information allows quick lease breakage; under *network* failures, however, this membership service waits for the lease to expire, as in ZooKeeper. One might be able to build a better membership service atop Pigeon; we plan to investigate this in future work. But even if such an improvement is possible, it would represent a different design point from our work here. Among other things, we expect the response times of this membership service to be longer (since it would achieve definitiveness via agreement as opposed to killing).

Various systems introduce abstractions that can be used in place of a failure detector. For example, cluster management services (ZooKeeper, Chubby, etc.) can be used to assist primary-backup replication, as explained in Section 2.3. Another example is FUSE [25], which reports the failures of process groups in overlay networks. Unlike Albatross, FUSE has symmetric guarantees for failure notification. However, the semantics of these reports cannot be used to implement primary-backup (since FUSE may report failure even if the primary and the backup are both up). One body of work that deserves special mention is *group communication services* [11, 16] (GCS); these maintain a *view* of processes and provide multicast within a view. GCS have an exclusion concept similar to Albatross, but the mechanisms are different: a GCS uses timeouts and distributed algorithms (§2.2) while Albatross observes and modifies the network.

Another related system is Fault-Tolerant CORBA (FT-CORBA) [2]. FT-CORBA has a hierarchical monitoring scheme, which is reminiscent of how Albatross implements end-host failure events (§6.2); FT-CORBA also includes object-specific monitoring functions, which are similar to the spy that Albatross borrows from Falcon (§6.3). However, Albatross and FT-CORBA have different goals and mechanisms: Albatross provides a membership service by leveraging SDN whereas FT-CORBA replicates objects.

Networking. The general area of resilience in networking has received much research attention. This work is largely orthogonal to Albatross, as Albatross concerns how to reliably report partitions to applications. However, some of this work has a similar ethos to Albatross. For instance, NetPilot [72] seeks to turn partial failures into total failures (called *failure mitigation*). Nevertheless, the two systems take different approaches, in the service of complementary goals: NetPilot restarts switches and ports, to mitigate network failures, whereas Albatross blocks traffic to make its reports accurate, when such failures do occur.

A large body of work (far more than we can cover in depth, but see [7, 8, 18, 19, 22, 31, 42, 43, 52, 53, 66, 71, 75, 76]) is concerned with extracting intelligence about the network, and reporting it to operators or applications. Although some of Albatross's elements are reminiscent of some of this work, generally the goals are different. For instance, some work [65, 70] monitors routing state to track topology (as does the Albatross manager), but their goal is different: analysis and diagnosis for researchers and operators. In Network Exception Handlers (NEH) [40], the network notifies hosts of state changes, but the purpose is end-host participation in traffic engineering. Like Albatross, Packet Obituaries (POs) [6] report network failure information to end-hosts, but POs do so at a different level of abstraction (which packets were dropped versus which processes are reachable).

Distributed systems and networking. Research that combines distributed systems and networking tends to apply distributed systems techniques to make better networks (whereas Albatross works the other way around).

Consistent networking aims to keep the network in a valid state at all times, under configuration changes. For instance, consensus routing [38] uses Paxos [46] to apply updates to BGP routers to avoid black holes and loops. More recent work has examined primitives for consistent updates to OpenFlow networks, to preserve routing state [63] and bandwidth guarantees [28]. These goals are different from Albatross's.

Distributed SDN. Some researchers have used techniques from distributed systems to replicate and distribute the SDN controller [20, 23, 24, 44, 68, 73]. This work would complement Albatross, as noted in Section 8.

10 Conclusion

> For all averred, I had killed the bird
>
> —"The Rime of the Ancient Mariner", Samuel Taylor Coleridge

Albatross leverages software defined networks to give distributed applications reports about the reachability of their component processes. These reports are timely and definitive, and cover network problems—a combination that we believe is new in membership services. Furthermore, Albatross as built handles failures at end-hosts, making it a complete solution for failure detection in a data center environment.

Albatross also brings up a potential direction for future work. A number of projects have applied distributed systems techniques to obtain better networks. However, we think that Albatross is the first to apply modern networking techniques to refine the guarantees of distributed systems. Perhaps there are other opportunities along these lines.

Acknowledgments

Youngjin Kwon helped implement an early prototype of Albatross. Many people helped to improve this work, including Sebastian Angel, Mahesh Balakrishnan, Manos Kapritsos, Jeff Mogul, John Ousterhout, Rama Ramasubramanian, Ryan Stutsman, Yang Wang, Emmett Witchel, and Edmund L. Wong. The presentation of this paper was greatly improved by the careful comments of Dejan Kostić, Jinyang Li, Scott Shenker, Riad Wahby, and the anonymous reviewers of EuroSys, OSDI, and SIGCOMM. The research was supported in part by NSF grants CNS-1055057, CNS-1040083, and CCF-1048269.

References

[1] `http://zookeeper.apache.org/doc/current/zookeeperStarted.html`.

[2] Faut tolerant CORBA. OMG Specification formal/2010-05-07, Object Management Group, 2010.

[3] M. K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, June 1999.

[4] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *International Conference on Software Engineering (ICSE)*, pages 562–570, 1976.

[5] L. Arantes, P. Sens, G. Thomas, D. Conan, and L. Lim. Partition participant detector with dynamic paths in mobile networks. In *IEEE International Symposium on Network Computing and Applications (NCA)*, pages 224–228, July 2010.

[6] K. Argyraki, P. Maniatis, D. Cheriton, and S. Shenker. Providing packet obituaries. In *ACM Workshop on Hot Topics in Networks (HotNets)*, Nov. 2004.

[7] H. Ballani and P. Francis. Fault management using the CONMan abstraction. In *INFOCOM*, Apr. 2009.

[8] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 335–348, Apr. 2009.

[9] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *International Conference on Dependable Systems and Networks (DSN)*, pages 354–363, June 2002.

[10] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)*, 9(3):272–314, Aug. 1991.

[11] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 123–138, Nov. 1987.

[12] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 141–154, Apr. 2011.

[13] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–350, Dec. 2006.

[14] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

[15] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(5):561–580, May 2002.

[16] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, Dec. 2001.

[17] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, May 2005.

[18] D. D. Clark, C. Partridge, J. C. Ramming, and J. T. Wroclawski. A knowledge plane for the Internet. In *ACM SIGCOMM*, pages 3–10, Aug. 2003.

[19] E. Cooke, R. Mortier, A. Donnelly, P. Barham, and R. Isaacs. Reclaiming network-wide visibility using ubiquitous end system monitors. In *USENIX Annual Technical Conference*, June 2006.

[20] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling flow management for high-performance networks. In *ACM SIGCOMM*, pages 254–265, Aug. 2011.

[21] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, Dec. 2004.

[22] A. Dhamdhere, R. Teixeira, C. Dovrolis, and C. Diot. Troubleshooting network unreachabilities using end-to-end probes and routing data. In *ACM Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, Dec. 2007.

[23] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an elastic distributed SDN controller. In *SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pages 7–12, Aug. 2013.

[24] C. Dixon, H. Uppal, V. Brajkovic, D. Brandon, T. Anderson, and A. Krishnamurthy. ETTM: A scalable fault tolerant network manager. In *Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2011.

[25] J. Dunagan, N. J. A. Harvey, M. B. Jones, D. Kostić, M. Theimer, and A. Wollman. FUSE: Lightweight guaranteed distributed failure notification. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 151–166, Dec. 2004.

[26] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computers*, 52(2):99–112, Feb. 2003.

[27] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[28] S. Ghorbani and M. Caesar. Walk the line: Consistent network updates with bandwidth guarantees. In *SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pages 67–72, Aug. 2012.

[29] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):48–51, June 2002.

[30] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *ACM SIGCOMM*, pages 350–361, Aug. 2011.

[31] S. Goldberg, D. Xiao, E. Tromer, B. Barak, and J. Rexford. Path-quality monitoring in the presence of adversaries. In *SIGMETRICS*, pages 193–204, June 2008.

[32] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 202–210, Dec. 1989.

[33] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an operating system for networks. *ACM Computer Communications Review (CCR)*, 38(3):105–110, July 2008.

[34] N. Hayashibara, X. Défago, R. Yared, and T. Katayama. The ϕ accrual failure detector. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 66–78, Oct. 2004.

[35] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity switch models for software-defined network emulation. In *SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pages 43–48, Aug. 2013.

[36] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX Annual Technical Conference*, pages 145–158, June 2010.

[37] Standard for local an metropolitan area networks: media access control (MAC) bridges. IEEE Standard 802.1d, Institute of Electrical and Electronics Engineers, 2004.

[38] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani. Consensus routing: The Internet as a distributed system. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 351–364, Apr. 2008.

[39] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *International Conference on Dependable Systems and Networks*

(DSN), pages 245–256, June 2011.

[40] T. Karagiannis, R. Mortier, and A. Rowstron. Network exception handlers: Host-network control in enterprise networks. In *ACM SIGCOMM*, pages 123–134, Aug. 2008.

[41] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: a scalable Ethernet architecture for large enterprises. In *ACM SIGCOMM*, pages 3–14, Aug. 2008.

[42] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. IP fault localization via risk modeling. In *Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[43] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Detection and localization of network black holes. In *INFOCOM*, pages 2180–2188, May 2007.

[44] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: a distributed control platform for large-scale production networks. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 351–364, Oct. 2010.

[45] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, July 1978.

[46] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998.

[47] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 312–313, Aug. 2009.

[48] E. K. Lee and C. Thekkath. Petal: Distributed virtual disks. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 84–92, Dec. 1996.

[49] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Improving availability in distributed systems with failure informers. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 427–442, Apr. 2013.

[50] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the FALCON spy network. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 279–294, Oct. 2011.

[51] Linux-HA, High-Availability software for Linux. `http://www.linux-ha.org`.

[52] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 367–380, Nov. 2006.

[53] H. V. Madhyastha, E. Katz-Bassett, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane Nano: path prediction for peer-to-peer applications. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 137–152, Apr. 2009.

[54] D. Mazières. Paxos made practical. `http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf`, as of Sept. 2011.

[55] T. Muraus. Service registry behind the scenes why we built it. `http://www.rackspace.com/blog/service-registry-behind-the-scenes-why-we-built-it`, Nov. 2012.

[56] NOX Zaku with OpenFlow 1.2 support. `http://github.com/CPqD/nox12oflib`.

[57] B. M. Oki and B. Liskov. Viewstamped replication: A general primary copy. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 8–17, Aug. 1988.

[58] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–309, June 2014.

[59] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–41, Oct. 2011.

[60] OpenFlow 1.2 Software Switch. `http://github.com/CPqD/of12softswitch`.

[61] Openflow. `http://www.openflow.org/`.

[62] Kernel based virtual machine. `http://www.linux-kvm.org/`.

[63] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *ACM SIGCOMM*, pages 323–334, Aug. 2012.

[64] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[65] A. Shaikh and A. Greenberg. OSPF monitoring: Architecture, design, and deployment experience. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 57–70, Mar. 2004.

[66] A. Shieh, E. G. Sirer, and F. B. Schneider. NetQuery: A knowledge plane for reasoning about network properties. In *ACM SIGCOMM*, pages 278–289, Aug. 2011.

[67] K. So and E. G. Sirer. Latency and bandwidth-minimizing failure detectors. In *European Conference on Computer Systems (EuroSys)*, pages 89–99, Mar. 2007.

[68] A. Tootoonchian and Y. Ganjali. HyperFlow: A distributed control plane for OpenFlow. In *Internet Network Management Workshop / Workshop on Research on Enterprise Networking (INM/WREN)*, Apr. 2010.

[69] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–104, Dec. 2004.

[70] D. Watson, F. Jahanian, and C. Labovitz. Experiences with monitoring OSPF on a regional service provider network. In *International Conf. on Distributed Computing Systems (ICDCS)*, pages 204–213, May 2003.

[71] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An information plane for networked systems. In *ACM Workshop on Hot Topics in Networks (HotNets)*, Nov. 2003.

[72] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang. NetPilot: Automating datacenter network failure mitigation. In *ACM SIGCOMM*, pages 419–430, Aug. 2012.

[73] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with DIFANE. In *ACM SIGCOMM*, pages 351–362, Aug. 2010.

[74] L. Zhang. Why TCP timers don't work well. In *ACM SIGCOMM*, pages 397–405, Aug. 1986.

[75] M. Zhang, C. Zhang, V. Pai, L. Peterson, and R. Wang. PlanetSeer: Internet path failure monitoring and characterization in wide-area services. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 167–182, Dec. 2004.

[76] Y. Zhao, Y. Chen, and D. Bindel. Towards unbiased end-to-end network diagnosis. In *ACM SIGCOMM*, pages 219–230, Sept. 2006.