The Dissertation Committee for Joshua Blaise Leners
certifies that this is the approved version of the following dissertation:

A new approach to detecting failures in distributed systems

Committee:

Lorenzo Alvisi, Supervisor

Marcos K. Aguilera

Vitaly Shmatikov

Michael Walfish

Emmett Witchel

A new approach to detecting failures in distributed systems

by

Joshua Blaise Leners, B.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2015

Acknowledgments

Now, at the end of writing my dissertation proper, I finally get to thank everyone who helped me along the way.

First, I would like to thank Marcos K. Aguilera and Michael Walfish for their mentorship and extensive help in preparing and presenting this research. Mike has pushed me to gain a deep appreciation for the connection between what I think and what I can describe, and has held me to a very high standard throughout my graduate career. Though the high bar has not always been pleasant, I have gained a new appreciation for careful consideration and clear communication. Marcos has helped me identify "bogus" arguments, and has shown me the surprising power of principled thinking about the design of computer systems.

I also thank the other students who helped the research of this dissertation. Wei-Lun Hung's multi-threading implementation in Falcon (Chapter 3) was crucial to our successful submission, as were Hao Wu's heroic implementation and evaluation efforts (both for submission and our camera-ready deadlines). Trinabh Gupta was instrumental in the design, implementation, and evaluation of Albatross and Pigeon (Chapters 4&5). I would also like to thank Trinabh for presenting Pigeon at NSDI 2013, thereby taking a major item off of my plate.

The reviewing process is not always pleasant, but it is usually helpful. Thus, I would like to than the anonymous reviewers of SOSP 2011, PODC 2012, SIGCOMM 2012,

OSDI 2012 SIGCOMM 2013, NSDI 2013, SIGCOMM 2014, OSDI 2014, and EuroSys 2014; this research would be much poorer without their help. To Marvin Theimer, Katerina Argyraki, and Dejan Kostić, who shepherded this research into three wonderful conferences, I also give thanks.

I thank the members of my thesis committee for (miraculously) appearing in the same room at my defense, along with their extremely helpful questions and comments. This was a rare event made even more special by the fact that three of them were remote from UT Austin. I especially thanks Lorenzo Alvisi for serving as my thesis supervisor in Mike's stead. I would also like to thank the friends and family who traveled for my defense.

Several people gave helpful and extensive comments on the drafts of this dissertation: Sebastian Angel, Kendra Garwin, Mary van Valkenburg, and Riad Wahby.

Learning how to speak and write correctly has been a major part of my graduate career, and in this I would like to give special thanks for the help of Marcos K. Aguilera, Lorenzo Alvisi, Mike Dahlin, Mark Handley, Brad Karp, Jinyang Li, Michael Walfish, Damon Wischik, and all of the students who attended my practice talks at UT Austin, NYU, and UCL. I would particularly like to thank Damon for telling me that "A graph is an act of violence on the audience. It may be necessary, but do not inflict it lightly," and his overall influence on my presentation style.

My graduate experience would have been much worse without several groups of people. First, the professors who helped extend the foundations of my computer science knowledge: Lili Qiu, Vitaly Shmatikov, Lorenzo Alvisi, Brent Waters, Michael Walfish, Indrajit Roy, and Allen Emerson. Second, the administrators who greased and turned the bureaucratic wheels: Lindy Aleshire, Leslie Cerve, Lydia Griffith, Sara Strandtman, and Leah Wimberly. Finally, I would like to thank the students with whom I worked while in grad school outside the contest of this dissertation: Alex Benn, Allen Clement,

Alan Dunn, Ann Kilzer, Manos Kapritsos, Youngjin Kwon, Michael Z. Lee, Sangmin Lee, Prince Mahajan, Srinath Setty, and Ed Wong.

I would not have been propelled to follow computer science in graduate school had it not been for two influential undergraduate professors, Todd Dupont and Anne Rogers, to whom I also give thanks.

Now we get to the potpourri from the many text files I have that track thanks. I thank Jeff Mogul for the connection between our approach and Lixia Zhang's SIGCOMM paper about end-to-end timeouts [115]. I also thank Carmel Levy for his inspirational backronym for Falcon (fast and lethal computer observation network), though we did tweak it for the paper.

Finally, I would like to thank my family and friends for their support during this long and uncertain process. Special thanks goes to my parents, Mark Leners and Mary van Valkenburg, have been supportive in my pursuit of a Ph.D. from the start, and Kendra Garwin, who has provided motivation and support during the preparation of this dissertation and its defense.

Joshua Blaise Leners

The University of Texas at Austin
August 2015

A new approach to detecting failures in distributed systems

Publication No. _____

Joshua Blaise Leners, Ph.D.
The University of Texas at Austin, 2015

Supervisor: Lorenzo Alvisi

Fault-tolerant distributed systems often handle failures in two steps: first, detect the failure and, second, take some recovery action. A common approach to detecting failures is end-to-end timeouts, but using timeouts brings problems. First, timeouts are inaccurate: just because a process is unresponsive does not mean that process has failed. Second, choosing a timeout is hard: short timeouts can exacerbate the problem of inaccuracy, and long timeouts can make the system wait unnecessarily. In fact, a good timeout value—one that balances the choice between accuracy and speed—may not even exist, owing to the variance in a system's end-to-end delays.

This dissertation posits a new approach to detecting failures in distributed systems: use information about failures that is local to each component, e.g., the contents

of an OS's process table. We call such information *inside information*, and use it as the basis in the design and implementation of three failure reporting services for data center applications, which we call Falcon, Albatross, and Pigeon.

Falcon deploys a network of software modules to gather inside information in the system, and it guarantees that it never reports a working process as crashed by sometimes terminating unresponsive components. This choice helps applications by making reports of failure reliable, meaning that applications can treat them as ground truth. Unfortunately, Falcon cannot handle network failures because guaranteeing that a process has crashed requires network communication; we address this problem in Albatross and Pigeon. Instead of killing, Albatross blocks suspected processes from using the network, allowing applications to make progress during network partitions. Pigeon renounces interference altogether, and reports inside information to applications directly and with more detail to help applications make better recovery decisions.

By using these services, applications can improve their recovery from failures both quantitatively and qualitatively. Quantitatively, these services reduce detection time by one to two orders of magnitude over the end-to-end timeouts commonly used by data center applications, thereby reducing the unavailability caused by failures. Qualitatively, these services provide more specific information about failures, which can reduce the logic required for recovery and can help applications better decide when recovery is *not* necessary.

Contents

Chapter 1

Introduction

How can we build reliable systems out of unreliable components?

Answering this question amounts to constructing *fault-tolerant systems*; such systems pervade our lives: from engineering (suspension bridges support traffic even if a single nut is missing), to commerce (banks remain open even if tellers need to stay home sick), and even entertainment (an understudy steps in if an actor literally breaks a leg). Likewise, fault tolerance permeates the design of computer systems: the layout of circuits that are correct despite flaws in silicon, the codes and protocols that carry data through an error-prone medium, and the websites from the Internet that are (almost) always available.

In this dissertation, we restrict our focus to a single kind of computer system: fault-tolerant distributed systems. This choice is motivated by the advent of data centers as a substrate for building highly available web services. The challenge is that data centers are often built with low-cost and failure-prone components, but the reliability of services they host is paramount: these services are replacing desktop applications in many domains and any downtime means losing users, revenue, and trust.

The fault-tolerant distributed systems that underly these services often handle failures in two steps: first, determine that a component process has failed and, second, take some recovery action to mitigate that failure. We are interested in building systems that handle failures *quickly*, since users are willing to wait mere seconds on web services [41]. To this end, this dissertation focuses on the problem of *fast failure detection*,

though we acknowledge the interplay between these two steps and note complementary work on fast failure recovery [16, 17, 30, 83].

We hypothesize that the relative lack of attention to fast failure detection owes to the fundamental difficulty of the problem and its widespread (but imperfect) solution: timeouts. The challenging kernel in the problem of detecting failures is distinguishing between that which has truly failed and that which is just slow. This difficulty is relatable: should you continue waiting for a delayed bus or catch a cab, how many times should you let a phone ring before hanging up, should you wait for your date's arrival or start drinking alone?

The widespread-but-imperfect solution to this hard problem is to use a timeout: after some fixed period of time, start treating the unresponsive party as failed (catch a cab, hang up, or drown your sorrows). In fault-tolerant distributed systems this solution is called *end-to-end timeouts*; with end-to-end timeouts, a process of the system considers another process to be failed if that process is unresponsive for some length of time.

End-to-end timeouts are problematic for two reasons. First, timeouts are inaccurate: just because a timeout fires does not mean there is a failure. Such mistakes can cause incorrect behavior if handled haphazardly; for example, in a system where some process serves as *backup* for some *master* process, the backup might take over for a functioning master, thereby causing a *split-brain scenario* where two processes believe that they have exclusive access to some shared resource (e.g., a database). To avoid incorrect behavior, applications currently cope with inaccuracy from end-to-end timeouts with several techniques; the common theme among these techniques is that they convert inaccurate suspicions into something that the distributed system can treat as ground truth (e.g., by requiring a majority of processes to agree that a suspected process should be ostracized or by forcefully terminating suspected processes).

The second problem with timeouts is that they trigger a troublesome trade-off: a long timeout increases a system's delay in responding to failure, but a short timeout exacerbates the problems of inaccuracy. To make matters worse, there may not even *be* a good timeout value [115]: exponential backoff in the network, disk access, and scheduling can all contribute delay that varies by several orders of magnitude even under normal conditions.

* * *

This dissertation addresses the shortcomings of end-to-end timeouts by proposing a new approach to detecting failures in distributed systems: *use information about failures that is local to each component.* We call such information *inside information*, and it is plentiful: applications have internal performance counters for tracking progress, operating systems have a process table that definitively lists working processes, and computer networks have dedicated protocols [18] for communicating the status of network components (switches, routers, and links) to network administrators.

We are interested in using inside information to build *failure reporting services* that aid distributed systems in handling failures; this approach brings three challenges:

Systematically collecting inside information. Abstraction conveniently hides the messier details of a system's internals, but gathering inside information requires sifting through exactly those details under adverse conditions (i.e., when there are failures). Not only must a failure reporting service's design be principled (so that the service's properties can be analyzed and its implementation improved), but its design and implementation must span many layers of the system. In our target setting, distributed systems in data centers, these layers of abstraction include applications, operating systems, virtual machines, networks, and more.

Defining an interface for reporting failures. A failure reporting service that uses inside information must coherently present a diverse set of information to applications. Even if the service simply uses the binary classification of "up" and "down", it still must define what those reports mean. Furthermore, binary classification could be inappropriate for certain kinds of failures. For example, the networks found in data centers are themselves fault-tolerant systems designed to carry traffic even if some switch or router fails, but the network may be unusable while it recovers. If two processes are disconnected during network recovery, how should the failure reporting service report their status? Should the service report both as "down", choose one over the other, or report both as "up"? Does it matter whether the network's recovery is fast or slow?

Limiting negative impact. In addition to the positive impact of providing information about failures quickly, a failure reporting service can have two negative impacts that should be limited. First, a failure reporting service can consume resources when gathering inside information, e.g., by constantly polling a component's status. Because failures occur infrequently, a failure reporting service should avoid tying up too many resources in their detection but without giving up fast detection time. Second, the failure reporting service might affect components of the system, so as to better determine their status, for example, by terminating an erratic component. Such termination should be rare, and it should be limited in its scope.

This dissertation examines the trade-offs in addressing these challenges through the design, implementation, and evaluation of three failure reporting services:

Falcon (Chapter 3). Falcon systematically collects inside information with a network of *spy modules*, or spies. Spies are layer-specific monitoring logic with a common infrastructure and protocol for reporting failures. Falcon exposes to its clients a *reliable failure detector* interface: clients query Falcon to learn about the failure of processes, and if Falcon reports that a process is "down" then that process has actually crashed. In underwriting this guarantee, Falcon grants its spies a license to kill when they suspect some layer has crashed, following the old technique of "shooting the other node in the head" (STONITH).[1] To limit its negative impact, Falcon employs a carefully designed callback-based architecture so that clients do not need to poll for fast reports, and it limits the scope of its killing to suspected components rather than whole machines.

Albatross (Chapter 4). Falcon's approach fails in the presence of network partitions: if a spy cannot communicate that a layer has crashed, Falcon cannot report any process failures caused by that crash. We observe that network partitions in data centers are generally small, so a failure reporting service might permanently disconnect the smaller partition to provide fast and reliable information about failures.

[1] STONITH appears to have existed as folklore knowledge since the 1970s, though to our knowledge no publication formally claims it as a contribution.

Of course, wholesale disconnection just to give reliable information would be insane; in the worst case, it could mean disconnecting hundreds of hosts to deal with a single problematic process. However, programmable network interfaces, such as those exposed by software defined network (SDNs), can allow a failure reporting service to target specific processes for disconnection and avoid disrupting processes that do not care about fast failure reporting (such as background data processing tasks).

Leveraging this observation, we build a failure reporting service called Albatross that disconnects only its monitored processes. Albatross also uses SDNs to systematically collect inside information about the failures of network elements and end-hosts; for handling process failures, Albatross borrows from Falcon. However, the semantics of permanent disconnection are different from those of a reliable failure detector: what happens when disconnected processes communicate with one another? To address this question, we formally specify the guarantees of Albatross's mechanism for making reports reliable. By leveraging SDNs (and Falcon's architecture for monitoring processes), Albatross responds to failures quickly and at a low cost; this combination is because Albatross re-uses the monitoring already done by SDNs and does not require clients to poll.

Pigeon (Chapter 5). Falcon and Albatross crash machines or disconnect processes to give reliable reports, but this is disruptive. We observe that distributed systems are already well-equipped to handle unreliable information due to the prevalence of end-to-end timeouts. This prompts the question of how a failure monitoring service can expose to applications its inside information (which may not be reliable), without requiring these applications to understand the specific details of such information? In Pigeon, we propose a new interface, called a *failure informer*, for that purpose. The failure informer distinguishes four *failure conditions*, where each may trigger a different kind of recovery action or even no recovery at all.

Pigeon systematically collects information about failures by extending Falcon's architecture to monitor the network (without assuming a SDN), with the goal of repurposing existing network monitoring where possible. This architecture avoids the negative impact of killing and helps keep Pigeon's costs low.

* * *

Before continuing, we share an important conclusion of this work:

Timeouts are inevitable. Even with inside information, we cannot eliminate the fundamental difficulty of distinguishing slow and failed components. For example, if an application fails to update a performance counter it is hard to tell if the process is deadlocked or, alternatively, if it is working hard on a challenging computation. However, inside information sidesteps this difficulty because the absolute costs in the trade-off for choosing a good timeout are mitigated by scale: a very long period of unresponsiveness locally may be very short from an end-to-end perspective (e.g., in Google's clusters, even heavily loaded machines rarely take longer than 10 milliseconds to schedule a runnable thread [108]). This observation can be used to set conservative timeouts locally that have much faster detection time than similarly conservative end-to-end timeouts. Furthermore, local timeouts can be set relative to resources other than real-time (e.g., how many CPU cycles a process has been given); this can prevent conditions like high load from being mistaken for failure.

Timeouts are also inevitable because inside information may lack *coverage*; even if all of the service's inside information says that a distributed system is healthy, that system may be stuck because of an unresponsive and unmonitored component. In this case, only end-to-end unresponsiveness indicates failure, and thus end-to-end timeouts must serve as a backstop mechanism for determining that something has failed, even when the vast majority of failures can be detected by inside information.

Roadmap. Chapter 2 contains an overview of work related to this dissertation. Chapters 3–5 describe Falcon, Albatross, and Pigeon. Albatross and Pigeon extend Falcon, and so assume familiarity with Chapter 3, but Chapters 4 & 5 can be read independently of each other. Chapter 6 summarizes and critiques all three systems, and so assumes familiarity with the rest of the dissertation.

Chapter 2

Related work

This chapter describes related work in failure detection, other services for distributed systems, and network monitoring. We also give an overview of work at the intersection of distributed systems and networking.

2.1 The theory and practice of failure detection

Chandra and Toueg formalized the theory of *failure detectors* in a seminal paper [21]. Specifically, they define a failure detector as a set of per-process oracles that each return a list of crashed processes when queried by their local process; the authors classify failure detectors based on the kinds of mistakes such lists can contain. Chandra and Toueg's main result is that *unreliable failure detectors*—those which sometimes mistake working processes for ones that have crashed—can be used to solve hard problems in distributed computing. The importance of this result is twofold: first, it succinctly captures the minimal assumptions required to solve certain fundamental problems; second, it introduces a simple and useful model for developing and reasoning about distributed algorithms.

Chandra and Toueg presented another important result in showing that *perfect failure detectors,* which never mistake working processes as crashed, permit simpler implementations of certain distributed algorithms. This result, along with the subsequent establishment of the theoretical advantages of *fast* perfect failure detectors [1], inspired

early work on this dissertation, specifically in the development of Falcon (Chapter 3) and its preceding workshop paper [2].

We were not alone in our inspiration: Chandra and Toeug's work inspired the design and implementation of both unreliable and perfect failure detectors. Much work has focused on choosing end-to-end timeout values used internally by failure detectors. Chen et al. [22] propose a failure detector that uses heuristics to select an end-to-end timeout adaptively based on delay and loss measurements. Bertier et al. [11] follow a similar method for estimating timeout values, but their failure detector makes different initial assumptions about the network. So and Sirer [102] developed a failure detector that uses assumptions about the underlying reliability of different components to minimize delay and bandwidth according to an optimization strategy. These works are complementary and could improve the choice of backstop timeout values in this dissertation's failure reporting services.

Fezter [39] designed and implemented a perfect failure detector using watchdogs, which are hardware components that reboot machines if they do not receive periodic messages. Similarly, the Linux-HA project [73] provides a service called Heartbeat, which is a failure detector based on end-to-end timeouts; this service can be configured to use a hardware watchdog (like Fetzer's failure detector), or to send RPCs that shut down suspected machines (real or virtual). Both of these mechanisms are a form of STONITH ("shoot the other node in the head"), which is an old technique for converting suspicion into fact. These works partially inspired our use of STONITH in Falcon (Chapter 3) and in Albatross (Chapter 4).

Accrual failure detectors [51] forgo binary classification and instead output a numerical value such that, roughly speaking, higher values mean there is a higher probability that a process has crashed. In practice [19], applications consider the output to be an indication of failure if it is above a certain threshold, but they can adjust this threshold on the fly; this is similar to how adaptive failure detectors work [11, 22], but with more flexibility given to the failure detector's clients. Expanding the failure detector interface to expose confidence is similar to the approach of Pigeon (Chapter 5), which expands the failure detector interface to expose uncertainty.

Other work has used the failure detector abstraction to improve other aspects of

end-to-end timeout based failure detection. For example, large process groups sometimes require pairwise monitoring, but the quadratic message complexity of a naïve implementation is costly. van Renesse et al. [106] address this problem with a failure detector that uses a gossip protocol to quickly and efficiently disseminate failure information. This technique is useful for failure detectors that use keep-alive messages, but it is unnecessary for the callback-based architectures used in this dissertation.

2.2 Other services for building distributed systems

Many services facilitate the design and implementation of distributed systems. These services operate at a different level of abstraction than failure reporting services, but they themselves are distributed systems and can thus benefit from fast failure reporting.

Group communication services [13, 23] (GCS) maintain a *view* of processes and provide multicast within a view. These services force applications into a particular design pattern, hence they are less general-purpose than a failure reporting service. However, a fast failure reporting service could improve how quickly GCS respond to failure (since GCS themselves are distributed systems). Early GCS used mechanisms of self-killing and exclusion for converting suspected failures into actual failures, and these mechanisms inspired some choices in our work; in fact, the exclusion mechanism of ISIS [12] is highlighted in Chandra and Toueg's treatise on failure detectors as an example of converting suspicion into fact [21, §9].

FUSE [37] tracks the mutual connectivity of a set of processes and guarantees that all processes will be notified if any process becomes crashed or partitioned. FUSE guarantees symmetric notification of failure; this contrasts Albatross's asymmetric guarantees (Chapter 4, §4.2). However, FUSE's guarantees are weaker than Albatross's: if a pair of processes sees that their group has failed, FUSE gives no guarantee that the failure is not a temporary network problem, and so the processes must take care to avoid a split-brain scenario.

Configuration services. Distributed systems often need to share small amounts of configuration information (membership lists, access control, meta-data, etc.); several services have been developed to facilitate this sharing for distributed systems built

in data centers [15, 53]. An important aspect of these services is that they include a mechanism for detecting failures by using leases [46]. If some *client processes* want to monitor a *target process*, the target process creates a special kind of meta-data at the configuration service. The target processes must periodically refresh this meta-data by sending a message to the configuration service, and if the target process fails to refresh by some deadline, the configuration service removes the meta-data (marking the process as crashed) and notifies the client processes.

Configuration services can be overloaded when configured to use short end-to-end timeouts to detect failures quickly (see Chubby [15, §2.8] and Chapter 4, §4.5.1). For this reason, we believe that configuration services can be enhanced by using a fast failure monitoring service, such as those described in this dissertation.

Replication libraries. State-machine replication [66, 97] is a common approach for building fault-tolerant distributed systems; in fact, Albatross (Chapter 4) uses it internally. The replicated state machine approach is predicated on the observation that a state machine's behavior is determined by its inputs and their order. Thus, building a reliable state machine is equivalent to replicating a log of its inputs; this approach requires detecting and recovering from failures, and can thus benefit from fast failure reporting services.

We make special note of two services related to those described in this dissertation: the fault-tolerant common object request broker architecture FT-CORBA [38] and the leader election service of Schiper and Toueg [95, 96]. FT-CORBA uses a monitoring hierarchy that is structurally similar to the systems described in this dissertation, but this hierarchy is restricted to specific layers of the system and uses only timeouts. The leader election service of Schiper and Toueg uses inside information to suspect failures, but the technique (tracking the presence of a process) is limited to a single component of the system.

2.3 Network monitoring

Many works in network monitoring [7, 9, 29, 34, 45, 61, 62, 116, 118] complement the failure reporting services described in this dissertation. Broadly speaking, network moni-

toring systems extract intelligence from network elements to aid diagnosis, a technique that failure reporting services could use to gather inside information from the network. Indeed, Pigeon (Chapter 5) borrows a network monitoring technique from Shaikh et al. [98, 99]. However, the goal of network monitoring is to help network operators perform diagnoses, while this dissertation aims to better inform distributed sysems about failures.

Providing a comprehensive service to distributed applications, using global information about the state of a network, is the goal of *information planes* [24, 110]. Works in this area include the Knowledge Plane [28], Sophia [94, 110] (which provides a distributed computational model for queries), iPlane [75, 76] (which helps end-host applications choose servers, peers, or relays, based on link latency, link loss, link capacity, etc.), and NetQuery [101] (which instantiates a Knowledge Plane under adversarial assumptions). These works are more flexible than the failure monitoring services in this dissertation (they usually expose an interface to arbitrary queries), while our goal is more focused: we aim to report failures to applications, a capability that these papers do not discuss.

More targeted works include Meridian [112] (a node and path selection service), King [49] (a latency estimation service), and Network Exception Handlers [58] (which delivers information from the network so end-hosts can participate in traffic engineering). Again, the goals of these systems are not that of this dissertation (informing applications about failures), and our work could be extended to use their techniques. In fact, our systems propagate inside information similar to the delivery mechanism in Network Exception Handlers.

While there are works that do report network failures and errors to end-hosts [5, 64, 104], they do not provide a comprehensive abstraction, in contrast to our goals. For example, Packet Obituaries [5] proposes that each dropped packet should generate a report about where the packet was dropped. Packet Obituaries uses different semantics for network failures (they are concerned with dropped packets in the Internet) and does not have coverage for host failures.

An important related system is NetPilot [113]. NetPilot aims to automate the network administrator's task of handling failures using *failure mitigation*, which is a form

of STONITH. Specifically, when NetPilot suspects that a network device has failed, it calculates the impact of deactivating that device and restarts the suspected device if the impact is sufficiently low. NetPilot's design is based on the fact that rebooting a device is often the first step a network administrator takes and that this step is easy to automate. NetPilot is complementary to this dissertation, though we note that Pigeon (Chapter 5) could be extended to take failure mitigation into account when estimating the expected duration of a failure (see Section 5.2).

2.4 Intersection of distributed systems and networking

Research that combines distributed systems and networking tends to apply distributed systems techniques to make better networks, while this dissertation uses information and mechanisms in the network to improve distributed systems. An exception to this generalization is recent work on using data center networks to improve state machine replication by making multicast more predictable [87]. This work has a similar ethos to Albatross (Chater 4) and was completed concurrently.

Consistent networking aims to keep the network in a valid state at all times, under configuration changes. For instance, consensus routing [56] uses state machine replication to apply updates to BGP routers to avoid black holes and loops. More recent work has examined primitives for consistent updates to OpenFlow [85] networks to preserve routing state [91] and bandwidth guarantees [43]. These systems' goals are distinct from this dissertation's: they aim to improve networks by reasoning about them as distributed systems, whereas we seek to improve distributed systems by using inside information from the network.

Fault-tolerant software-defined networking. Traditional network infrastructure was fault-tolerant because its routers and switches were physically distributed and its protocols were designed under extremely adverse assumptions [27]. Software defined networking (SDN) centralizes the network's routing logic into a *controller*. Some researchers have used techniques from distributed systems to replicate and distribute the SDN controller [31, 35, 36, 63, 105, 114]. This work would complement Albatross (Chapter 4), which itself leverages SDN. Onix [63] in particular would fit well with Albatross (pro-

vided Onix is configured to replicate a transactional and persistent state database across controllers). HyperFlow [105] also addresses distributed controllers; however, because HyperFlow controllers can establish rules locally (without synchronizing with logically centralized state), additional support is needed to integrate this work with Albatross. DevoFlow [31] relieves load on the controller by arranging for it to handle only "significant" flows; this is consistent with the design of Albatross, since we expect failures to be relatively rare and "significant" events. DIFANE [114] relieves controller load by distributing the handling of events to *authority switches*, but this is orthogonal to Albatross; the events that Albatross cares about are comparatively rare and can be handled by a single controller.

Chapter 3

Falcon: using inside information for reliable failure detection

> *Two hunters are out in the woods when one of them collapses. He doesn't seem to be breathing and his eyes are glazed. The other guy whips out his phone and calls the emergency services. He gasps, "My friend is dead! What can I do?" The operator says "Calm down. I can help. First, let's make sure he's dead." There is a silence, then a gun shot is heard. Back on the phone, the guy says "OK, now what?"*
>
> - LaughLab's "World's Funniest Joke" [14]

Inside information promises fast reports of failures, but leveraging such information in a failure reporting service requires addressing three challenges: (1) systematically collecting inside information, (2) presenting that information coherently to applications, and (3) limiting negative impact. In this chapter, we first focus our attention on the second challenge by restricting the interface of a failure reporting service to that of a *reliable failure detector*; this choice guides our design in addressing the remaining two challenges. A reliable failure detector reports processes as "up" or "down", with the guarantee that any process reported as "down" has actually crashed; this guarantee is inspired by Chandra and Toueg's perfect failure detector [21] (we purposefully avoid calling anything implemented in this dissertation "perfect").

Reliable failure detectors benefit applications by removing uncertainty about failures. Since an application can trust "down" reports, it can avoid split-brain scenarios

This chapter revises [70]: J. B. Leners, W.-L. Hung, H. Wu, M. K. Aguilera, and M. Walfish. *Detecting failures in distributed systems with the FALCON spy network*, In ACM SOSP, Oct. 2011. Co-authors Marcos K. Aguilera and Michael Walfish contributed to the presentation and design of Falcon. Wei-Lun Hung and Hao Wu contributed to the implementation and evaluation of Falcon.

without using majority-based techniques such as Paxos [67]. Instead, applications can employ simpler techniques, like primary-backup [4].

Building a reliable failure detector is not a new idea, and the standard approach is to purposefully crash processes when they are suspected of failure [39, 73]; this approach is sometimes referred to as "shoot the other node in the head", or STONITH for short. The problem with STONITH is that it can cause *collateral damage* (e.g., halting a machine to terminate a single suspected process).

In this chapter we present Falcon, a reliable failure detector that leverages inside information to detect failures quickly and that kills surgically to avoid collateral damage. Falcon uses a network of *spies*, which are layer-specific modules for determining which components of a system are working. Spies sometimes kill components to back up their decisions, but they can avoid collateral damage by surgically killing exactly the component suspected of failure.

A challenge that we address in building Falcon is a careful, thorough, and general design for spies to maximize coverage and limit unnecessary killing. Spies are deployed in a chained network, where the spy in one layer monitors the spy at the next layer up (e.g., the OS spy monitors the process spy). Thus, in the common case, if any layer in the system crashes, some spy will observe it. There are, however, two limiting cases in Falcon. First, Falcon cannot assume that spies will detect every failure, so Falcon includes a backstop: a large end-to-end timeout to cover (the ideally rare) cases that spies miss. Second, to report "down" reliably, Falcon must be able to communicate with some spy. The result is that if a network partition happens, Falcon hangs until the network heals. We revisit the second case in Albatross (Chapter 4) and Pigeon (Chapter 5), though we emphasize that these failure reporting services are *not* reliable failure detectors.

We have implemented and evaluated Falcon. Our implementation deploys spies on four layers: application, OS, hypervisor, and switch. We find that for a range of failures, Falcon has sub-second detection time, which is one or two orders of magnitude faster than the end-to-end timeouts used by existing systems. This yields higher-availability: adding Falcon to ZooKeeper [53] (which provides configuration management) and to a replication library [77] reduces unavailability following some crashes by

roughly 6 times. Falcon's overheads and per-platform requirements are small, and it can be integrated into an application with tens of lines of code. Finally, Falcon realizes the benefits of accurate failure reporting: applications can shed complex logic for handling inaccuracies from the failure reporting service (e.g., a replicated state machine can be implemented with primary-backup [4] instead of Paxos [67]), thereby using roughly half the code in our estimate.

3.1 Design principles of Falcon

The design principles underlying Falcon are as follows:

Give reliable reports. With a *reliable* failure detector, applications that use Falcon need not handle failure detector mistakes and the resulting complexity.

Peek inside the layers. Inside information can reveal crashes accurately and quickly. For example, a process absent from the OS's process table is certainly dead and a process lacking some key thread is as good as dead. Extracting this information requires layer-specific modules, which we call *spies*. Spies may sometimes use timeouts on internal events (e.g., an event loop has not progressed for one second), but these timeouts can be tailored to more predictable local behavior.

Kill surgically, if needed. A spy may not always observe failures correctly, but its reports must be reliable. Thus, it may kill when it suspects a crash (e.g., a local timeout has fired). Killing is disruptive and so should be limited to the smallest necessary component, rather than entire machines [39, 72, 73]. Such surgical killing conserves resources (e.g., a single processes is killed while others on the same machine are not) and improves recovery time (restarting a process is faster than restarting a machine). A similar argument was made by Candea et al. [16, 17] in the context of reboot.

Watch the watchers. Spies themselves may crash, either along with their layer, or independently. This calls for a *spy network*, in which lower-level spies monitor higher-level ones.

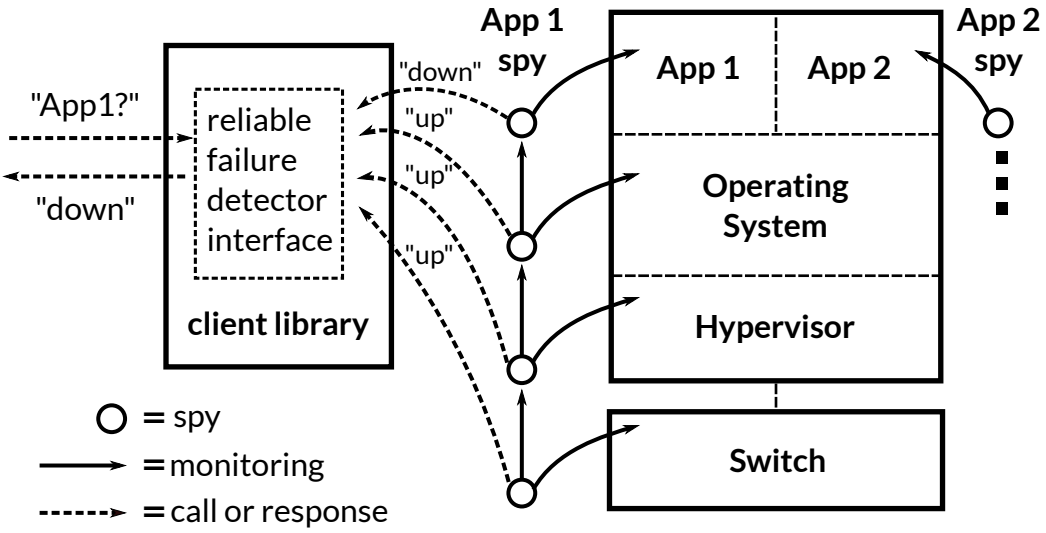

Figure 3.1: Architecture of Falcon. The application spy provides accurate information about whether the application is up; this spy is the only one that can observe that the application is working. The next spy down provides accurate information not only about its layer but also about whether the application spy is up; more generally, lower-level spies monitor higher-level ones.

Use end-to-end timeouts a last resort. As noted in Chapter 1, end-to-end timeouts have problems, but they also have the useful property of *completeness*: end-to-end timeouts eventually catch all failures. The completeness property makes end-to-end timeouts useful as a catch-all for detecting failures unforeseen in the failure reporting service's design, even if they are sub-optimal for quickly detecting common case failures.

3.2 Design of Falcon

Figure 3.1 depicts Falcon's architecture. Falcon consists of a *client library* as well as several *spy modules* (or *spies*) deployed at various layers of the system. The client library provides the reliable failure detector interface to the client, and it coordinates the spies. Roughly speaking, the client library takes as input the identifier of a *target process*, which specifies a process whose operational status the client would like to know, and returns

function	description
init(*target*)	register with spies
uninit()	deregister with spies
query()	query the operational status
setCallback(*callback*)	install callback function
clearCallback()	cancel callback function
startTimer(*timeout*)	start end-to-end timeout timer
stopTimer()	stop end-to-end timeout timer

Figure 3.2: Falcon's reliable failure detector interface to clients.

"up" or "down". A spy is a layer-specific monitor, and spies are named by the layer monitored (e.g., the OS spy monitors the OS) but may have parts running at several layers. The layers monitored by our current implementation are application, OS, hypervisor, and network. Falcon assumes that lower layers enclose higher ones, i.e., whenever a lower layer crashes, the layers above it also crash or stop responding. As an example, if the hypervisor crashes, then both the OS and application crash; as another example, if the network crashes, then the higher layers become unresponsive.

The difficulty in designing Falcon is using the knowledge and placement of spies to meet the desired properties. Our experience is that ad-hoc approaches lead to erroneous designs or fail to satisfactorily address the three challenges in Chapter 1. We present the design of Falcon by explaining, in turn, how we define the reliable failure detector interface, determine the interface to spies, specify exactly what spies do, orchestrate spies, and handle various corner cases. Section 3.3 describes the details of the spies in our implementation.

3.2.1 Reliable failure detector interface

The reliable failure detector interface that Falcon presents to clients is shown in Figure 3.2. Function init() indicates the target to be monitored, which identifies each layer (process name, VM id, hypervisor IP address, switch IP address). Function query() returns "up" or "down" for the target. However, a client may wish to monitor the target

continuously while waiting for a response or another event. Thus, rather than invoking query() repeatedly, it may be more efficient for the client to use a callback interface. To that end, function setCallback() installs a callback function to be called when the status of a target process changes from "up" to "down". Function clearCallback() uninstalls the callback function. To support end-to-end timeouts, Falcon needs to know when to start and stop the timeout timer, which the client indicates by calling functions startTimer() and stopTimer().

3.2.2 Objective and operation of spies

A given layer is supposed to perform some activity, and if the layer is performing it, then the layer is alive by definition. In a web server, activity may mean serving HTTP requests; for a map-reduce task, activity may mean reading and processing from disk; for a numerical application, activity may mean finishing a small stage of the computation; for a generic server, it may mean placing requests on an internal work queue and waiting for a response; for the OS, it may mean scheduling a ready-to-run process; and for a hypervisor, it may mean scheduling virtual machines and executing internal functions.

The purpose of a spy is to sense the presence or absence of such activity using this inside information. A spy exposes three remote procedures:

- REGISTER() to register a remote callback (which is distinct from the callback to the client in §3.2.1: the one here goes from a spy to the client library);
- CANCEL() to cancel it; and
- KILL() to kill the monitored layer.

If the layer that the spy is monitoring crashes, the spy immediately calls back the client library, reporting DOWN.

A spy is designed to recognize the common case when the monitored layer is clearly crashed or healthy. What if the spy is uncertain? To support reliable failure detection, a report of DOWN must be true, always. Thus, if the spy is inclined to report DOWN but is not sure, the spy resorts to killing: it terminates the layer that it is monitoring and *then* reports DOWN. (Section 3.3 explains how spies at each layer kill reliably;

```
remote-procedure REGISTER()
  add caller to Clients
  return ACK

remote-procedure CANCEL()
  remove caller from Clients
  return ACK

remote-procedure KILL()
  kill layer we are spying on and wait to confirm kill
  return ACK

background-task monitor()
  while true
    sense layer and set rc accordingly
    if rc = CERTAINLY_DOWN then
      callback(LAYER_DOWN)
    if rc = SUSPECT_CRASH then
      kill()
      callback(LAYER_DOWN)

function callback(status)
  for each client ∈ Clients do
    send status to client
```

Figure 3.3: Pseudocode for spies.

the basic idea is to use a component embedded in the layer below the layer to be killed.) Of course, spies should be designed to avoid killing.

Figure 3.3 gives pseudocode for our spies. Below, in Section 3.2.3, we describe how the client library coordinates the spies, assuming that (1) spies are ideal and (2) network partitions do not happen. Sections 3.2.4 and 3.2.5 relax these two assumptions in turn.

3.2.3 Orchestration: watching the watchmen

To report the operational status of the target, the client library uses the following algorithm. On initialization, it registers callbacks with each spy at the target and sets a local status variable to "up". If the client library receives a DOWN callback from any of the spies, it sets the status variable to "down". When the client library receives a query

tag	error / limiting case	cause	effect
A	layer L is down, but spy on layer L thinks layer L is up	bug in layer-L spy	triggers end-to-end timeout
B	layer L is down but spy on layer L is unresponsive	bug in layer-L spy	triggers end-to-end timeout
C	layer L is up, but spy on layer L or below reports DOWN	should not happen	would compromise guarantees
D	none of the spies responds	network partition	Falcon hangs or watchdog activates

Figure 3.4: Errors and limiting cases in Falcon, and their effects.

from the application, it returns the value of the status variable.

To see why this algorithm works, first note that if the target application is responsive then none of the spies returns DOWN—we are assuming ideal spies—and therefore the client library reports the status of the target correctly. If the target application crashes but the application spy remains alive, then the application spy returns DOWN and subsequently the client library reports the status of the target correctly. However, the application spy may never return, because it might have crashed. In that case, we rely on the spy at the next level—the OS spy—to sense this problem: in fact, the role of the layer-L spy can be seen as monitoring the layer-$(L + 1)$ spy, as shown in Figure 3.1. Here, the OS spy is monitoring the application spy, and if the application spy is crashed, the OS spy will eventually return DOWN—provided the OS spy itself is alive. If the OS spy is not alive, this procedure continues at the spy at the next level, and so on. The ultimate result is that if a spy never responds, a lower-level spy will sense the unresponsive spy and will report DOWN, causing the client library to report "down" to the client.

We have not yet said how the spy on layer $L+1$ is monitored by the spy on layer L. The spy on layer $L + 1$ has a component at layer L, for killing and for responding to queries. Given this component, the spy on layer L can monitor the spy on layer $L+1$ by *monitoring layer L itself*. This avoids the complexity of a signaling protocol among spies. It works because, assuming ideal spies, the spy on layer $L + 1$ is down (permanently unresponsive) if and only if layer L is down.

3.2.4 Coping with imperfect spies

The last section assumed ideal spies. In this section, we identify the types of mistakes that a spy can make and explain how Falcon deals with these mistakes. While Falcon may take drastic actions (killing or waiting for a long time), we expect them to be rare.

There are four types of spy errors that we consider, as shown in Figure 3.4. Error A takes place when a spy does not recognize a rare failure condition and thus wrongly thinks that a layer is up; for instance, an OS spy thinks that the OS is up because it shows some signs of life, yet the OS has stopped scheduling processes. Error B happens when there is a violation in the assumption from Section 3.2.3 that a layer L is up if and only if the spy on layer $L + 1$ is responsive. Error C occurs if a spy reports DOWN when either the monitored layer is up or any spy above the monitored layer is up. Error D occurs when none of the spies responds, because of a network problem such as a partition.

Errors A and B cause the *query* function to always return "up" despite the application's being down. To address this problem, Falcon has a backstop: an end-to-end timeout started by the client. If this end-to-end timeout expires, Falcon kills the highest layer that it can and subsequently reports the target as "down".

Error C is outside of the scope of Falcon because Falcon is expressly designed *not* to have this error: when a spy reports DOWN, it must absolutely ensure that the layer is down: forever disconnected from the outside world. Error D requires a more in-depth treatment, which we give in Section 3.2.5.

Figure 3.5 describes the client library's pseudocode. There are several points to note here. First, end-to-end timeouts are used to indicate a failure only in the unlikely case that none of the spies can determine that a layer is up or down. Second, each spy's *kill* procedure is invoked by the client library when the end-to-end timeout expires. This procedure attempts to kill the highest layer and, if not successful after SPY-RETRY-INTERVAL, targets each lower layer successively. In this manner, killing is surgical. A reasonable value for SPY-RETRY-INTERVAL is 3 seconds; this parameter affects detection time (by imposing a floor) but only when a large end-to-end timeout expires, an event that we expect to be rare.

```
function init(target)
  for L ← 1 to N do
    invoke REGISTER() at spy in target[L]
  Target ← target
  Status ← "up"
  Callback ← dummyFunction()

function uninit()
  for L ← 1 to N do
    invoke CANCEL() at spy in Target[L]

function query()
  return Status

function setCallback(callback)
  Callback ← callback

function clearCallback()
  Callback ← dummyFunction()

function startTimer(timeout)
  start countdown timer with value timeout

function stopTimer()
  stop countdown timer

upon receiving callback (status) from spy in Target[L] do
  if status = DOWN then
    Status ← "down"
    Callback("down" )

upon expiration of countdown timer do
  for L ← N downto 1 do
    invoke KILL() at spy in Target[L]
    if L ≠ 1 then wait for reply for SPY_RETRY_INTERVAL
    else wait for reply        // blocks on network partition; see §3.2.5.
    if got reply then
      Status ← "down"
      Callback("down" )
      return
```

Figure 3.5: Pseudocode for the client library. N is the number of monitored layers and the layer number of the application.

3.2.5 Network partition

We said above that lower-level spies monitor higher-level ones, but no spy monitors the lowest level spy. This is not a problem because that spy inspects the network switch attached to the target so it is conceptually a spy on the target's network connectivity. Thus, if the client library does not hear from that spy, then the network is slow or partitioned. (Our current implementation assumes that a machine is attached to one switch; we discuss the case of multiple switches in Section 3.5.)

There are three ways to handle network partition in Falcon. First, the client library can block until it hears from the switch; this is what our implementation does. Second, the client library can, after the client-supplied timeout expires call back with "I don't know"; this is an implementation convenience conceptually identical to blocking. Third, the client library can report "down" *after* it is sure that a watchdog timer on the switch has disconnected the target; meanwhile, in ordinary operation, the watchdog is serviced by heartbeats from the client library to the switch.

3.2.6 Application restart

If the application crashes or exits, and restarts, the client library should not report the application as "up" because clients typically want to know about the restart (e.g., the application may have lost part of its state in a crash). Therefore, when the application restarts, Falcon treats it as a different instance to be monitored, and the original crashed instance is reported "down".

To implement the above, the spy on a layer labels the layer with a generation number, and the spy includes this number in messages to the client library. Upon initialization, the client library records each layer's generation number. If it receives a mismatched generation number from a spy, then the associated layer has restarted and the client library considers the monitored instance as down. (Generation numbers are omitted from the pseudocode for brevity.)

Implementing generation numbers carries a subtlety: the generation number of a layer needs to increase if any layer below it restarts. Thus, a spy at layer L constructs its generation number as follows. It takes the entire generation number of layer $L-1$,

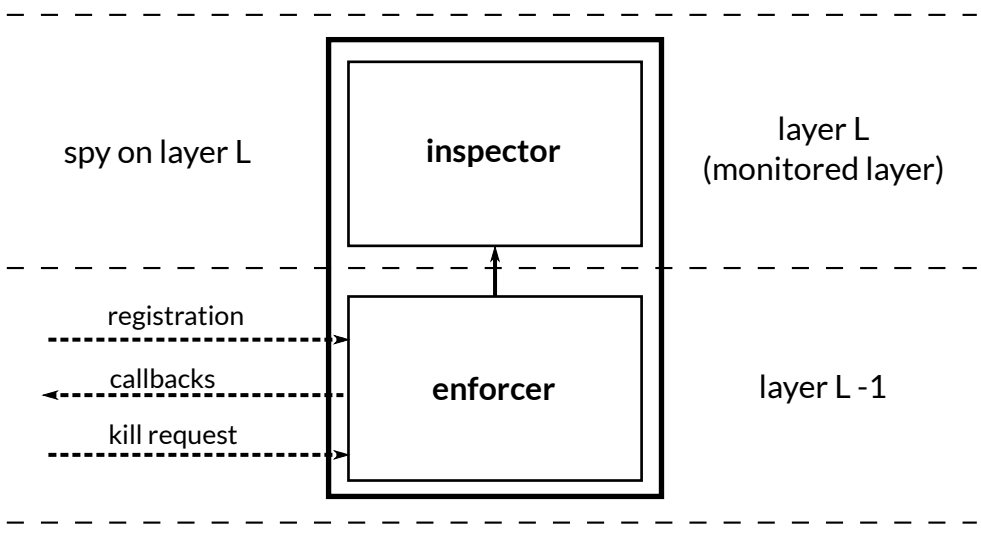

Figure 3.6: Architecture of spies. A spy has two components: an *inspector* that gathers inside information and an *enforcer* that ensures the reliability of DOWN reports (and may also use inside information). The client library communicates with the enforcer.

left shifts it 32 bits, and sets the low-order 32 bits to a counter that it increments on every restart. (The base case is the generation number of the lowest layer, which is just a counter.) At the application level, therefore, the generation number is a concatenation of 32-bit counters, one for each layer. 32 bits are sufficient because a problem occurs only if (a) the counter wraps around very quickly as crashes occur rapidly, and (b) the counter suddenly stops exactly where it was the last time that the client library checked.

3.3 Details of Falcon's spies

The previous section described Falcon's high-level design. This section gives details of four classes of spies that we have built: application spies, an OS spy, a hypervisor spy, and a network connectivity spy. We emphasize that these spies are illustrative reference designs, not the final word; one can extend spies based on design-time application knowledge or on failures observed in a given system. Nevertheless, the spies that we present should serve as an existence proof that it is possible to react to a large class of failures.

As shown in Figure 3.6, a spy has two components:

1. *Inspector*: This component is embedded in the monitored layer and gathers detailed inside information to infer the operational status, for example by inspecting the appropriate data structures.
2. *Enforcer*: This component communicates with the client library and is responsible for killing the monitored layer; for these reasons, it resides one layer below the monitored layer. This component may also use inside information.

A spy has only two technical requirements (§3.2.2): it must eventually detect crashes of the layer that it is monitoring (and even then, Falcon handles the case that the spy fails in this charge, per §3.2.4), and it must be reliable, meaning that its DOWN answers are accurate. However, in practice, a spy should be more ambitious: it should provide guarantees that are broader than the letter of its contract implies. To explain these guarantees and how they are achieved, we answer the questions below for each spy in our implementation, which is depicted in Figure 3.7.

- *What are the spy's components, and how do they communicate?* There is a lot of latitude here, but we discuss in Section 3.5 the possibility of a uniform intra-spy interface.
- *How does the spy detect crashes with sub-second detection time?* Detecting failures quickly is the high-level goal of Falcon (and this dissertation more generally), so Falcon's spies should detect failures quickly locally.
- *How does the spy avoid false suspicions of crashes and the resulting needless kills?* Avoiding false suspicion is not an explicit requirement of a spy, but it is far better if the resulting needless kills are kept to a minimum to limit the negative impact of Falcon.
- *How does the spy give a reliable answer?* Giving a reliable answer requires two abilities from spies. First, spies must be able to determine with certainty when their layer is down. Second, spies must be able to kill a layer when they are uncertain of its status (or requested to kill via KILL).
- *What are the implementation details of the spy?* Spies are unavoidably platform-specific, and we try to give a flavor of that specificity as we describe the implementation details. Section 3.5 discusses how Falcon might work with a different set of layers (e.g., with a JVM and nested VMs, or without VMs) and different instances of each

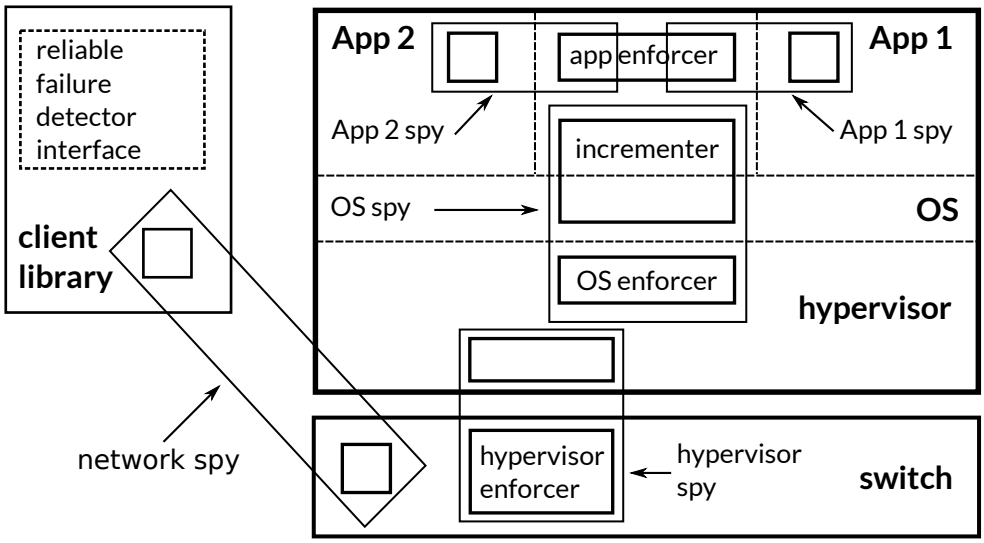

Figure 3.7: Our implementation of Falcon.

layer (e.g., Windows instead of Linux).

Application spies. All of our application spies have a common organization and approach.

Components. The inspector is a dedicated thread inside the application; it calls a function getStatus(), whose implementation depends on the application. For example, in our primary-backup application spy, getStatus() checks whether the main event loop is processing events; in our ZooKeeper [53] spy, getStatus() tests whether a client request has been recently processed, while a separate component submits no-op client requests at a low rate.

The enforcer is a distinguished high-priority process, the *app enforcer*, which serves as the enforcer for all monitored applications on the same OS. An assumption is that if the OS is up, then so is the app enforcer; this is an instance of the assumption, from Section 3.2.3, that "if layer-L is up, then so is the spy on layer-$(L+1)$". As discussed in Section 3.2.4, in the uncommon case that this assumption is violated, Falcon Falcon relies on an end-to-end timeout. The enforcer communicates with each inspector over

a connected inter-process communication (IPC) channel.

Sub-second detection time. If the inspector locally detects a problem, it closes its handle to the connected IPC channel, causing the enforcer to suspect a crash immediately (which it then handles per *Reliability*, below). Similarly, if the application process exits or crashes, then it brings the inspector down with it, again causing an immediate notification along IPC.

In addition, every $T_{app\text{-}check}$ time units, the enforcer queries the inspector thread, which invokes getStatus(). The enforcer infers a crash if getStatus() returns "down", if the IPC handle returns an error, or if the inspector thread does not respond within an application specific $T_{app\text{-}resp}$ time; the enforcer again handles these cases per *Reliability*, below. We note that getStatus() can use timing considerations apart from $T_{app\text{-}resp}$ and $T_{app\text{-}check}$ to return "down" (e.g., the inspector might know that if a given request is not removed from an internal queue within 10 ms, then the application is effectively down).

The periodic queries from enforcer to inspector achieve sub-second detection time in the usual cases because our implementation sets $T_{app\text{-}check}$ to 100 ms. While the precise choice is arbitrary, the order of magnitude (tens or hundreds of milliseconds) is not. Checking does not involve the network, and it is inexpensive—less than 0.02% CPU overhead per check in our experiments (see Figure 3.14, Section 3.4.4 and divide by 10 to scale per check). That is, we accept a minimal processing cost to get rapid detection time in the usual cases. The remaining case is covered by $T_{app\text{-}resp}$, which our implementation sets to 100 ms of CPU time, yielding sub-second detection time under light to medium load.

Avoiding false suspicions. The application spy avoids false suspicion in two ways. First, as mentioned above, the enforcer measures $T_{app\text{-}resp}$ by the CPU time consumed by the monitored application, not real time; this is an example of inside information and avoids the case that the enforcer declares an unresponsive application down when in fact the application is temporarily slow because of load. We note that this approach does not undermine any real-time deadlines since those are expressed and enforced by Falcon's end-to-end timeout (§3.2.4).

A second use of inside information is that $T_{app\text{-}resp}$ is set by the application itself. One choice is $T_{app\text{-}resp} = \infty$; in that case, if the app inspector is unresponsive, Falcon

relies on the end-to-end timeout. Or, an application might expect to be able to reply quickly, if it is scheduled and given CPU cycles, in which case it can set a smaller value of $T_{app\text{-}resp}$ for faster detection when the application process is unexpectedly stuck.

Reliability. If the enforcer suspects a crash, it inspects the process table. If the application process is not there, the enforcer no longer has doubt and reports DOWN to the client library. On the other hand, if the process is in the process table, then the enforcer kills it (by asking the OS to do so) and waits for confirmation (by polling the process table every 5 ms) before reporting DOWN. If the process does not leave the process table, then Falcon relies on the end-to-end timeout.

Implementation details. The inspector and app enforcer run on Linux and we assign app enforcer the maximum real-time priority. We also lock the processes address space in memory (to prevent its being swapped out). The inspector is implemented in a library; using the library requires only supplying getStatus() and a value of $T_{app\text{-}resp}$. The IPC channel between inspector and app enforcer is a Unix domain socket. The enforcer kills by sending a SIGKILL. We are assuming that process ids are not recycled during the (short) process table polling interval; if a process id *is* recycled, the end-to-end timeout applies.

OS spy. Our OS spy currently assumes virtualization; Section 3.5 discusses how Falcon could handle alternate layerings.

Components. The inspector consists of (a) a kernel module that, when invoked, increments a counter in the OS's address space and (b) a high-priority process, the *incrementer*, that invokes this kernel module every $T_{OS\text{-}inc}$ time units, set to 1 ms in our implementation. The enforcer is a module inside the hypervisor. The communication between the enforcer and the inspector is implicit: the enforcer infers that there was a crash if the counter is not incremented. Before detailing this process, we briefly consider an alternate OS spy: the enforcer could inspect a kernel counter like jiffies, instead of a process-incremented counter. We rejected this approach because an observation of increasing jiffies does not imply a functional OS. With our approach, in contrast, if the counter is increasing the enforcer knows that at least the high priority incrementer process is being scheduled. The cost of this higher-level assurance is an extra point of

failure: if the incrementer crashes (which is unlikely), then Falcon treats it as an OS crash. Specifically, the OS enforcer would detect the absence of increments, kill, and report DOWN.

Sub-second detection time. Every $T_{OS\text{-}check}$ time units, the enforcer checks the OS. To do so, it first checks whether the VM of the OS is running. If not, the enforcer reports DOWN to the client library. Otherwise, it checks whether the counter has incremented at least once over an interval of $T_{OS\text{-}resp}$ time units, and if the counter is the same, the enforcer suspects that the OS (or virtual machine) has crashed, which it handles per *Reliability* below. This approach achieves sub-second detection time by choosing $T_{OS\text{-}check}$ and $T_{OS\text{-}resp}$ to be tens or hundreds of milliseconds; our implementation sets them to 100 ms.

Avoiding false suspicions. Given the detection mechanism above, a false suspicion happens when the counter is not incremented, yet the VM is up. This case is most likely caused by temporary slowness of the VM, which in turn results from load on the whole machine. To ensure that the OS spy does not wrongly declare failure in such situations, we carefully choose $T_{OS\text{-}inc}$, $T_{OS\text{-}check}$, and $T_{OS\text{-}resp}$ to avoid premature local timeouts most of the time, even in extreme cases. This approach is inexact, as the VM could in theory slow down arbitrarily—say, due to a flood of hardware interrupts—triggering a premature local timeout. However, we do not expect this case to happen frequently; if it happens, the enforcer will kill the OS, but the spy will not return incorrect information.

We validate our choice of parameters by running a fork+exec bomb inside a guest OS, observing that in a 30 minute period (18,000 checks) the enforcer sees, per check, a mean of 97.8 increments, with a standard deviation of 3.9, and a minimum of 34 (where one increment satisfies the enforcer). Of course, the operators of a production deployment would have to validate the parameters more extensively, using an actual peak workload. We note that these kinds of local timing parameters have to be validated only once and are likely to be accurate; this is an example of inside information and does not have the disadvantages of end-to-end timeouts.

Reliability. If the VM is no longer being scheduled, the enforcer can verify that case, using its access to the hypervisor. If the enforcer suspects a crash, it asks the

hypervisor to stop scheduling the VM and waits for confirmation.

Implementation details. Like the app enforcer, the incrementer is a Linux process to which we assign the maximum real-time priority and which we also lock into memory. Our hypervisor is standard Linux; the VMs are QEMU/KVM [88] instances. The enforcer runs alongside these instances and communicates with them through the libvirtd daemon, which exposes the libvirt API, an interface to common virtualization functions [71]. We extend this API with a call to check the incrementer's activity. Since all calls into the libvirt API are blocking, we split the OS enforcer into two types of processes. A singleton main process communicates with the client library and forks a worker process, one per VM, sharing a pipe with the worker process. The workers use the libvirt API to examine the guests' virtual memory, kill guest VMs, and confirm kills.

Hypervisor spy. Our implementation assumes the ability to deploy new functionality on the switch. We believe this assumption to be reasonable in our target environment (data center networks; Chapter 1), particularly given the trend toward programmable switches. We also assume that the target is connected to the network through a single interface; Section 3.5 discusses how this assumption could be relaxed.

Components. The inspector is a module in the hypervisor, while the enforcer is a software module that runs on the switch to which the hypervisor host is attached. The enforcer infers that the hypervisor is crashed if (a) the switch has not seen any traffic from the hypervisor for a period of time and (b) the enforcer cannot solicit traffic by pinging the inspector (this detection method saves network bandwidth, versus more active pinging). The two communicate by RPC over UDP.

Sub-second detection time. Every $T_{hypervisor\text{-}check}$ time units, the enforcer checks that the hypervisor is alive. This check takes one of two forms. Usually, the enforcer checks whether the switch has received network packets from the hypervisor over the prior interval. If this check fails or if an interval of $T_{hypervisor\text{-}check\text{-}2}$ time units (set to 5 seconds in our implementation) has passed since the last probe, the enforcer probes the inspector with an RPC. If it does not get a response within $T_{hypervisor\text{-}resp}$ time units (set to 20 ms in our implementation), it does $N_{hypervisor\text{-}retry}$ more tries (set to 5 in our implementation),

for a total waiting period of $T_{hypervisor\text{-}resp} \cdot (N_{hypervisor\text{-}retry} + 1)$ time units (120 ms in our implementation). After this period, the enforcer suspects a crash and handles that case per *Reliability*, below. Similar to the other spies, this one achieves sub-second detection time by choice of $T_{hypervisor\text{-}check}$: 100 ms in our implementation.

Avoiding false suspicions. First, our enforcer test is conservative: most of the time, any traffic from the hypervisor host placates the enforcer. Second, we validate our choice of parameters by running an experiment where 2000 processes on the hypervisor contend for CPU. We set the enforcer to query the inspector 100,000 times, observing a mean response time of 397 μs, with standard deviation of 80 μs, and a maximum of 12.6 ms, which suffices to satisfy the enforcer. As with the OS spy, operators would need to do more extensive parameter validation for production. Finally, although $N_{hypervisor\text{-}retry}$ is a constant in our implementation, a better implementation would set $N_{hypervisor\text{-}retry}$ in proportion to the traffic into the hypervisor. Then the test would permit more retransmissions under higher load, accommodating a message's lower likelihood of getting through.

Reliability. If it suspects a crash, the enforcer "kills" the hypervisor, by shutting down the network port to which the hypervisor is connected. The enforcer then reports DOWN to the client library. Here, Falcon assumes that every process running on every VM running on that end-host is shut down before that end-host is allowed to reconnect.

Implementation details. The hypervisor inspector runs as a process on the hypervisor (which is standard Linux, as described above). The hypervisor enforcer is a daemon process that we run on the DD-WRT open router platform [32], which we modified to map connected hosts to physical ports and to run our software.

Network spy. The inspector is a software module that runs on the network switch connected to the target, and the enforcer is a module in the client library. However, under our current configuration and implementation of Falcon, the network spy does not check for failures and does not affect Falcon's end-to-end behavior or our experimental results. The reason is that Falcon's knowledge of the network is limited to the switch attached to the target, so Falcon has no way to (a) know whether the switch is crashed or just slow, and (b) kill the switch if it is in doubt. The consequence is that Falcon blocks when the switch is unresponsive. We revisit this choice in Chapters 4 and 5.

3.4 Evaluation of Falcon

We evaluate our implementation of Falcon by considering challenges of building a fast failure reporting service set forth in Chapter 1—systematically collecting inside information, coherently presenting that information to applications, and limiting the negative impact of deployment—and asking to what degree Falcon meets those challenges. We also evaluate higher-level benefits for the applications that are clients of Falcon. To do so, we experiment with Falcon, other failure detectors [11, 22, 51], ZooKeeper, ZooKeeper modified to use Falcon, a minimal Paxos-based replication library [77], that library modified to use Falcon, and a primary-backup-based replication library that uses Falcon. Figure 3.8 summarizes our evaluation results.

Most of our experiments involve two panels. The first is a *failure panel* with 12 kinds of model failures that we inject to evaluate Falcon's ability to detect them (the kernel failures are from a kernel test module [74]). The second is a *transient condition panel* with seven kinds of imposed load conditions, which are *not* failures, to evaluate Falcon's ability to avoid false suspicions. The failure panel is listed in Figure 3.9, and the transient condition panel is detailed in Section 3.4.3. Since the panels are synthetic, our evaluation should be viewed as an initial validation of Falcon, one within the means of academic research. An extended validation requires deploying Falcon in production environments and exposing it to failures "in the wild."

Our testbed comprises three hosts connected to a switch. The switch is an ASUS RT-N16. The software on the switch is the DD-WRT v24-sp [32] platform (essentially Linux), extended with our hypervisor enforcer (§3.3). Our hosts are Dell PowerEdge T310, each with a quad-core Intel Xeon 2.4 GHz processor, 4 GB of RAM, and two Gigabit Ethernet ports. Each host runs an OS natively that serves as a hypervisor. The native (host) OS is 64-bit Linux (2.6.36-gentoo-r5), compiled with the kvm module [88], running QEMU (v0.13.0) and a modified libvirt [71] (v0.8.6). The virtual machines (guests) run 32-bit Linux (2.6.34-gentoo-r6), extended with a kernel module and accompanying kernel patch (for the OS inspector).

high-level question	evaluation result	section
What is the effect of Falcon's spy network?	• Even simple spies are powerful enough to detect a range of common failures.	§3.4.1
	• For these failure modes, Falcon's 99th percentile detection time is several hundred ms; existing failure detectors take one or two orders of magnitude longer.	§3.4.1
	• Augmenting ZooKeeper [53] and a replication library (PMP) [77] with Falcon (minus killing) reduces unavailability by roughly 6 times (or more, for PMP) for crashes below application level.	§3.4.2
What is the effect of the reliable failure detector interface?	• As a reliable failure detector, Falcon enables primary-backup replication [4], which requires fewer processes than Paxos [67] for the same fault-tolerance, and which requires less complexity (48% less code in our comparison).	§3.4.5
How does Falcon limit negative impact?	• For a range of failures, Falcon kills the smallest problematic component that it can.	§3.4.3
	• Falcon avoids false suspicions (and kills) even when the target is unresponsive end-to-end.	§3.4.3
	• Falcon's CPU costs at each layer are single digits (or less) of percentage overhead.	§3.4.4
	• Falcon requires per-platform code: about 2300 lines in our implementation. However, the added code is likely simpler than the application logic that can be removed by using an reliable failure detector.	§3.4.5
	• Falcon can be introduced into an application with tens or hundreds of lines of code.	§3.4.2, §3.4.5

Figure 3.8: Summary of main evaluation results.

where injected?	what is the failure?	what does the failure model?
application	forced crash	application memory error, assert failure, or condition that causes exit
application	app inspector reports DOWN	inside information indicates an application crash
application/ Falcon itself	unresponsive app inspector	since the app inspector is a thread inside the application, this models a buggy application (or app inspector) that cannot run but has not exited
kernel	infinite loop	kernel hang or liveness problem
kernel	stack overflow	runaway kernel code
kernel	kernel panic	unexpected condition that causes assert failure in kernel
hypervisor/host	hypervisor error; causes guest termination	hypervisor memory error, assert failure, or condition that causes guest exit
hypervisor/host	disable network card on host	hardware crash that separates hypervisor from network
Falcon itself	crash of app enforcer	bug in Falcon app spy
Falcon itself	crash of incrementer	bug in Falcon OS spy
Falcon itself	crash of OS enforcer	bug in Falcon OS spy
Falcon itself	crash of hypervisor inspector	bug in Falcon hypervisor spy

Figure 3.9: Panel of synthetic failures in our evaluation. The failures are at multiple layers of the stack and model various error conditions.

3.4.1 How fast is Falcon?

Method. We compare Falcon to a set of *baseline* failure detectors (FDs), focusing on detection times under the failure panel.

Figure 3.10 describes the baselines. These FDs are used in production or deployed systems (the ϕ-accrual FD is used by the Cassandra key-value store [19], static timers are used in many systems, etc.); we borrow the code to implement them from a Google Summer of Code project [119]. All of these FDs work as follows: the client pings the target according to a fixed *ping interval* parameter p, and if the client has not heard a response by a *deadline*, the client declares a failure. We define the *timeout* T to be the duration from when the last ping was received until the deadline for the following ping. The difference in these FDs is in the algorithm that adjusts the timeout or

baseline FD	T: timeout (ms)	error	parameters
Static Timer	10,000	0.0	timer = 10,000
Chen [22]	5,001	0.0	α = 1 ms
Bertier [11]	5,020	0.0	$\beta = 1, \phi = 4, \gamma = 0.1$, mod_step = 0
ϕ-accrual [51]	4,946	0.01	ϕ = 0.4297
ϕ-accrual	4,995	0.001	ϕ = 0.4339

Figure 3.10: Baseline failure detectors that we compare to Falcon. The implementations are from [119]. We set their ping intervals as $p = 5$ seconds, which is aggressive and favors the baseline FDs. For all but Static Timer, the timeout value T is a function of network characteristics and various parameters, which we set to make the error, e, small (e is the fraction of ping intervals for which the FD declares a premature timeout). We set ϕ-accrual for different e; in our experiments with no network delay, Chen and Bertier have no observable error.

deadline (based on empirical round-trip delay and/or on configured error tolerance).

We configure the baselines with $p = 5$ seconds, which is pessimistic for Falcon, as this setting allows the baselines to detect failures more quickly than they would in data center applications, where ping intervals are tens of seconds [15, 42, 55]. Likewise, we configure the ϕ-accrual failure detector to allow many more premature timeouts (one out of every 100 and 1000 ping intervals) than would be standard in a real deployment, which also decreases its timeout and hence its detection time.

We configure Falcon with an end-to-end timeout of 5 minutes; Falcon can afford this large backstop because it detects common failures much faster. For a like-to-like comparison between the baselines (which are unreliable) and Falcon (which is reliable), we also experiment with an unreliable version of Falcon called *Falcon-NoKill*, which is identical to Falcon except that it does not kill.

Each experiment holds constant the FD and the failure from the panel, and has 200 iterations. In each iteration, we choose the failure time uniformly at random inside an FD's periodic monitoring interval of duration p (for the baselines, p is the ping interval and for Falcon it is 100 ms, per §3.3). To produce a failure, a failure generator running at the FD client sends an RPC to one of the *failure servers* that we deploy at different layers on the target.

For convenience, our experiments measure detection time at the FD client, as

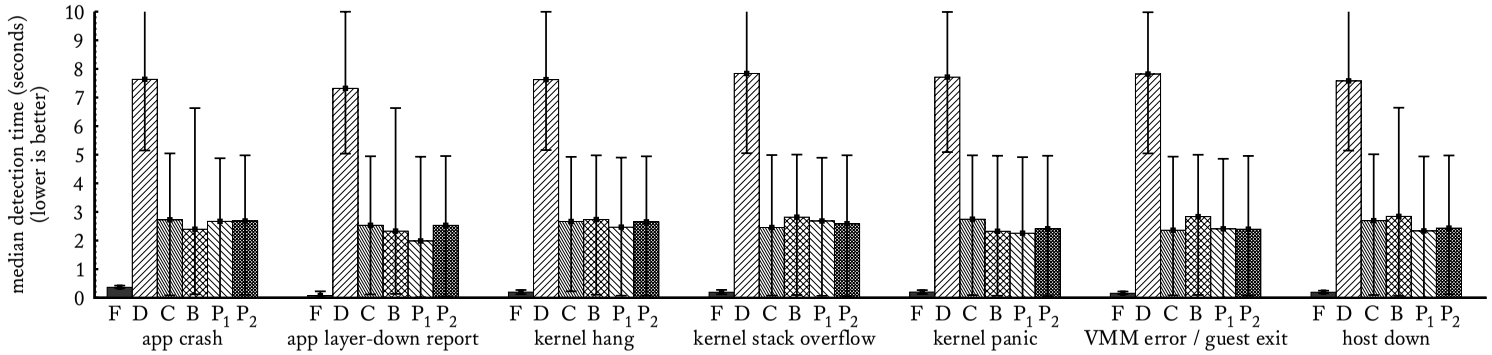

Figure 3.11: Detection time of Falcon (F) and baseline failure detectors under various failures. The baselines are Static Timer (D), Chen (C), Bertier (B), ϕ-accrual with 0.01 error (P_1), and ϕ-accrual with 0.001 error (P_2); see Figure 3.10 for details. Rectangle heights depict medians, and the bars depict 1st and 99th percentiles. The baseline FDs wait for multiple-second timers to fire. In contrast, Falcon has sub-second detection time, owing to inside information and callbacks. Moreover, the comparison is pessimistic for Falcon: with ping intervals that would mirror a real deployment, the baselines' bars would be higher while Falcon's would not change.

the elapsed time from when the client sends the RPC to the failure server to when the FD declares the failure. This approach adds one-way network delay to the measurement. However, we verified through separate experiments with synchronized clocks that the added delay is 2–3 orders of magnitude smaller than the detection times.

Experiments and results. We measure the detection times of the baseline FDs and of Falcon-NoKill, for a range of failures. Under constant network delay, we expect the baseline FDs' detection times to be uniformly distributed over $[T-p+d, T+d]$;[1] here, T and p are the timeout and ping interval, as defined above and quantified in Figure 3.10, and d is the one-way network delay. We hypothesize that Falcon's detection times will be on the order of 100 ms, given spies' periodic checks (§3.3).

Figure 3.11 depicts the 1st, 50th, and 99th percentile detection times, under no network delay (d = 0). The baselines behave as expected. For application crashes, Falcon's median detection time is larger than we had expected: 369 ms. The cause is the time taken by the Java Virtual Machine (JVM) to shut down, which we verified to be several hundred milliseconds on average. For the failure in which the app inspector reports DOWN, Falcon's median detection time is 75.5 ms. This is in line with expectations: the app enforcer polls the app inspector every $T_{app\text{-}check}$ = 100 ms, so we expect an average detection time of 50 ms plus processing delays.

For the kernel hang, kernel overflow, and kernel panic failures, Falcon's median detection times are 204 ms, 197 ms, and 207 ms, respectively. The expected value here is 150 ms plus processing delays: every $T_{OS\text{-}check}$ = 100 ms, the OS enforcer checks whether the prior interval saw OS activity (§3.3), so the OS enforcer in expectation has to wait at least 50 ms (the duration from the failure until the end of the prior interval) plus 100 ms (the time until the OS enforcer sees no activity). The processing delays in our unoptimized implementation are higher than we would like: 15 ms per check, for a total of 30 ms per failure, plus tens of milliseconds from supporting libraries and the client.

[1]The largest detection time occurs when the target fails just after replying to a ping; the client receives the ping reply after d time and declares the failure at the next deadline after T time, for a detection time of $T + d$. The smallest detection time occurs when the target fails just before replying to a ping; after d time (when the ping reply would have arrived), the client waits for $T - p$ time longer, then declares the failure, for a detection time of $T - p + d$.

Nevertheless, these delays, plus the expected value of 150 ms, explain the observations.

For the guest exit and host crash failures, Falcon's median detection times are 160 ms and 197 ms, respectively. For the guest exit, the observed detection time matches an expected 50 ms (since $T_{OS\text{-}check}$ = 100 ms) plus cleanup by the hypervisor of 90 ms plus processing delays of tens of milliseconds. Likewise, for the host crash, the observed detection time matches an expected 50 ms (since $T_{hypervisor\text{-}check}$ = 100 ms) plus the 120 ms of waiting (see §3.3), plus processing delays.

Falcon's detection time is an order of magnitude faster than that of the baseline FDs, for two reasons. First, inside information reveals the crash soon after it happens; second, the spies call back the client library when they detect a crash. With larger ping intervals p (which would be more realistic), the baselines' detection times would be even worse.

Our depicted measurements, here and ahead, are under no network delay (roughly modeling an uncongested network in a data center). However, we ran some of our experiments under injected delays ($d > 0$) and found, as expected, that Falcon's detection time increased by d. We did not experiment with the baselines under network delay; our prediction of their detection times (distributed over $[T - p + d, T + d]$) is stated above. We did not experiment under non-constant delay; based on their algorithms, we predict that the baselines, except for Static Timer, would react to network variation by increasing their timeout T. Falcon, meanwhile, would continue to detect crashes quickly, improving its relative performance.

3.4.2 What is Falcon's effect on availability?

We now consider the effect of improved detection time on system availability. We incorporate Falcon into two applications that use failure detectors based on static timers and majority-based techniques to handle FD errors: ZooKeeper [53] and a replication library [77] (PMP). The modifications are straightforward: roughly 150 lines of Java and 100 lines of C, respectively. We compare unavailability of these systems and their unmodified versions, in the case of a leader crash.

To apply Falcon, we use the spy for ZooKeeper, as described in Section 3.3 ("Application spies"), and a PMP spy that checks whether the main event loop is running;

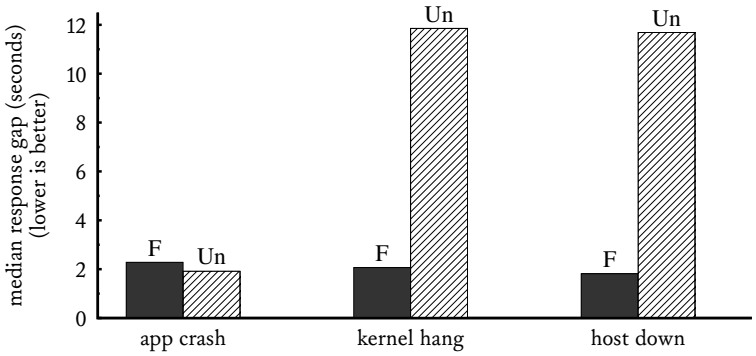

Figure 3.12: Median response gap (unavailability) of ZooKeeper [53] with Falcon-NoKill (F) and unmodified (Un) under injected failures at the leader. In unmodified ZooKeeper, followers quickly detect application crashes but not kernel- or host/VMM-level crashes. Under the latter types, Falcon reduces median ZooKeeper unavailability by roughly a factor of 6. In all cases, unavailability is several seconds on top of detection time because of ZooKeeper's recovery time.

in both cases, we use Falcon-NoKill, as both systems' unmodified failure detectors are unreliable. The unmodified ZooKeeper detects a crashed leader either via a ten-second timeout or if the leader's host closes the transport session with the followers. The unmodified PMP runs with its default of a ten-second timeout.

We configure ZooKeeper to use 4 nodes: 3 servers and 1 client (our testbed has 3 hosts, so the client and a server run on the same hypervisor). ZooKeeper partitions the servers into 1 leader and 2 followers. The ZooKeeper client sends requests to one of the followers (alternating "create" and "delete" requests) when it gets a response to its last one, recording the time of every response. For each of three failure types and the two ZooKeepers, we perform 10 runs. In each run, we inject a failure into the leader at a time selected uniformly at random between 3 and 4 seconds after the run begins. The result is a gap in the response times. Example runs look like this:

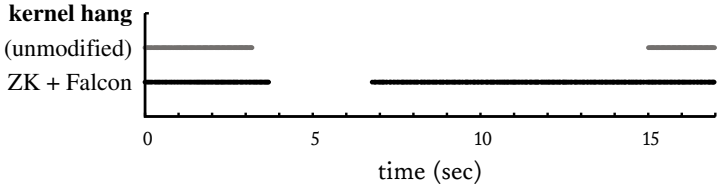

Figure 3.12 depicts the durations of those response gaps. Under application failures, ZooKeeper reacts relatively quickly because the follower explicitly loses its transport session with the leader. Though the median of ZooKeeper+Falcon is 350 ms slower than with unmodified ZooKeeper, this difference appears due to experimental variation (ZooKeeper+Falcon also experiences transport session loss, and the standard deviations are 566 ms for ZooKeeper+Falcon and 762 ms for unmodified ZooKeeper). Under kernel and hypervisor/host failures, the ZooKeeper follower receives no word that the system is leaderless, so it infers failure—and initiates leader election—only after not having heard from the leader for 10 seconds. Under all failures, Falcon's detection time is sub-second. However, unavailability is detection time plus recovery time, and in all of the depicted cases, recovery takes roughly 2 seconds: the ZooKeeper follower, in connecting to the new leader, usually requires two attempts separated by one second, and the client also has a retry discipline that imposes delays of one second or more.

We run analogous experiments for PMP, and the results are similar: tens of seconds of unavailability without Falcon and less than one second with Falcon.

3.4.3 What is the impact of killing in Falcon?

We consider how Falcon limits its negative impact, beginning with the impact of killing, which has two aspects: (1) *If* Falcon must kill, it should kill the smallest possible component, and (2) Falcon should not kill if not required (e.g., if the target is momentarily slow); that is, Falcon should avoid false suspicions. To evaluate these aspects, we run Falcon against our two panels, failures and transient conditions, reporting the component killed, if any. Figure 3.13 tabulates the results.

For aspect (1), Falcon's reactions to the injected failures match our expectations. If the failure is in the target, Falcon detects it and, if needed, kills the smallest component of the target. If, however, the failure is in Falcon itself (the last four injected failures), then there are two cases: either Falcon falls back on the end-to-end timeout, killing the layer at which the spy failure occurred, or else Falcon interprets the spy's failure as a layer failure and kills the layer quickly (e.g., as mentioned in Section 3.3, Falcon treats an incrementer crash as an OS crash). Falcon's surgical approach to reliability should be contrasted with STONITH, which kills the entire machine (though some

failure	action taken by Falcon
app crash	app enforcer detects failure
app layer-down report	app enforcer kills application
app inspector hangs	app enforcer kills application
kernel hang	OS enforcer kills guest OS
kernel stack overflow	OS enforcer kills guest OS
kernel panic	OS enforcer kills guest OS
hypervisor error / guest exit	OS enforcer detects failure
host down	hypervisor enforcer kills hypervisor/host
crashed app enforcer + app crash	end-to-end timeout kills guest OS
crashed incrementer	OS enforcer kills guest OS
crashed OS enforcer + OS crash	end-to-end timeout kills hypervisor/host
crashed hypervisor inspector	hypervisor enforcer kills hypervisor/host
transient condition	**action taken by Falcon**
hung system call	none
CPU contention within guest	none
CPU contention across guests	none
memory contention within guest	none
memory contention across guests	OS enforcer kills guest OS
packet flood between guests	none
packet flood between hypervisor	hypervisor enforcer kills hypervisor/host

Figure 3.13: Falcon's actions under the failure panel and transient condition panel. (Falcon-specific failures are augmented with target failures because otherwise the Falcon failure has no effect.) Under the failures, Falcon kills surgically. Under the transient conditions, Falcon correctly holds its fire in most cases but sometimes suspects falsely and thus kills.

implementations can target virtual machines [72]).

To show that Falcon avoids spurious killing, we apply the panel of transient conditions, listed in the bottom part of Figure 3.13. We expected Falcon to hold its fire in all of these cases, but there are two for which it does not. First, when guests contend for memory, the hypervisor (Linux) swaps QEMU processes that contain guests, to the point where there are intervals of duration $T_{OS\text{-}check}$ when some guests—and their embedded incrementers—do not run, causing the OS enforcer to kill. An improved

OS enforcer would incorporate further inside information, not penalizing a guest in cases when the guest is ready to run but starved for cycles. Second, when the network is heavily loaded, the communication channel between hypervisor enforcer and hypervisor inspector degrades, causing the hypervisor enforcer sometimes (in 4 out of 15 of our runs) to infer death and kill. As mentioned in Section 3.3, a better design would set $N_{hypervisor\text{-}retry}$ adaptively. In the other transient conditions, Falcon's inside information prevents it from killing. For example, the app enforcer measures $T_{app\text{-}resp}$ based on CPU time (§3.3), so a long block (e.g., the "hung system call" row) does not cause a kill.

3.4.4 What are the computational costs of deploying Falcon?

Falcon's benefits derive from gathering inside information with spies. Such platform-specific logic incurs computational costs and programmer effort. We address the former in this section and the latter in the next one.

Falcon's main computational cost is CPU time to execute periodic local checks (described in Section 3.3). To assess this overhead we run a Falcon-enabled target with an idle dummy application for 15 minutes, inducing no failures. We then run the same target and application but with the Falcon components disabled (and with QEMU and libvirtd enabled). In both cases, we measure the accumulated CPU time over the run, reporting the CPU overhead of Falcon as the difference between the accumulated CPU times divided by the run length.

Figure 3.14 tabulates the results. For the most part, Falcon's CPU overhead is small (less than 1% per component). The exception is the QEMU process in the hypervisor layer. Two factors contribute to this overhead. First, the Falcon-enabled virtual machine is scheduled more frequently than the Falcon-disabled virtual machine (because of Falcon's multiple checks per second in the former case versus an idle application in the latter case). To control for this effect, we perform the same experiment above, except that we run another application, alongside the dummy, that uses 90% of the CPU. Under these conditions, as depicted in Figure 3.14, QEMU contributes only 1.8% overhead in the Falcon-enabled case. Second, the remaining overhead is from QEMU's reading guest virtual memory inefficiently (when requested by the OS enforcer; see §3.3). We verified this by separately running the experiment above (Falcon enabled,

component (§3.3)	CPU overhead (percent of a core's cycles): app uses no CPU	app uses 90% CPU
app inspector	0.06	0.04
app enforcer	0.11	0.07
incrementer	0.58	0.31
VM total	**0.75%**	**0.42%**
OS enforcer (main)	0.01	0.01
OS enforcer (worker)	0.04	0.03
libvirtd	0.91	0.95
QEMU	6.92	1.79
hypervisor inspector	0.39	0.27
hypervisor total	**8.27%**	**3.07%**
hypervisor enforcer	0.00	0.00
switch total	**0.00%**	**0.00%**

Figure 3.14: Background CPU overhead of our Falcon implementation, under an idle dummy application and under one that consumes 90% of its CPU. Each enforcer performs a local check 10 times per second. The switch's CPU overhead is less than one part in 10,000 so displays as 0. QEMU's contribution to the overhead is explained in the text.

90% CPU usage by the dummy application) except that memory reads by the OS enforcer were disabled. The difference in QEMU's CPU usage was 1.4%, explaining nearly all of the CPU usage difference between the Falcon-enabled and Falcon-disabled cases.

To mitigate the overhead of QEMU's guest memory reads, we could increase $T_{OS\text{-}check}$ (which would reduce the number of checks but increase detection time) or improve the currently unoptimized implementation of guest memory reads.

3.4.5 What is the code and complexity trade-off?

Although we can use Falcon in legacy software (as in §3.4.2, where the gain was availability), Falcon provides an additional benefit to the applications that use it: shedding complexity. However, this is not "moving code around": the platform-specific logic required by Falcon has a simple function (detect a crashed layer and kill it if necessary)

module (§3.3)	spy component (§3.3)	lines of code
platform-independent modules		
thread in app; glue (C++)	app inspector	101
thread in app; glue (Java)	app inspector	241
shared enforcer code	all enforcers	465
client library	client library	1287
client library glue (Java)	client library	310
platform-independent total		**2404**
platform-specfic modules		
app enforcer process	app enforcer	403
incrementer	OS inspector	43
kernel module	OS inspector	39
libvirt extensions	OS enforcer	606
OS enforcer (main)	OS enforcer	509
OS enforcer (worker)	OS enforcer	83
libvirtd extensions	OS enforcer	53
RPC module	hypervisor inspector	103
DD-WRT extension	hypervisor enforcer	450
platform-specific total		**2289**
application-specific modules		
getStatus() for Paxos (from [77])	app inspector	17
getStatus() for primary-backup	app inspector	42
getStatus() for ZooKeeper [53]	app inspector	159

Figure 3.15: The modules in our Falcon implementation and their lines of code. The platform-independent modules assume a POSIX system.

while the logic shed in applications is complex (tolerate mistakes in an unreliable failure detector).

Figure 3.15 tabulates the lines of code in our implementation, according to an existing tool [111]. (We do not count external libraries in our implementation: sfslite for RPC functions, yajl for JSON functions, and libbridge for functions on the switch.) The platform-specific total is fewer than 2300 lines. The application-specific code is much smaller, for our sample implementations of getStatus() (though a production application might wish to embed more intelligence in its getStatus()).

Next, we assess the gain to applications that use failure detectors (FDs). Exam-

replication approach	lines of code	# processes
Paxos (from [77])	1759	3
Primary-backup	932	2

Figure 3.16: Comparison of two different approaches to replicating state machines: Paxos [67], as implemented in [77], and primary-backup [4], as implemented by us. The Paxos row excludes FD code and generated RPCs. The primary-backup approach is fewer lines of code because it is simpler: it does not tolerate unreliable failure detection. Primary-backup also has 50% lower replication overhead in the usual case.

ples of such applications are ZooKeeper, Chubby, state machine replication libraries, and systems that use end-to-end timeouts based on pings of remote hosts.[2] As noted at the start of this chapter, unreliable failure detectors necessitate complex algorithms to handle failure detector mistakes; for example, it might use Paxos [67] for replication. However, if the application has access to a reliable failure detector (as provided by Falcon), then it can use simpler approaches; for example it can use primary-backup [4] for replication. Measuring simplicity is difficult, but we compare the lines of code in (1) PMP—which uses a static timer for failure detection and Paxos for replication (see §3.4.2)—and (2) a replication library that we implemented, which uses Falcon for failure detection and primary-backup for replication. To make the comparison like-to-like, we exclude PMP's failure detection code from the count.

Figure 3.16 lists the numbers, again according to the tool used above [111]. The difference is 827 lines, which is 48% of the original code base. Additionally, primary-backup has lower replication overhead than Paxos: to tolerate a crash, Paxos requires three processes while primary-backup requires just two.

Assessing Falcon's reliability. The simplification results only if Falcon is truly reliable, meaning that it reports "down" only if the target is down. Falcon's spies are carefully designed and implemented *not* to violate this property, and in our experience, Falcon has never reported an up target as "down". However, we cannot fully guarantee reliability without formally verifying our implementation.

[2] A non-example is an application that uses ZooKeeper, Chubby, or another higher-level service that itself incorporates FDs. In these cases, the simplicity benefit of Falcon accrues to the higher-level service, not its user.

3.5 Summary & discussion

This section summarizes Falcon and discusses the limitations of our implementation.

Summary: revisiting the three challenges. Chapter 1 describes three challenges in building a fast failure reporting service; we revisit these in the context of Falcon.

Systematically collecting inside information. Falcon collects inside information with a network of layer-specific modules called spies. Spies are bicameral, with an inspector embedded in the monitored layer and an enforcer in the layer below. In occupying two layers, Falcon's spies hierarchically monitor one another implicitly, without requiring communication among spies. The key observation motivating this design choice is that encapsulation of layers causes higher-level layers to share fate with lower-level ones (e.g., a crashed virtual machine takes its processes down with itself). This design allows enforcers to expose a common interface, which simplifies the collection of inside information as done by Falcon's client library.

Defining an interface for reporting failures. Falcon exposes a reliable failure detector interface, which reports processes as "up" or "down" and guarantees that processes reported as "down" are permanently crashed. Applications benefit from this interface because they need not doubt the failure detector, and can thus eschew complex majority-based techniques (e.g., Paxos [67]) in favor of simpler algorithms (e.g., Primary-Backup [4] or Chain replication [107]). However, providing this interface has a price: Falcon's spies sometimes kill their monitored layer in order to make progress. Killing for certainty is not new (STONITH in high-availability clusters [73]), even with "virtual" killing (ISIS [93] excludes processes suspected of failure), though achieving certainty by killing targeted layers is new.

Limiting negative impact. Falcon's design limits its negative impact in two ways. First, killing layers can be disruptive, so Falcon kills surgically. Unlike prior implementations of STONITH, Falcon does not always kill at the granularity of machines (real or virtual): Falcon only kills suspected components, and sometimes does not kill. Second, Falcon's design limits the cost of deploying spies: instead of a allowing processes to query spies directly (potentially causing thousands of checks per second), spies operate indepen-

dently (with only tens of checks per second) and execute a callback in the rare case that there is a problem.

Limitations of our implementation. We now discuss three limitations of our current implementation. First, spies would be more coherent and modular with a uniform inspector-enforcer communication protocol. Currently, extending Falcon to support new layers (e.g., in a system without virtualization, or by adding, say, a JVM spy) requires both modifying current spies and implementing new ones. A uniform protocol would eliminate the need to modify existing spies, and promote code reuse.

Second, Falcon is tied to a strict layering scheme, where each layer encloses the next. Such layering may not be the case, for example, in networks where there may be many paths between end-hosts. We designed an extension to Falcon that handles the case that an end-host is connected to multiple switches, and even extended this design to the full network. However, crashing network switches would be prohibitively expensive, especially if there was any chance a spy monitoring a network switch might make a mistake, so we designed Falcon to hang when there is a network partition.

Finally, Falcon offers no access control. This is a vulnerability because starting a too-short end-to-end timer via the reliable failure detector interface can needlessly crash a machine. A real deployment of Falcon would need to extend the client library and spies so that buggy or malicious applications could not send KILL() RPCs to spies.

Chapter 4

Albatross: using new network interfaces for fast failure reporting

And I had done a hellish thing,
And it would work 'em woe:
For all averred, I had killed the bird
That made the breeze to blow.
Ah wretch! said they, such birds to slay,
That made the breeze to blow!

- Samuel Taylor Coleridge, *The Rime of the Ancient Mariner*

In the prior chapter we described Falcon, a failure reporting service that exposes a reliable failure detector interface in order to realize the benefits of Chandra and Toeug's perfect failure detector [21]. However, Falcon cannot handle network failures because its spies rely on network communication to report the failure of their monitored layers to the client library. Thus, Falcon's clients, by design, remain ignorant of any failures while there is a network partition. This design decision was based on the assumption that network failures are both infrequent and catastrophic (e.g., they disrupt vital services like DNS).

Unfortunately, this assumption—even in the context of data centers—is incorrect. After building Falcon, we analyzed a year-long trace of the failures that occurred in

This chapter revises [69]: J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. *Taming uncertainty in distributed systems with help from the network*, In EuroSys, Apr. 2015. Co-authors Marcos K. Aguilera and Michael Walfish contributed to the presentation and design of Albatross. Trinabh Gupta contributed to the design, implementation, and evaluation of Albatross.

several data centers of a company with a strong Internet presence, using similar methods to Gill et al. [44]. Data center operators collected the events generated by in-device monitoring into a central repository using monitoring protocols (such as SNMP [18]) or manual intervention. Events have associated meta-data, 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 large data centers (more than a thousand network elements) had about 12 partitions per month, of which about half disconnected an entire rack; the rest 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.

We present *Albatross* to address the problem of network failures, a failure reporting service that uses the network itself, both for gathering inside information and for converting suspicion into fact. The key observation in the design of Albatross is that the programmable interfaces exposed by modern data center networks, such as Software Defined Networks (SDNs), can be readily used for both purposes. Specifically, Albatross uses SDN to gather inside information about network and host failures,[1] and to prevent processes suspected of failure from communicating on the network.

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

In addressing the first two challenges of building a fast failure reporting service (Chapter 1), Albatross uses SDN (and a Falcon spy) to systematically collect inside information, and, like Falcon, Albatross exposes a binary interface for reporting failures. However, Albatross cannot sidestep the fundamental issues of communicating failures

[1]Albatross uses a modified Falcon spy to gather inside information about process failures.

across network partitions. Instead, Albatross relaxes the guarantees of the reliable failure detector interface by providing *asymmetric* guarantees: it categorizes processes as *excluded* or *non-excluded* and promises reliable answers only to non-excluded processes. We find that these guarantees are useful to applications, and sufficient to provide the benefits of Falcon's reliable failure detector interface, under certain conditions (§4.2).

By using SDN to block individual processes, Albatross reduces its negative impact; Albatross avoids the collateral damage of killing hosts (which exists in Falcon) and can make progress during network partitions without disconnecting entire switches or subnetworks. Specifically, Albatross never disables an entire switch, or even an entire end-host. However, Albatross's drop rules consume a scarce resource, namely the memory in switches for storing these rules [52]. To work within this constraint, Albatross names processes according to their starting time and enclosing application; this naming scheme allows aggregation of drop rules when failures affect many processes.

Albatross faces an additional challenge in that Albatross itself a distributed system and, as such, is subject to the very failures that it wishes to detect and report. To be useful, Albatross must function under reasonable and common failures. (As an analogy, a fire alarm must function under usual types of fires.) To handle these failures, Albatross internally uses the replicated state machine approach [66, 97] and a majority-based technique (Paxos [67]), which requires that a majority of the servers are responsive and mutually connected; Albatross achieves this by requiring careful placement of its managers (§4.3.3).

In evaluating Albatross (§4.5), we find that it has low costs: it requires a small amount of 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 ZooKeeper [53] configured to act as a failure reporting service (we will refer to this service as simply "ZooKeeper"). This advantage owes to the design of Albatross: if ZooKeeper were to lower its timeouts to achieve the same speed, its servers would be overwhelmed (§4.5.1).

Furthermore, we demonstrate that Albatross's guarantees are useful to applications: Albatross can be used similarly to a perfect failure detector in distributed algorithms, despite its different guarantees (§4.5.2).

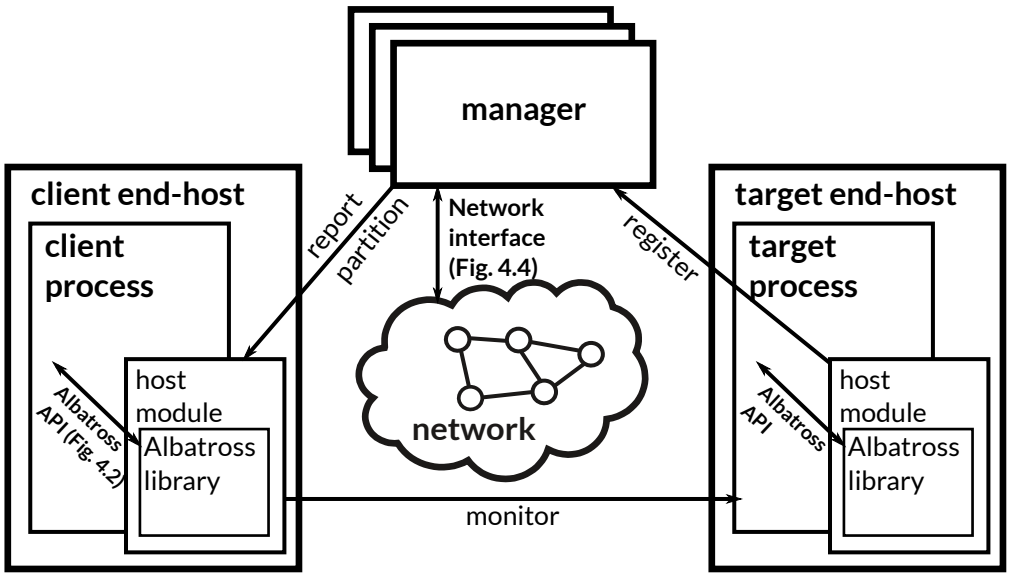

Figure 4.1: High-level view of Albatross. The host module provides the API through which applications use Albatross; the host module helps detect crashes of local processes. The manager is replicated at dedicated servers and coordinates Albatross's response to network and host failures. The manager interacts with the network through an abstract interface, and notifies clients about partitioned processes.

4.1 Overview of Albatross

Albatross is a failure reporting 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.

Components. Figure 4.1 depicts the components of Albatross. We survey them briefly below. (Section 4.3 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 repli-

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 4.2: The Albatross API.

cation [66, 97]. 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 from Falcon (Chapter 3). The host module also implements the Albatross API, described immediately below.

Albatross API. Figure 4.2 shows the Albatross API; Figure 4.3 gives an example use of the API and explains what causes communication among the components in Figure 4.1. 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().

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, which 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.

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 "de-register" 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 4.3: Example sequence of actions using the Albatross API and the resulting communication by Albatross.

Informal contract. Albatross covers all host failures and common network failures. Its reports are reliable but asymmetric: it excludes some processes, and promises that "disconnected" reports are reliable 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 4.2.

In addition, Albatross provides fast (sub-second) detection time achieved 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 negative impact by using inside information.

Rationale. Reporting *all* network failures is impossible [40]. Similarly, providing reliable, symmetric reports seems infeasible: how can a service 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 "pls grant degree" always, which is feasible to implement but useless). Fortunately, Albatross's guarantees are useful to applications (§4.5.2), though there are some small corner cases, covered in the next section.

4.2 Albatross's contract

This section precisely describes Albatross's guarantees. We will define a set of *excluded* processes, and the guarantees will be asymmetric: assurances are granted only to processes outside the excluded. The high-level concepts of exclusion and asymmetric guarantees have appeared before [13, 23] but not, to our knowledge, in our specific context of failure reporting services.

Albatross's guarantees refer to a notion of *time*, which is a *logical* time; 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, Albatross should aim above the letter of its contract and should thus report failures quickly. 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 4.1 (processes register, set backstop timeouts, etc.), and if a majority of Albatross managers remains alive and mutually connected (per the fault-tolerance discussions at the start of this chapter and in 4.3.3).

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 [66] 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".

As an example, we consider a primary-backup application [4], and then explain when the guarantees of Albatross require some care in the context of this example. In a primary-backup application, the primary receives a request from a client, replicates that request at the backup, and only then executes the request and responds. This setup provides fault-tolerance through the invariant that replication happens before responding to the requestor. For availability, the application needs a way to make progress if the primary or backup fails, which is where a failure reporting service like Albatross comes into play. If a backup learns that its primary is "disconnected" it can immediately take over and operate autonomously because it knows that Albatross is preventing the (old) primary from using the network.

We now consider the effect of the consequences listed above for a primary-backup application. First, suppose 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).

Third, we consider 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.

Finally, the application may be unavailable while the backup waits to learn of the primary's failure.

4.3 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 4.4 describes notable implementation details.

4.3.1 Names and identifiers

Under Albatross, each target process receives a *process id (pid)* when it registers (§4.1) 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:

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 4.4: Network interface used by Albatross.

- A *host id* (for example, an IP address);
- An *application id* (appid), which is programmer-supplied and unique to applications within the given network (§4.1);
- 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.

4.3.2 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 4.6). Here, we describe the functionality in terms of an abstract interface, depicted in Figure 4.4. (Section 4.4.2 describes an implementation of this interface, using OpenFlow and NOX [48].)

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 (§4.3.1). 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 4.4). 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. These events are Albatross's inside information from the network, and are sent to the manager, as described next.

4.3.3 Manager

Albatross's manager coordinates the response to end-host and network 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 not using Albatross; it also does not affect applications that use Albatross but are launched after the failure is resolved. (One can think of this approach as virtualizing partitions, in that different applications see different views of the network topology.)

In more detail, the manager runs the logic in Figure 4.5. The manager maintains a model of the current network topology. Upon starting the manager requests notifications about topology changes using SUBSCRIBE (our implementation assumes that the manager also begins with a correctly configured base topology; a more ambitious implementation could build the topology as switches join). When the manager receives 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,

```
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) // may generate exception
      else: // switch.port connects to another switch
        for each epoch in activeEpochs:
          call CUT-EPOCH(switch, epoch, switch.port) // may generate exception
        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 4.5: Logic for detecting, enforcing, and reporting partitions.

so enforcing a partition requires few per-application rules. On the other hand, if the port bordering *P* connects to another switch, 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.[2] 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.

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 (§4.1) fires; this event causes the monitoring process to request help from the manager, which disconnects the target process, using BLOCK. Figure 4.6 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 4.4.3.

[2]This failure is detected with timeouts in the SDN control network. These timeouts differ from *end-to-end* timeouts because, first, they are monitoring a constrained component, and, second, SDN control traffic can be prioritized, which would make message latencies predictable and thus avoid spurious timeouts. This is an example inside information.

```
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 4.6: Logic to handle client backstop timeout on target *pid*.

Example. Consider the example network in Figure 4.7. If switch 3 reports to the manager an *end-host failure* event about host 2, 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), 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 uses 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 4.4.2, 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 (§4.1), following the state machine replication approach [66, 97] with a majority-based technique for fault tolerance (Paxos [67]). If manager replicas are placed at diverse parts of the network (such as different racks), then under common network partitions (described in at the start of this chapter), the majority of servers remains with the majority of network elements.[3]

[3]Even if the manager-majority partition holds a minority of processes of a particular application, that minority can continue operating if the application does not use majority-based algorithms. Applications using techniques like primary-backup [4] or chain replication [107] can make progress with even one working process.

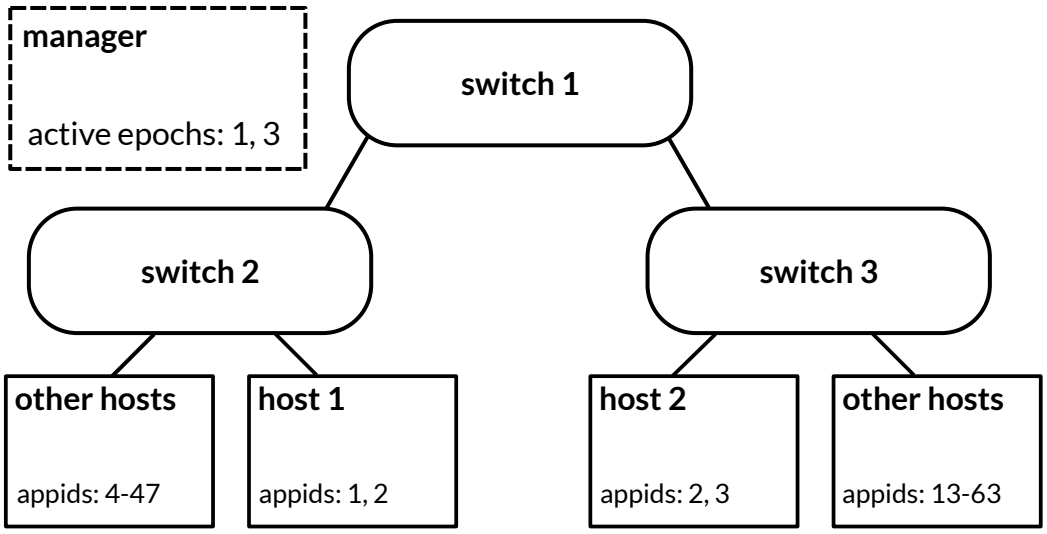

Figure 4.7: Network for example of how Albatross's manager enforces a partition.

Revisiting the guarantees. The formal guarantees (§4.2) 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 4.6). Albatross guarantees Isolation by configuring network switches to drop the traffic of excluded processes (Figure 4.5). Albatross guarantees Monotonicity because it never unblocks processes; however, it does recycle identifiers as described in the next section. The first part of Correspondence is also provided 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 part of Correspondence is provided by the sequencing of events; the manager reports partitions only after enforcing them.

Albatross provides speed by reacting to inside information (i.e., failure events)

as opposed to end-to-end timeouts, in the common case. It limits negative impact by inspecting network state, using surgical rules, and allowing reconnection (described next).

4.3.4 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 (§4.4.1), 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 (§4.1,§4.2). 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 (§4.2); if the CUT-APP rules are undone before the excluded processes are blocked by their host module locally, 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 that a host module were to give to a new process the pid of a recently terminated process; 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 which clients have references in a persistent write-ahead log; periodically, the local host module queries remote host modules to confirm that clients referencing its allocated pids are still running.

4.4 Selected implementation details

4.4.1 Packet marking

Figure 4.8 depicts the format of a pid. It consists of a 4-byte host identifier (the host's IP address in our implementation) together with 16 per-process bits. The per-process bits are the process's 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 zero. 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 [60]).

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

Figure 4.8: 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 (§4.3.4). The local id disambiguates multiple processes of the same application on the same host.

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 4.6.

4.4.2 Network interface implementation

Our implementation of Albatross assumes a network with OpenFlow switches and a NOX controller [48]. Given this environment, one can implement the network interface (Figure 4.4, page 59) 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 (§4.3.2). *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 we describe next. 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 second (the smallest OpenFlow timeout); this setting provides reasonably fast detection while tolerating dropped or delayed heartbeats.

4.4.3 Detecting process crashes

As noted in Section 4.3.3, process crash detection involves an additional module. That module a Falcon spy (Chapter 3) modified to drop an unresponsive process's traffic locally, instead of terminating it with a SIGKILL. This modification reduces the negative impact of a false suspicion.

4.4.4 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 4.2) 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 [77]. (§4.6 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, with the tradeoff that that Albatross treats such processes as a unit.

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 4.9: Panel of synthetic failures. We inject failures in the network, at the end hosts, and into Albatross itself.

4.5 Evaluation of Albatross

In this section, we evaluate how Albatross addresses the three challenges of Chapter 1—systematically collecting inside information, choosing an interface for reporting that information, and limiting negative impact. First, we evaluate the benefits of using inside information in Albatross's architecture for reporting host and network failures (§4.5.1). Second, we evaluate Albatross's interface by reprising the benefits of a failure reporting service that provides reliable reports of failure showing in the process that Albatross's specific contract is sufficient to derive these benefits (§4.5.2). Third, we evaluate how well Albatross limits its negative impact (§4.5.3).

All experiments run on a prototype network (with 13 switches connected in a complete ternary tree) implemented using QEMU/KVM [88] virtual machines (version 1.0.1) and CPqD OpenFlow 1.2 software switches [84]. The hypervisor is a 64-core Dell PowerEdge R815 with AMD Opteron Processors and 128 GB of memory, running Linux (kernel version 3.7.10-gentoo-r1). The network controller is NOX [48], modified to work with OpenFlow 1.2 [80].

failure type	action taken by Albatross
link failure	the network interface reports link failures (§4.3.2); the manager detects a partition, enforces an excluded set, and reports it to clients (Figure 4.5)
switch failure	handled as the above
network misconfiguration	detected by client timeout (§4.1) and enforced by the manager's backstop logic (Figure 4.6)
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 4.5)
spurious failure event	handled as a link failure (see above)
host crash	detected with an end-host failure event (§4.3.2), and enforced by the manager (Figure 4.5)
process crash	Falcon spy (Ch. 3, §3.3) detects and reports failure (§4.4.3)
crash of host module	detected by backstop timeout (§4.1) and enforced by the manager (§4.3.3)
crash of manager replica + partition	replication library (§4.4.4) elects a new leader (§4.3.3), then the manager handles the failure as above

Figure 4.10: Albatross's reaction to the failure panel (Figure 3.9). Albatross detects all failures save network flooding (a non-failure), and its enforcement actions affect only applications that use it.

4.5.1 Does Albatross's design yield a fast failure reporting service?

We now experimentally investigate the qualities of Albatross: (a) how it responds to failures, and (b) how its timeliness compares with two baseline mechanisms. The experiments use a panel of synthetic failures, depicted in Figure 4.9. These failures model problems in the network, at end-hosts, and in Albatross itself; link and switch failures are derived from the failure analysis described at the start of this chapter. 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

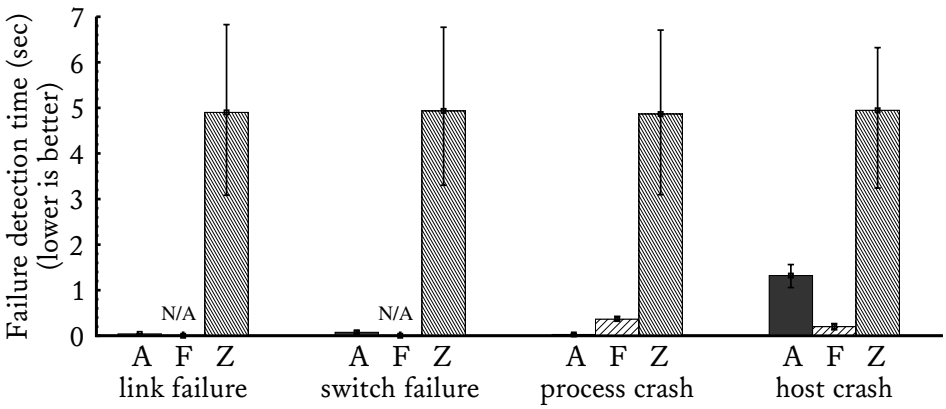

Figure 4.11: Detection time and coverage of Albatross (**A**), compared to Falcon (Chapter 3) (**F**) and ZooKeeper [53] (**Z**). Error bars are minimum and maximum observed detection times. Two of Falcon's bars are labeled N/A because it does not detect link or switch failures. Albatross detects failures quickly by using information at end-hosts and in the network. ZooKeeper's detection time reflects its timeout (4 seconds); a shorter one causes overload (see text).

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 4.10.

How does Albatross compare with baseline mechanisms? We take as baselines (1) ZooKeeper [53], and (2) Falcon. 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 the results of Chapter 3 (because two of Falcon's spies are incompatible with the testbed used for Albatross).

Figure 4.11 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 switches; 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 results include a delay for confirming that a process has left the process table whereas Albatross needs only to install an iptables rule (§4.4.3).

On host failures (e.g., kernel panics), Albatross takes 1 second longer than Falcon; the difference is that Albatross detects host failures using OpenFlow rule timeouts (§4.4.2), which have a minimum duration of 1 second. However, unlike Albatross, Falcon cannot detect switch or link failures.

ZooKeeper's detection speed reflects its timeout, which we configure to be 4 seconds, as suggested in its tutorial [120]. 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. First, we find that ZooKeeper can monitor 1500 targets, each using a 4 second 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 [15]: timeouts shorter than 12 second overwhelmed Chubby's servers in Google's clusters (which monitor 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.

4.5.2 What are the benefits of Albatross's contract?

Chandra and Toueg established that perfect failure detectors (which allow all processes to detect all crashes correctly) enable "easier" algorithms than unreliable failure detectors, such as those based on end-to-end timeouts [21] (see also Chapter 2, §2.1 and Chapter 3, §3.4.5). As just one example, Chain Replication [107] (a form of primary-

backup) is simpler than Viewstamped Replication [81], Paxos-based replication [67], and Raft [82].

Albatross's contract (§4.2), with its asymmetric guarantees, does not precisely meet this theoretical ideal. 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 reliable reports of failures; demonstrating that Albatross's guarantees are sufficient to simplify distributed algorithms; and describing the subtle relationships among Albatross, ZooKeeper, and majority-based agreement.

Without reliable reports, what can go wrong? We use RAMCloud [83] 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. RAMCloud could use a reliable failure reporting service, but RAMCloud instead 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.

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 (§4.2). This benefit is not unique to Albatross; other failure reporting services (such as Falcon) could have prevented this error.

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 reliable reports [33]. We specifically compare (a) *Zab* [57], a protocol that uses majority-based agreement (as opposed to reliable reports), and (b) *Aab*, a protocol that uses Albatross.

Zab: atomic broadcast without reliable reports. Zab [57] 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 (§4.2), the other leader is excluded—a 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 [33], 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

Algorithm	Aab (§4.5.2)	Zab [57]
# 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 processes (n)	$f < n$	$f < n/2$

Figure 4.12: 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 [57] uses majority-based agreement; both algorithms are described in the text.

the recovery phase, the leader completes the broadcast of pending messages from prior periods (if any). 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 4.12 compares the two algorithms. Aab has 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.

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 (§4.1, §4.3.3) 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.

The same kind of amortization only works partway for Zab. 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. However, ZooKeeper cannot achieve the same performance under the same number of clients as Albatross because short leases require frequent polling, which can overwhelm a server with many clients; this is demonstrated in Section 4.5.1.

ZooKeeper could be modified to use Albatross. 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.

And, Albatross could be modified to use ZooKeeper. 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.

4.5.3 How does Albatross limit its negative impact?

We now evaluate how well Albatross limits its negative impact, focusing on its mechanism for disconnecting processes and the resources it uses at end-hosts and in the network.

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 (§4.3.4). Reconnection takes about one second in our experiments.

max additional rules installed at a switch	
rules installed	1 rule
CPU usage per component (§4.3)	
host module	1.8 %
manager	0.03 %
bandwidth used	
at each end-host	61.0 kBps
at manager	6.9 kBps

Figure 4.13: Summary of Albatross's costs under link failure. Albatross uses few resources. Scalability is discussed in the text.

What are Albatross's other costs? We measure Albatross's resource cost for detecting and enforcing a partition for a single application. Figure 4.13 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. Numbers like these are acceptable: the HP ProCurve J9451A switch, for example, has capacity of 1500 OpenFlow rules [52]. However, 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 (§4.3.3), 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 (§4.4.2). 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.

4.6 Summary & frequently asked questions

Albatross leverages SDN to gather inside information from the network and to enforce its decisions about failures. This choice yields a design that can report failures quickly while avoiding the collateral damage of killing entire machines, and it allows Albatross to extend its coverage beyond Falcon's to include common network failures. We now answer some common questions about Albatross.

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 (§4.3.2), 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 Chapter 2, Section 2.4).

Must Albatross repurpose the source MAC field? Albatross's embedding of process identifiers in packets' source MAC field (§4.4.1) 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 [54], 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 (§4.4.1), a Layer 2 protocol deployed alongside Albatross should be modified to never learn

from these packets. Fortunately, Albatross works in the context of SDNs, and so many Layer 2 protocol changes would require only software changes at the SDN controller.

Can Albatross work with virtual machine migration? Albatross assumes that processes and end-hosts remain stationary, which conflicts with virtual machine migration [25]. This issue is surmountable, if the manager and migration mechanism collaborate to migrate filter rules, though we do not implement this.

Does Albatross consider network policy? Albatross models only the *physical* network topology (§4.3.3). 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 (§4.1, §4.3.3).

Can cooperating applications have inconsistent views of the network? As mentioned in Section 4.3.3, one can think of Albatross as virtualizing partitions. This "virtualization" does not cause discrepancies in how applications see the network: 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 4.2). Adding access control to Albatross's API is future work.

Chapter 5

Pigeon: reporting inside information without violence

> *Cher Ami was a homing pigeon owned and flown by the U.S. Army Signal Corps in France during World War I. He helped save the Lost Battalion of the 77th Division in the battle of the Argonne, October 1918. In his last mission, he delivered a message despite having been shot through the breast, being blinded in one eye, covered in blood, and having a leg hanging only by a tendon. The bird was awarded the Croix de Guerre for heroic service delivering 12 important messages in Verdun, France.*
>
> - http://nationalpigeonday.blogspot.com/, retrieved Apr. 2015

As argued in Chapter 1, uncertainty about whether a process is crashed or merely slow is fundamental—even with access to local information. Even a Falcon spy, for example, can be uncertain about whether a layer has failed (Chapter 3). Uncertainty must be handled carefully to avoid problems like split-brain scenarios.

Failure reporting services can handle uncertainty on behalf of their clients by killing components when they suspect failure. However, killing to hide uncertainty brings several issues: killing can take out functioning-but-slow process (see Chapter 3, §3.4.3); killing can cause collateral damage, even when used judiciously (as discussed in prior chapters); and killing is unnecessary if an application's recovery strategy can

This chapter revises [68]: J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. *Improving availability in distributed systems with failure informers.*, In NSDI, Apr. 2013. Co-authors Marcos K. Aguilera and Michael Walfish contributed to the presentation and design of Pigeon. Trinabh Gupta contributed to the presentation, design, implementation, and evaluation of Pigeon.

handle uncertainty (see Chapter 3, Section 3.4.2), which is the case for most existing applications.

These problems with killing motivated us to consider a failure reporting service that forgoes guaranteeing reliability for every report in favor of not killing. An application would benefit from such a service because it could still use inside information to report failures quickly. For example, Falcon-NoKill reports failures quickly without reliable reports, thereby allowing ZooKeeper [53] to recover from some failures more quickly (see Figure 3.12 on page 40). Embracing uncertainty yields two other benefits: first, a failure reporting service's design no longer needs to consider the collateral damage of killing and second, the failure reporting service can guarantee that every failure is eventually detected by relying on end-to-end timeouts as a backstop.[1]

In the rest of this chapter, we describe Pigeon, a failure reporting service that uses inside information and does not kill. Pigeon extends Falcon's architecture to include network switches and routers to address the first challenge of coherently gathering inside information, with an emphasis on reusing existing techniques and information.[2] The combination of nonviolence and Pigeon's architecture helps to keep its negative impact low, so only one of the three challenges presented in Chapter 1 remains: choosing an interface for reporting failures.

Pigeon implements a *failure informer* interface; this interface exposes four *failure conditions* to its clients, each of which abstracts a different kind of failure, and can lead to different recovery actions. An alternative would have been to keep a binary interface of "up" and "down", but with no guarantees associated with either report; however, such an interface would miss an opportunity, as inside information about failure can inform recovery action. For example, if a lease server [46] knows that a lease holder has crashed (because its main process is not in the process table), then the lease server can immediately revoke that lease without waiting for a (potentially lengthy) timeout to expire.

[1]Although both Falcon and Albatross include a backstop timeout on end-to-end behavior, neither service reports a failure to clients until the service itself has taken some action because an end-to-end timeout does not guarantee that a failure occurred.

[2]Albatross (Chapter 4) also uses inside information from the network, but it requires a higher-level interface than Pigeon (see Chapter 4, §4.3.2).

In evaluating Pigeon (§5.4), we find this interface benefits applications not only by reporting failures quickly, but also by allowing applications to react in qualitatively different ways to different failures. Furthermore, in a multidimensional study we find that the failure informer interfaces allows Pigeon to improve upon Falcon and other failure reporting services in at least one of the following: coverage (what failures can be detected), detection time, or information beyond "up" or "down". Both of these benefits come at a low cost, in that Pigeon uses few system resources.

In the rest of this chapter we expand on the design choices in Pigeon (§5.1), explain Pigeon's design (§5.2), present a prototype implementation (§5.3), evaluate that prototype (§5.4), and summarize and discuss the overall approach (§5.5).

5.1 Design challenges and principles

As noted in Chapter 1, building a fast failure reporting service that uses inside information presents three challenges: systematically collecting inside information, choosing an interface to report such information, and limiting negative impact. We now explain Pigeon's guiding principles in response to these challenges.

Renounce killing. Reliable reports of failure are useful to applications because they eliminate the need to handle uncertainty from a failure reporting service, and one way to provide reliable reports of failure is through killing, either in effect [13] or in fact [39, 73]. Killing requires a failure reporting service to judge whether some part of the system is working, and these judgments can be wrong, thereby causing unnecessary damage. To make matters worse, killing can cause collateral damage whereby many healthy processes are killed for the sake of giving reliable answers about a single suspected process. To avoid the negative impacts of killing, Pigeon renounces violence entirely.

Provide full coverage. Without killing, Pigeon can rely on the universal backstop: an end-to-end timeout. This backstop is different from Falcon's backstop because it does not require communication among any components of the system. Using end-to-end

timeouts for common failures is a poor choice (as we argued in Chapter 1), so Pigeon still uses inside information for fast detection in the common case.

Expose uncertainty. Some inside information about failure is certain (e.g., a process absent from a process table is certainly crashed), while other inside information is not (e.g., a process present in the process table can be unresponsive to interprocess communication). Exposing when Pigeon is certain about failures allows applications to take qualitatively different recovery actions, for example by immediately selecting a new leader instead of invoking a leader election algorithm. Applications must still handle uncertainty, but this is not a burden as applications do so already when end-to-end timeouts expire.[3]

Design for extensibility. No implementation is ever perfect, so we design for extensibility: Pigeon accommodates add-on modules that provide better information and indicate different kinds of faults, potentially expanding the kinds of failures that it can report quickly. These extensions do not require redesigning Pigeon or applications; a key factor in avoiding redesign is exposing failures through an abstraction, versus exposing all details.

5.2 Design of Pigeon

This section presents the interface exposed by Pigeon (§5.2.1), describes the guarantees (§5.2.2), explains how Pigeon is used (§5.2.3), describes its architecture (§5.2.4), and explains errors and their effects (§5.2.5).

5.2.1 The failure informer interface

The failure informer interface exposes *conditions* to applications, where each condition abstracts a class of problems in a remote *target process* that all affect the distributed application in similar ways. There are four conditions, shown in Figure 5.1.

[3]In Chapters 3 and 4 we argued in favor of reliable reports of failure because they can be used to implement simpler distributed algorithms, but many existing applications already use majority-based algorithms to handle uncertainty. Such algorithms can use Pigeon (or Falcon, or Albatross) directly.

condition	occurred?	permanent?	description	example causes
stop	certain	certain	target stopped executing	core dump, machine reboot
unreachability	certain	uncertain	target unreachable	network link down
stop warning	expected; imminent	certain	target may stop executing	disk about to crash
unreachability warning	expected; imminent	uncertain	target may become unreachable	network link close to capacity, CPU overloaded

Figure 5.1: Conditions reported by Pigeon. These conditions abstract specific failures affecting a remote *target* process and encapsulate two kinds of uncertainty.

1. In a *stop*, the target process has stopped executing and lost its volatile state. The problem has already occurred and it is permanent. This condition abstracts process crashes, machine reboots, and similar problems.

2. In an *unreachability*, the target process may be operational, but the client cannot reach it. The problem has already occurred, but it is potentially intermittent. This condition abstracts a timeout due to, say, a network partition or a slow process.

3. In a *stop warning*, the target process may stop executing soon, as a critical resource is missing or depleted. The problem has not yet occurred, but if it does occur it is permanent. This condition abstracts cases such as a report about an imminent disk failure [50, 86, 103].

4. In an *unreachability warning*, the target process may become unreachable soon, as an important resource is missing or depleted. The problem has not yet occurred; if it occurs, it is potentially intermittent. This condition abstracts cases such as a network link being nearly saturated or overload in the host CPU of the target process.

The four conditions above reflect a classification based on two types of uncertainty that are useful to applications: uncertainty in permanence (stop vs. unreachability) and uncertainty in occurrence (actual vs. warning).

function	description
h = init(target, callback)	request monitoring of target process; returns a handle for use in future operations
uninit(h)	stop monitoring
c = query(h)	get status; returns a list of conditions
res = getProp(h, c, propname)	get condition property value
startTimer(h, timeout)	set/reset timer
stopTimer(h)	cancel timer

Figure 5.2: Pigeon's programmatic interface.

The interface also returns *properties*: information specific to the condition, which may help applications recover. A property of all conditions is their expected duration. (Note that a duration estimate does not subsume certainty: certainty-vs-unreachability captures a quality other than duration, and the duration estimate itself is fundamentally uncertain.[4]) We describe how the duration property is set in Section 5.3.4. A property of warning conditions is a bit vector indicating the critical resource(s) responsible for the warning (disk, memory, CPU, network bandwidth, etc.).

Client API. Client applications use the interface in Figure 5.2.

The client calls init() to monitor a target process, named by an IP address and an application identifier in some name space (e.g., port space). The function returns a handle referencing the target process which is used in other functions. The init() function takes as a parameter a callback function, which the implementation calls as new failure conditions emerge.

The query() function returns a (possibly empty) list of active conditions. The getProp() function returns properties, described above.

The startTimer() and stopTimer() functions start and cancel end-to-end timeouts. Clients use timeouts as a catch-all: if the client does not cancel or reset the timer before it expires, Pigeon reports an unreachability condition.

[4]In fact, a failure informer can report an unreachability with indefinite (unknown) duration. This is different from a stop, which is known to be permanent.

5.2.2 Guarantees

We now describe the guarantees provided by Pigeon along three axes: coverage, accuracy, and timeliness. Pigeon provides these guarantees in spite of failures in both the network and Pigeon itself, as described in Section 5.2.5.

Coverage. If a client uses Pigeon's end-to-end timeout, Pigeon guarantees full coverage: if a target process stops responding to the client, Pigeon reports either a stop or an unreachability condition.

Accuracy. Pigeon's accuracy guarantee means that any failure Pigeon reports is justified; we address the correctness of duration estimates in Section 5.4.1. We designed Pigeon not for perfect accuracy in its reports but for accuracy in its certainty: Pigeon knows when it knows, and it knows when it doesn't know. Specifically, if Pigeon reports a stop condition, the application client can safely assume that the target process will not continue; Pigeon returns an unreachability when it cannot confirm that the condition is permanent. When Pigeon reports a warning, it guarantees that a motive exists (*some* fault occurred) but not that an unreachability or stop will occur.

Timeliness. If a condition occurs, Pigeon reports it as fast as it can. This is a "best effort guarantee."

5.2.3 Using the interface

We now give a general description of how applications might use Pigeon; Section 5.4.2 considers specific applications (RAMCloud [83], Cassandra [65], lease-based replication [46]). For each of the four conditions, we explain the implications for the application and how it could respond.

Recall that a stop condition indicates that the target process has lost its volatile state and stopped executing permanently; this has both a quantitative and a qualitative implication. Quantitatively, it is safe for the client to initiate recovery immediately. Qualitatively, the client can use simpler recovery procedures: because it gets closure—that is, because it knows that the target process has stopped—it does not have to handle

the case that the target process is alive. For example, a stop condition allows the client to simply restart the target on a backup.

By contrast, an unreachability condition implies only that the target is unreachable; the target process may in fact be operational, or the condition may disappear by itself. This has two implications. First, if the client takes a recovery action, the system may have multiple instances of the target process. Recovering safely therefore requires coordinating with other nodes using mechanisms like Chubby [15], ZooKeeper [53], or Paxos [67], which allow nodes to *agree* on a single master or action. Note that reports of unreachability are still useful—and that using these agreement mechanisms is not overly burdensome—because systems already have the appropriate logic: this is the logic that handles the case that an end-to-end timeout fires without an actual failure.

Second, based on the expected duration of the condition, the application may consider the costs and benefits of just waiting versus starting recovery proactively. Conceptually, each application has an *unavailability threshold* such that if the expected duration of the condition is smaller, the application should wait; otherwise, it should start recovery.

In fact, "eager recovery" can be taken a step further: warnings allow applications to take precautionary actions even without failures. For example, a stop warning could cause an application to bring a stand-by from warm to hot, while an unreachability warning could cause an application to degrade its service.

To illustrate the use of Pigeon concretely, consider a synchronous primary-backup system [4], where the primary serves requests while a backup maintains an up-to-date copy of the primary. The backup can use Pigeon to monitor the primary:

- If Pigeon reports a stop, the backup takes over;
- If Pigeon reports an unreachability, the backup must decide whether to fail over the primary, or instantiate a new replica (either of which requires mechanisms to prevent having multiple primaries), or simply wait. These decisions must weigh the cost of the recovery actions against the expected duration of the condition.
- If Pigeon reports a stop warning, the backup provisions a new replica without failing over the primary.

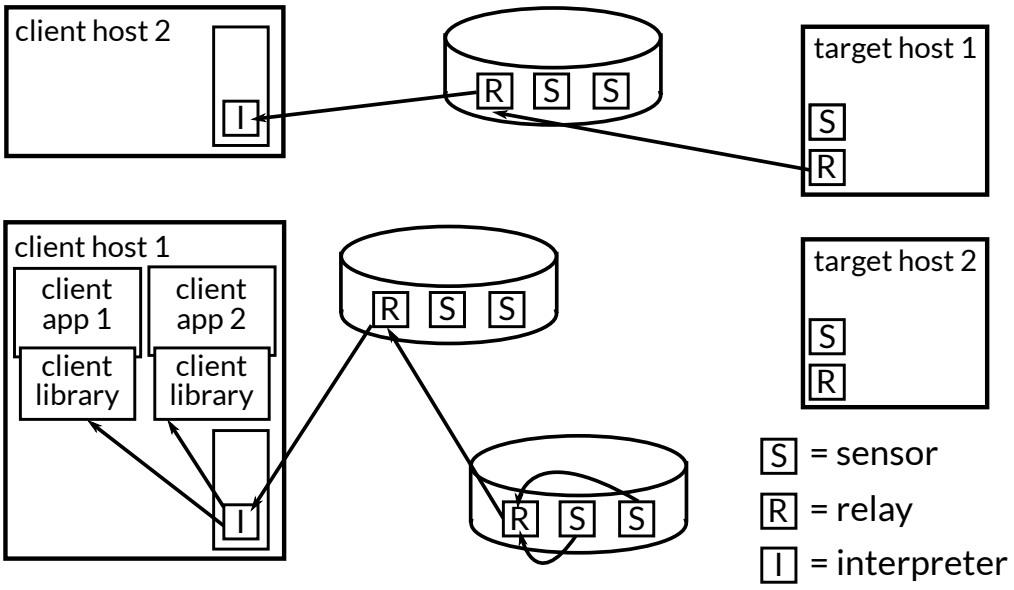

Figure 5.3: Architecture of Pigeon. Pigeon has sensors (S), relays (R), and interpreters (I). Sensors are component-specific. Sensors and relays are shared by multiple clients and end-hosts; an interpreter is shared by all client applications on its host. The client library presents the client API (§5.2.1) to applications.

- Under an unreachability warning, the backup logs the warning so that, if the condition is frequent, operators can better provision the system in the future.

5.2.4 Architecture of Pigeon

As stated in Chapter 1, Pigeon targets a data center network under a single administrative domain. Pigeon's architecture is geared toward extracting and exploiting inside information about failures already present in the system; for example, the failed links in a network collectively yield information about a network partition. To use inside information, Pigeon needs mechanisms to (a) sense information inside components, (b) relay information to end-hosts, and (c) interpret information for client applications. These mechanisms are embodied, respectively, in *sensors*, *relays*, and *interpreters* (Figure 5.3). We describe their abstract function below and their instantiations in our prototype in

Section 5.3.

A *sensor* is component-specific and tailored; it is embedded in a component, and detects faults in that component. A *fault* is a local event, possibly a malfunction, that may contribute to one of the four failure conditions (§5.2.1). A *critical fault* is one that may lead to a stop condition; a *regular fault*, to an unreachability condition; and an *advisory fault*, to a warning condition. Faults need not cause conditions; they may be masked by recovery mechanisms outside the application (e.g., route convergence).

Relays communicate with sensors and propagate these sensors' fault information to end-hosts. Sensors and relays may be installed for Pigeon or may already exist in the system.

Each end-host has an *interpreter* that receives information about faults from the relays. Interpreters render this information as failure conditions and estimate the expected duration of conditions. Clients interact with interpreters through a *client library*, which implements end-to-end timeouts and the client API (§5.2.1). Interpreters also determine which sensors are relevant to the client-supplied identifier for a target (§5.3.4).

5.2.5 Coping with imperfect components

In this section we describe the effect of errors in Pigeon's own components and the network. These errors include crash failures and misjudgments; they do not include Byzantine failures, which Pigeon does not tolerate. Figure 5.4 summarizes the effect of errors.

Before continuing, we note *non*-effects. First, Pigeon does not compromise on coverage since it uses a backstop end-to-end timeout; this timeout is implemented in the client library (linked into the application) and hence shares fate with the client application, despite failures elsewhere. Second, Pigeon is designed to not compromise *safety*, meaning that Pigeon guarantees stop conditions are reliable, by design (§5.3).

If a sensor, relay, or interpreter crashes or is disconnected from the network, Pigeon loses access to inside information, which affects accuracy and timeliness (§5.1). Loss of inside information also causes missed opportunities to report some failures as stop conditions (e.g., remote process exit) rather than as unreachability conditions

compromise	cause
coverage	nothing
safety	nothing
timeliness	sensor, relay, or interpreter crashes
	sensor misses fault
	interpreter does not report stop or unreachability
accuracy	sensor, relay, or interpreter crashes
	sensor falsely detects regular or advisory fault
	interpreter falsely reports unreachability or warning

Figure 5.4: Effect of errors on Pigeon's guarantees. Errors in duration estimates are covered in Section 5.4.1.

triggered by the backstop end-to-end timeout.

Pigeon may need to rely on the backstop end-to-end timeout if a sensor does not detect a fault, compromising timeliness. If a sensor falsely detects a regular fault, then Pigeon may misreport an unreachability condition. This error in turn compromises accuracy (potentially causing an unwarranted application recovery action) but not safety, as Pigeon reports unreachability conditions as unreliable. The effect when a sensor falsely detects an advisory fault is similar (misreports of warning conditions).

If the interpreter crashes or fails to report a condition, then Pigeon relies on the end-to-end timeout, again compromising timeliness. If the interpreter misreports an unreachability or warning, Pigeon compromises accuracy but not safety (as with a sensor). Errors in the interpreter's duration estimates are covered in Section 5.4.1.

We have designed Pigeon to be extensible so that new components can reduce the errors above. However, Pigeon's current components, which we describe next, already yield considerable benefits.

5.3 Prototype of Pigeon

We describe our target environment (§5.3.1), and the implementations of sensors (§5.3.2), relays (§5.3.3), and the interpreter (§5.3.4) used in our prototype. The prototype borrows many low-level mechanisms from prior work, as we will note, but the synthesis is new (if unsurprising).

5.3.1 Target environment

Our prototype targets networks that use link-state routing protocols, which are common in data centers and enterprises [47, 60]. Currently, the prototype assumes the Open Shortest Path First (OSPF) protocol [78] with a single OSPF *area* or routing zone. This assumption may raise scalability questions, which we address in Section 5.4.3. We discuss multi-area routing and layer 2 networks in Section 5.5.

We assume a single administrative domain, where an operator can tune and install our code in applications and routers; this tuning is required at deployment, not during ongoing operation.

5.3.2 Sensors

Sensors must detect faults quickly and confirm critical faults; the latter requirement ensures that Pigeon does not incorrectly report stops. The architecture accommodates pluggable sensors, and our prototype includes four types: a *process sensor* and an *embedded sensor* at end-hosts, and a *router sensor* and an *OSPF sensor* in routers. For each type, we describe the faults that it detects, how it detects them, and how it confirms critical faults. Faults are denoted F-⟨type⟩ (critical faults are noted in parentheses).

Process sensor. This sensor runs at end-hosts. When a monitored application starts up, it connects to its local process sensor over a UNIX domain socket. The process sensor resembles Falcon's application spy (Chapter 3, Section 3.3), but it does not kill. The sensor detects three faults:

F-exit (critical). The target process is no longer in the OS process table and has lost its volatile state, but the OS remains operational. This fault can be caused by a graceful exit, a software bug (e.g., segmentation fault), or an exogenous event (e.g., the process was killed by the out-of-memory killer on Linux). To detect this fault, the sensor monitors its connection to the target processes. When a connection is closed, the sensor checks the process table every $T_{proc\text{-}check}$ time units; after confirming the target process is absent, it reports F-exit. Our prototype sets $T_{proc\text{-}check}$ to 5 ms, a value small enough to produce a fast report, but not so small as to clog the CPU.

F-suspect-stop. The target process is in the process table but is not responding to local probes. An example cause would be a bug that causes a deadlock in the target process. To detect this fault, the sensor queries the monitored process every $T_{app\text{-}check}$ time units. If the target process reports a problem or times out after $T_{app\text{-}resp}$ time units, the sensor declares the fault. Our prototype sets $T_{app\text{-}check}$ to 100 ms of real time and $T_{app\text{-}resp}$ to 100 ms of CPU time of the monitored application (the same values are justified in Chapter 3, Section 3.3).

F-disk-vulnerable. A disk used by the target process has failed or is vulnerable to failure (based on vendor-specific reporting data such as SMART [103]). To detect this fault, Pigeon checks the end-host's SMART data every $T_{disk\text{-}check}$ time units, which our prototype sets to 500 ms.

Embedded sensor. The next sensor is logic embedded in the end-host operating systems. This sensor resembles Falcon's OS-layer spy but has additional logic to confirm critical faults without killing. It detects three faults:

F-host-reboot (critical). The OS of the target process is rebooting. The embedded sensor reports this fault during the shutdown that precedes a reboot, but only after all of the processes monitored by Pigeon have exited (waiting prevents falsely reporting a stop condition).

F-host-shutdown (critical). The OS of the target process is shutting down. The sensor uses the same mechanism as for F-host-reboot.

F-suspect-stop. The OS of the target process is no longer scheduling a high priority process that increments a counter in kernel memory every T_{inc} time units (Falcon's incrementer process, Ch. 3, §3.3). The sensor detects a fault by checking that the counter has incremented at least once every $T_{inc\text{-}check}$ time units. Our prototype sets T_{inc} and $T_{inc\text{-}check}$ to 1 ms and 100 ms, respectively, providing fast detection of failures with negligible CPU cost (we borrow these settings from Falcon).

Router sensor. A process on the router implements a sensor that detects two faults:

F-suspect-stop. An end-host is no longer responding to network probes. This fault could occur because of a power failure or an OS bug. The router sensor detects this fault by running a keep-alive protocol with any attached end-hosts. (This keep-alive protocol

is borrowed from Falcon.)

F-link-util. A network link has high utilization. Our prototype checks the utilization of the router's links every T_{util} time units and detects a fault if utilization exceeds a fraction F_{bw} of the link bandwidth. Our prototype sets F_{bw} to 63% (which we measured to be the lowest utilization at which a router starts to drop traffic in our testbed) and T_{util} to 1 second (which corresponds to the maximum rate at which this fault can be reported; see Section 5.3.3).

OSPF Sensor. A router's OSPF logic acts as a sensor that detects two faults:

F-link. A link in the network has gone down. The routers in our environment detect link failures using Bidirectional-Forwarding Detection (BFD) [59].

F-router-reboot. A network router is about to reboot. The sensor detects this fault because the operating system notifies it that the router is about to reboot.

5.3.3 Relays

The prototype uses three kinds of relays: one at end-hosts, called a *host relay*, and two at routers, called a *router relay* and an *OSPF relay*. Relays may be faulty, as discussed in Section 5.2.5.

Host relay. This relay communicates faults detected by the process sensor, and it runs in the same process as the process sensor. When a client begins monitoring a target process, the client's interpreter registers a callback at the target's host relay. The host relay invokes this callback whenever the process sensor detects a fault. Callbacks improve timeliness, as the interpreter learns about faults soon after they happen; this technique is used in other systems [26, 58].

Router relay. This relay communicates the F-suspect-stop fault detected by the router sensor, as well as all faults detected by the embedded sensors. The relay runs in the same process as the router sensor, and it uses the same callback protocol as the host relay.

OSPF relay. This relay uses OSPF's link-state routing protocol to communicate information about links. Under this protocol, routers generate information about their links in *Link-State Advertisements* (LSAs) and propagate LSAs to other routers using OSPF's flooding mechanism. For link failures (F-link), the OSPF relay uses normal LSAs, and for graceful shutdowns (F-router-reboot), the relay uses LSAs with infinite distance [92]. To announce overloaded links (F-link-util), the router relay uses *opaque LSAs* [10], which are LSAs that carry application-specific information.

Using the network to announce overload and failures might compound problems, so we rate-limit opaque LSAs to R_{opaque}, which our prototype sets to 1 per second (the maximum rate at which routers should accept LSAs [10]). Similarly, a buggy client could deplete the resources of the OSPF relay and the router relay, since these relays are shared; mitigating such behavior is outside our current prototype's scope, but standard techniques should apply (rate-limiting at the client, etc.). Note that the concern is buggy clients, not malicious ones, because Pigeon targets a single administrative domain (as noted in Chapter 1).

5.3.4 The interpreter

The interpreter gathers information about faults and outputs the failure conditions of §5.2.1. The interpreter must (1) determine which sensors correspond to the client-specified target process, (2) decide if a condition is implied by a fault, (3) estimate the condition's duration, (4) report the condition to the application via the client library, and (5) guarantee that stop conditions are never falsely reported. We discuss these duties in turn.

(1) The interpreter determines which sensors are relevant to a target process by using knowledge of the network topology, the location of sensors, and the location of the client and target processes.

(2) The interpreter must not report every fault as a condition; for example, a failed link that is not on the client's path to the target does not cause an unreachability condition. If the interpreter cannot determine the effect of a fault from failure information alone, it uses *hints*. For example, if a link becomes loaded along one of multiple paths to the target process, the interpreter sends an ICMP Echo Request with the Explicit Congestion Notification (ECN) option [90] set, to determine if the client's current path is affected.

The router sensors intercept these packets, and, if a link is loaded, mark them with the Congestion Encountered (CE) bits. If the interpreter receives an Echo Reply with these bits set, or times out after $T_{probe\text{-}to}$ time units, the interpreter reports an unreachability warning; in this warning, the network is marked as the critical resource (§5.2.1). Our implementation sets $T_{probe\text{-}to}$ to 50 ms.[5] The interpreter uses a similar hint (a network probe packet) to determine the effect of link failures.

The interpreter determines which paths are available to clients by passively participating in OSPF, a technique used in network monitoring [58, 98, 99]. For detecting link failures, this technique adds little overhead to the network; however, detecting link utilization has additional overhead (because it generates extra LSAs), and OSPF itself has some cost. We evaluate these costs in Section 5.4.3.

(3) As mentioned earlier, the interpreter estimates the duration of some unreachability conditions. Currently, these durations are hard-coded based on our testbed measurements, which we describe next; a better approach is to estimate duration using on-line statistical learning.

Our prototype estimates the duration of unreachability conditions as follows. If a link fails or a router reboots along the current path from the client to the target process, but there are alternate working paths, the interpreter reports a duration of $T_{new\text{-}path\text{-}delay}$—the average time that the network takes to find and install the new path. If a router reboots and there are no working paths from the client to the target process, the client must wait for the router to reboot, so the interpreter reports a duration of $T_{router\text{-}reboot}$—the average time that the router takes to reboot. The interpreter reports all other conditions as having an indefinite duration.

In our testbed, we set $T_{new\text{-}path\text{-}delay}$ and $T_{router\text{-}reboot}$ to 2.8 seconds and 66 seconds, respectively. We determine these values by measuring the unavailability caused by a fault, as observed by a host pinging another every 50 ms. In each experiment, we inject a link failure or router reboot and measure the failure's duration as the gap in ping replies observed by the end-host. We repeat this experiment 50 times for each fault. The means

[5]We validate this timeout by running an experiment where one host sends an ICMP Echo Request to another host for 10,000 iterations in a closed loop. We observe a response latency (which includes round-trip time and packet processing time) of 760 μs (standard deviation 96 μs) and a maximum of 1.2 ms, well below the timeout value.

Compared to existing failure reporting services, Pigeon improves, either in coverage, accuracy, timeliness, or quality	§5.4.1
Pigeon's richer information enables applications to react quickly or prevent costly recoveries	§5.4.2
Pigeon uses negligible CPU and moderate network bandwidth	§5.4.3

Figure 5.5: Summary of main evaluation results.

are as reported; the standard deviations are 27 ms and 2.5 seconds, respectively, for the two conditions.

(4) The interpreter reports all conditions (and their expected duration) to the client library; the interpreter also informs the client library if a condition clears or changes expected duration. The client library in turn calls back the client, and also exposes active conditions via the query() function (§5.2.1).

(5) To avoid reporting false stop conditions, the interpreter reports a stop only for the critical faults (F-exit, etc.), which sensors always confirm (by design).

5.4 Experimental evaluation

We evaluate Pigeon by assessing its reports (§5.4.1), its benefit to applications (§5.4.2), and its costs (§5.4.3). Figure 5.5 summarizes the results.

Fully assessing Pigeon's benefit would require running Pigeon against real-world failure data. We do not have that data, and gathering it would be a separate study [44]. Instead, we consider several real-world applications and failure scenarios, and show Pigeon's benefit for these instances.

Specifically, our evaluation compares our prototype to a set of baselines, in a test network, under synthetic faults. The three *baselines* in our experiments are:

1. End-to-end timeouts, set aggressively (200 ms timeout on a ping sent every 250 ms) and to more usual values (10 second timeout; ping every 5 seconds).
2. Falcon, with and without killing to confirm failure. We call the version without killing Falcon-NoKill.

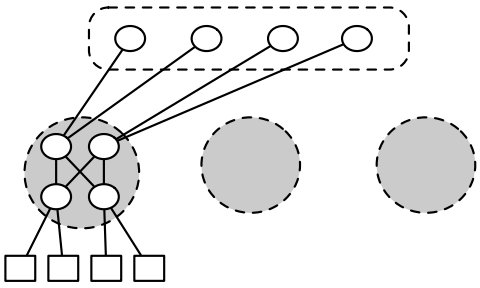

Figure 5.6: Illustration of the testbed used in Pigeon's evaluation.

3. A set of Linux system call: send() invoked every 250 ms, recv(), and epoll(), with and without error queues.

Our test network has 16 routers and 3 physical hosts, each multiplexing up to 4 virtual machines (VMs).[6] Our testbed appears in Figure 5.6; it comprises three *pods* (gray circles), consisting of four routers (white circles) and hosts (white squares). This is a *fat-tree* topology [3], which we use to model a data center. Note that our operating assumptions are data centers, fat-tree, and OSPF; these assumptions are compatible, as data centers use OSPF.[7] Our topology has the same scale as the one evaluated by Al-Fares et al. (minus one pod), albeit with different hardware [3].

Our routers are ASUS RT-N16s that run DD-WRT (basically Linux) [32], and use the Quagga networking suite [89] patched to detect link failure with BFD [59]. Our hypervisors run on three Dell PowerEdge T310s, each with a quad-core Intel Xeon 2.4 GHz processor, 4 GB of RAM, and ten Gigabit Ethernet ports (four of which are designated for VMs). The VMs are guests of QEMU v1.1 and the KVM extensions of the Linux 3.4.9-gentoo kernel. The guests run 64-bit Linux (2.6.34-gentoo-r6) and have either 768 MB of memory (labeled *small*) or 1536 MB of memory (*large*). Each VM attaches to the network using the host's Intel 82574L NIC, which it accesses via PCI passthrough.

[6]We do not expect much loss of fidelity in network performance from using VMs. The peak throughput achieved by a benchmark tool, netperf [79], is the same for a virtual and physical machine in our testbed, and in our experiments, VMs do not contend for physical resources.

[7]A non-assumption is using layer 3: there are data center architectures, based on fat-tree variants, that use OSPF at layer 2 [47].

what problem is modeled?	how is the fault injected?
process crash	segmentation fault
host reboot	issue reboot at host
link failure (backup paths exist)	disable router port
link failures (partition)	disable multiple router ports
router reboot (disrupts all paths)	issue reboot at edge router
network load	flood network path with burst
disk failure	change SMART attributes [103]

Figure 5.7: Panel of modeled faults. The three groups should generate stop, unreachability, and warning reports, respectively.

Figure 5.7 lists the panel of faults in our experiments. Although the *faults* are synthetic, the resulting *failures* model classes of actual problems.

5.4.1 How well does Pigeon do its job?

In this section, we first evaluate Pigeon's reports and then the effect of duration estimation error.

Multi-dimensional study. There are many competing requirements in failure reporting; the challenge is *not* to meet any one of them but rather to meet all of them. Thus, we perform a multi-dimensional study of Pigeon and the baselines.

Quantitatively, we investigate timeliness: for each pair of failure reporter and fault, we perform 10 runs in which a client process on a (small) VM monitors a target process on another (small) VM in the same pod. We record the detection time as the delay between when the apparatus issues an RPC (to fault injection modules on the routers and hosts) and when the client receives an error report; if no report is received within 30 seconds, we record "not covered". Qualitatively, we develop a rating system of failure reporting features: certainty, ability to give warnings, etc.

Figure 5.8 depicts the comparison. Pigeon's reports are generally of higher quality than those of the baselines; for instance, Falcon offers certainty, but it kills to do

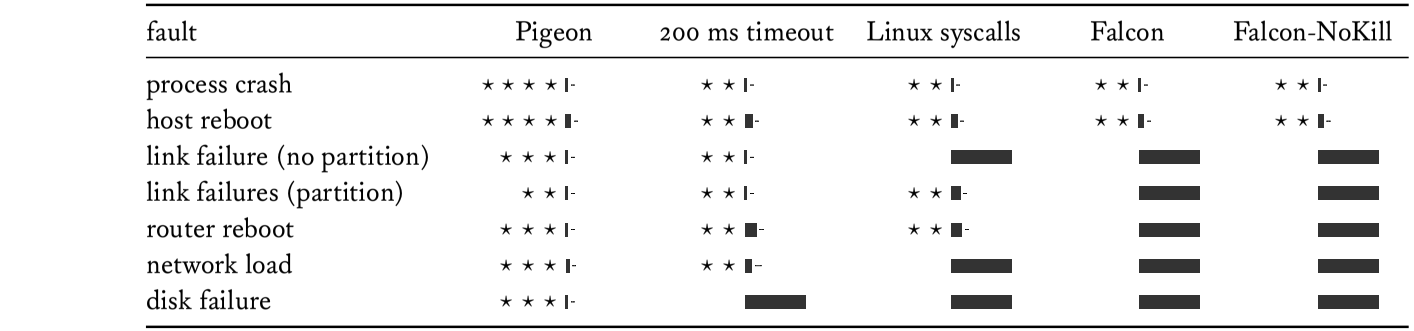

Figure 5.8: Pigeon compared to baseline failure reporters under our fault panel. **More stars and smaller bars are better.** Stars indicate the quality of a report; bars indicate the detection time. A maximum of four stars are awarded for detecting a failure, giving a certain report, giving more information than just crashed-or-not (e.g., indicating the cause as network load), and for not killing. Bar length and error bars depict mean detection time and standard deviation. These quantities are scaled; maximum is 30 seconds (long bars), which means "not covered". For the faults in our panel, Pigeon has higher quality, lower detection time, or both.

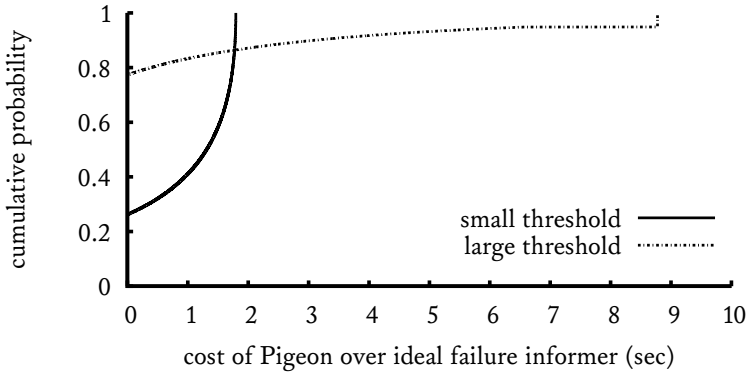

Figure 5.9: CDF of Pigeon's cost over the ideal failure informer for two sample applications, with availability thresholds (§5.2.3) smaller and larger than Pigeon's duration estimate.

so. And none of the baselines gives proactive warnings, as Pigeon does for the final two faults in the panel. In Section 5.4.2, we investigate how these qualitative differences translate into benefits for the application.

Pigeon's reports are timely. For process crashes, single link failure, partition, and router reboot, the mean detection times are 10 ms, 710 ms, 660 ms, and 690ms. For host reboots, Pigeon has a mean detection time of 1.9 seconds. (Detecting host reboot takes longer because we measure from when the reboot command is issued, and there is a delay before the reboot affects processes.)

Pigeon has full coverage in our experiments without needing the backstop end-to-end timeout. We also find that Pigeon never incorrectly reports a fault that has not occurred (a production deployment would presumably see some false reports and could adjust its parameters should such reports become problematic; see Section 5.3). Next, we consider the effect of duration estimation error in Pigeon's reports.

Duration estimation error. To understand the effect of duration estimation error, we compare our prototype to an *ideal failure informer* that predicts the exact duration of a failure condition. Specifically, we measure the additional unavailability that Pigeon causes in two applications: one that always recovers when using Pigeon because its

unavailability threshold (§5.2.3) is smaller than Pigeon's estimate (which is static; see Section 5.3.4), and one that always waits (because its threshold is higher).

We perform a simulation; we sample failure durations from a Weibull distribution (shape 0.5, scale 1.0), which is heavy-tailed and intended to stress the prototype's static estimate by "spreading out" the range of actual failures. For each sample, we record the *cost*, defined as the additional unavailability of the application when it uses Pigeon versus when it uses the ideal. We model the application's recovery duration and availability threshold as equal to each other.

Figure 5.9 depicts the results. For the small threshold, Pigeon matches the ideal for fewer than 30% of the samples because a significant fraction of the actual durations are very close to zero. Since this application always recovers with Pigeon, it frequently incurs (unnecessary) unavailability from recovery: waiting out these short failures would have resulted in less unavailability. For the large threshold, Pigeon matches the ideal for almost 80% of the samples but sometimes does much worse, since it waits on a long tail of failure durations. However, both applications' costs are capped, owing to their backstop timeouts.

5.4.2 Does Pigeon benefit applications?

We consider three case study applications that use Pigeon differently: RAMCloud [83], Cassandra [65], and lease-based replication [46]. For each, we consider the unmodified system, the system modified to use Pigeon, and the system modified to use one or more baselines.

RAMCloud [83]. RAMCloud is a storage system that stores data in DRAM at a set of *master servers*, which process client requests. RAMCloud replicates data on the disks of multiple *backup servers*, for durability. To reduce unavailability after a master server fails, a *coordinator* manages recovery to reconstruct data from the backups quickly. There are two notable aspects of RAMCloud for our purposes. First, although recovery is fast, it is expensive (it draws data from across the system, and it ejects the server, reducing capacity). Second, RAMCloud has an aggressive timeout: it detects failures by periodically pinging other servers at random and then timing out after 200 ms.

	RAMCloud using		
fault	timeout	Falcon	Pigeon
process crash	2.7s, eject	2.1s, eject	1.9s, eject
host reboot	2.6s, eject	1.8s, eject	1.9s, eject
link failure (no partition)	2.8s, eject	2.6s, wait	2.6s, wait
link failures (partition)	2.6s, eject	∞, wait	2.6s, eject
router reboot	2.6s, eject	∞, wait	1.7s, eject
network load	∞, eject	0.5s, wait	0.5s, wait

Figure 5.10: Mean unavailability observed by a RAMCloud client when RAMCloud uses different detection mechansims (standard deviations are within 15% of means). We also note whether RAMCloud ejects a server or waits for the fault to clear. Pigeon is roughly as timely as highly aggressive timeouts but can save RAMCloud the cost of recovery (specifically, under link failure (no partition) and network load faults). Falcon [70] hangs on network failures, so RAMCloud+Falcon does too (represented with ∞). Using timeouts, RAMCloud sometimes hangs if network load triggers multiple recoveries.

Thus, we expect that unmodified RAMCloud recovers more often than needed, and that Pigeon could help it begin recovery quickly or avoid recovering; we also expect that Pigeon can offer this benefit while providing full coverage and timely information. To investigate, we modify RAMCloud servers to use Pigeon and Falcon (with long backstop timeouts that do not fire in these experiments). We run a RAMCloud cluster on six large VMs (one client, five servers; two VMs in each pod), where each server stores 20 MB of data. This configuration allows RAMCloud to recover quickly on our testbed, at the cost of ejecting a server. For each injected fault, we perform 10 iterations and measure the gap in response time that is seen by a client querying in a closed loop.

Figure 5.10 depicts the results. Pigeon is roughly as timely as very aggressive timeouts, deriving its timeliness from sensors. Pigeon also enables RAMCloud to forgo recovery when possible. For instance, RAMCloud waits under network load when it receives a warning from Pigeon. Under a link failure, RAMCloud receives an unreachability condition with a short duration (equal to the network convergence time), so it waits. By contrast, under router reboot, RAMCloud receives an unreachability condition with a long duration (see Section 5.3.4), so it recovers.

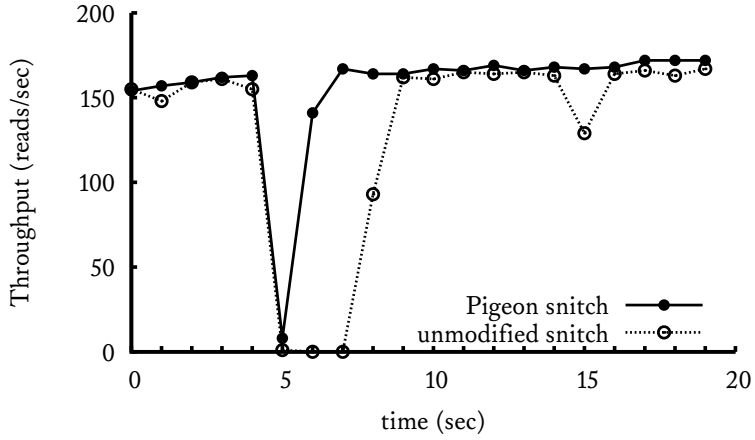

Figure 5.11: Cassandra's read throughput with and without Pigeon, after a network link fails 5 seconds into the run, temporarily disrupting a single server. Using Pigeon, the Cassandra snitch avoids using an unreachable replica; without Pigeon, Cassandra waits for the server to become reachable again. This example is representative: in our experiments, clients observed a mean unavailability of 1 second ($\sigma < 0.1$) using Pigeon and 2.2 seconds ($\sigma = 1.3$) using the unmodified snitch.

Cassandra [65]. Cassandra [65] is a distributed key-value storage system used widely (e.g., at Netflix, Cisco, and Reddit [20]). Cassandra servers read data from a primary replica and request *digests* from the other replicas. Thus, the choice of primary is important: if the primary has a problem, the server blocks until the problem is solved or the request times out. A server chooses as its primary the replica with the lowest expected request latency, as reported by an *endpoint snitch*.

We expect that Pigeon could help a snitch make better server selections. To measure this benefit, we run a client in a closed loop, inject two faults (network load and link failure) at a server in a five-server cluster (using large VMs), and measure the throughput.

Under network load (not depicted), the unmodified snitch and the Pigeon snitch offer comparable (and significant) benefit over no snitch, as the unmodified snitch's decisions are based on latencies—but only if the network is working. Figure 5.11 depicts the behavior in the case of link failure: here, Pigeon's report to the snitch allows the

server to quickly choose a better primary, resulting in higher throughput. Compared with RAMCloud: Pigeon lets Cassandra act more quickly than it otherwise would (because Pigeon reports the case and because switching is cheap), whereas this same report lets RAMCloud wait when it would otherwise act.

Lease-based replication [46]. A common approach to replication is to use a *lease server* [15, 46, 53], which grants a lease to a *master* replica, which in turn handles client requests, forwarding them to *backups*. If a backup detects or suspects a failure, it tries to become the master, by requesting a lease from the lease server. However, this process is delayed by the time remaining on the lease.

We expect that Pigeon's stop reports would be particularly useful here: they report that a lease holder has crashed with certainty, which allows the system to break the lease, increasing system availability.[8] To investigate, we build a demo replication application and lease server, which offers 10-second leases, and run it with and without Pigeon. We run a client (10 iterations) that issues queries in a closed loop, measuring the response gap seen by the client after we inject a process crash at the master.

The results are unsurprising (but encouraging): the response gap measured at the client averages 2.7 seconds (standard deviation 0.4 seconds) when using Pigeon, versus 6.1 seconds (standard deviation 2.5 seconds) using unmodified lease expiration.

Which applications do not gain from Pigeon? We considered simple designs for many applications; Pigeon usually provides a benefit but sometimes not. For example, a DNS client can use Pigeon to monitor its DNS server and quickly failover to a backup server when it encounters a problem. However, because the client's recovery is lightweight (retry the request), there is little benefit over using short end-to-end timeouts, since the cost of inaccuracy is low. Some applications do not make use of *any* information about failures; such applications likewise do not gain from Pigeon. For example, NFS (on Linux) has a *hard-mount mode*, in which the NFS client blocks until it can communicate with its NFS server; this NFS client does not expose failures or act on them. However, such applications are not our target since they consciously renounce availability.

[8] Note that Falcon would also enable such lease-breaking, but Falcon is incompatible with the availability requirement: if the problem is in the network, a query to Falcon literally hangs.

component (§5.3)	detecting network load	idle
CPU used at end-hosts		
process sensor/host relay	0.1%	0.0%
embedded sensor	3.0%	0.0%
interpreter	0.0%	0.0%
CPU used at routers		
router sensor/relay	0.2%	0.0%
OSPF sensor/relay	0.1%	0.0%
bandwidth used		
at each end-host	2.3 kbps	0 bps
at each router	3.4 kbps	1.3 kbps

Figure 5.12: Resource overheads of our Pigeon implementation.

5.4.3 What are Pigeon's costs?

Implementation costs. Pigeon has 5425 lines of C++ and Java. Sensors are compact, and the system is easy to extend (e.g., the disk failure logic required only 34 lines). Integrating Pigeon into applications is easy: it required 68 lines for RAMCloud and 414 lines for Cassandra.

CPU and network overheads. Figure 5.12 shows the resource costs of Pigeon. CPU use is small; the main cost is a high-priority process in the embedded sensor, which periodically increments a shared counter (§5.3.2). Pigeon's network overheads come from OSPF LSAs to hosts.

Scalability. The main limiting factor is bandwidth to propagate failure data; this overhead is inherited from OSPF, which generates a number of LSAs proportional to the number of router-to-router links in the network. This overhead is reasonable for networks with thousands of routers and tens of thousands of hosts. Specifically, we estimate that in a 48-port fat-tree topology with 2880 routers and 27,648 end-hosts [3], OSPF would use less than 11.8 Mbps of bisection bandwidth (or 1.1% of 1 Gbps capacity), which is consistent with our smaller-scale measurements. Larger networks would presumably use multiple areas; we briefly discuss extending Pigeon to that setting in the next section.

5.5 Discussion

We now consider assumptions and limitations of the failure informer abstraction (§5.2.1–§5.2.2), the Pigeon architecture (§5.2.4), and our prototype implementation (§5.3).

The abstraction. As with any abstraction, this one is based on generalizing from specific difficult cases, on judgment, and on use cases. It is hard to prove that an abstraction is optimal (but ours is better than at least our own previous attempts). A critique is that an implementation of the abstraction is permitted to return spurious "uncertain" reports. However, uncertainty is fundamental and hence some wrong answers are inevitable (§5.1); thus, this critique is really a requirement that the implementation have few false positives (§5.4.1).

The architecture. Our architecture assumes a single administrative domain. This scenario has value because many data centers satisfy this assumption, but extending to a federated context may be worthwhile. However, this requires additional research; prior work gives a starting point [5, 6, 8, 101, 117].

One benefit of Pigeon's architecture is that it can be shared across many different applications, with different purposes. For example, Pigeon could be integrated into an existing group communication service [13, 23], a configuration management service [15, 53], or even a new failure reporting service that uses majority-based techniques to kill when Pigeon reports an unreachability.

The prototype. Our prototype assumes OSPF, runs on layer 3, and monitors only end-hosts and routers (not middleboxes). We designed Pigeon for extensibility, so expanding it to other routing protocols would require implementing appropriate relays and sensors (§5.3.2–§5.3.3). We could also extend to layer-2 networks, either with OSPF (some layer-2 architectures run OSPF for routing [47]), or without; in the latter case, the prototype would need different sensors and relays. Another extension is to monitor middleboxes using additional types of sensors. Neither our current prototype nor these extensions requires structural network changes. (The logic for sensors and relays is small and runs in software, on a router's or switch's control processor.)

We estimated our prototype's scalability in Section 5.4.3: it ought to scale to tens of thousands of hosts in a single area, with the limit coming from OSPF itself. OSPF can scale to more hosts, by using multiple areas; we could extend Pigeon to this case using additional sensors and relays at area borders to address what would otherwise be a loss of accuracy (since areas are opaque to each other). We leave this for future work.

Chapter 6

Summary & Outlook

6.1 Revisiting the three challenges

In Chapter 1, we presented three challenges in building a fast failure reporting service that uses inside information: systematically collecting inside information, defining an interface for reporting failures, and limiting negative impact. We now examine the trade-offs in the choices made by the three failure reporting services described in this dissertation, Falcon (Chapter 3), Albatross (Chapter 4), and Pigeon (Chapter 5).

Systematically collecting inside information. Each failure reporting service gathers inside information by periodically checking components locally, or by re-purposing existing monitoring infrastructure. The services differ in how they get this information back to clients. Falcon exclusively uses callbacks to communicate inside information, while Albatross sometimes relies on network-level broadcast when there is a host failure or network partition. Pigeon uses a network of relays to communicate inside information; some of these relays communicate among themselves, while others call back directly to Pigeon's interpreter.

The reason for these differences is in the scope of monitored components. Falcon monitors components located at end-hosts, whereas Albatross and Pigeon additionally monitor the health of the network. Because every client of Albatross and Pigeon monitors the network, using Falcon's register-callback architecture would require state

at either the manager (in Albatross) or at each of the routers (in Pigeon) for every client. Thus, Albatross uses broadcast (so the manager need not track active clients) and Pigeon piggybacks on OSPF (to avoid having every client monitor every router directly).

Defining an interface for reporting failures. Our work on this dissertation began in search of a *perfect failure detector* [21], one that eventually detects every failure and guarantees that a "down" report means that a process has crashed. In Falcon, we settled for a service with a reliable failure detector interface, where "down" reports have the same guarantee (though Falcon may have caused the crash) and failures are always reported—when there is network connectivity. Because of this choice, Falcon has two disadvantages: Falcon itself sometimes causes crashes, and Falcon cannot handle network failures. To keep the benefits of the reliable failure detector interface while addressing these disadvantages, we designed Albatross. Instead of killing components, Albatross blocks processes from using the network, which softens the blow of interference. This choice also allows Albatross to provide asymmetric guarantees about disconnection despite common network partitions. Pigeon forgoes making all failure reports reliable thereby allowing it to renounce interference; Pigeon instead exposes to applications its certainty about a problem. This choice allows applications to make qualitatively different recovery choices, which can reduce the unavailability caused by failures. There may be other, better ways of exposing failure information, but our experience shows that these interfaces benefit applications.

Limiting negative impact. All three services limit their overheads by polling locally at relatively low rates (tens of checks per second), and by sharing this work among all clients. The services differ in how they handle uncertain information. Falcon has the greatest negative impact because it kills, and, although Falcon aims to kill surgically, it sometimes kills at the granularity of machines. Albatross lowers this burden by disconnecting processes at the network level but pays for this reduction by consuming scarce resources in switch memory. Pigeon eliminates the impact of killing entirely.

6.2 Choosing from the aviary

Two factors determine which bird is the best choice: (1) the client applications, and (2) the target environment. We examine these considerations in turn.

The failure reporting service's interface determines how easily it can be used by existing applications. Client applications can most readily use Falcon because it has a familiar failure detector interface that is already used by many applications (e.g., Facebook's Cassandra [19] or LinkedIn's Voldemort [109]). Albatross also has a familiar binary interface, and can be used in-place of an existing failure detector—assuming the application developer understands Albatross's asymmetric guarantees. Pigeon's failure informer interface makes it the most difficult to integrate with applications (though it was not overly burdensome; see Chapter 5, Section 5.4) since developers need to understand the difference between a stop condition and an unreachability condition, as well as how to determine an "unavailability threshold." Despite these difficulties, Pigeon offers more benefit than Falcon or Albatross from inside information since Pigeon can report intermittent conditions (like temporary load).

All three services require the ability to modify end-host software, and they each require different modifications to the network. To detect end-host failures, Falcon requires the ability to run arbitrary software on top of rack switches, though recent work (e.g., Sidecar [100]) could relax this requirement. In contrast, Albatross requires an abstract programmable interface from the network so that its manager can detect, enforce, and report partitions. Because Pigeon gathers more information from the network than either Falcon or Albatross, it requires the ability to run arbitrary code on all network routers (to detect congestion) and the ability to snoop on network protocols (to detect link failures).

6.3 Next steps

A fast failure reporting service lets distributed systems respond more quickly to failure, and the quality of its reports can help the system to make better recovery decisions.

Nevertheless, several steps remain before any of these birds can fly free and be used in production systems.

First, a small scale real-world deployment is needed to measure the presence and impact of false positives in local monitoring. Such a deployment would help to validate the approach of using inside information to detect failures but would require access to production data centers (beyond what is usually granted to academic research).

Second, distributed systems that run on data center networks are beginning to span multiple data centers, bringing concerns of handling additional delays and failures of wide-area networks. In particular, Albatross's current approach would excluded one data center from another in order to provide Albatross's asymmetric guarantees; using Albatross to handle wide-area problems would likely involve extending its guarantees to differentiate between small partitions (its current assumption) and large partitions.

Finally, each of these failure reporting services exposes an interface that is not commonly assumed in distributed systems; this presents a challenge and an opportunity. The challenge is incorporating these services into existing code, either by factoring out an application's failure detection logic or replacing it in-line. The opportunity is using both sub-second detection time and better information about failures to develop new recovery strategies for improving fault tolerance.

6.4 Conclusion

One take-away from this dissertation is that using inside information can help distributed systems better respond to failures. However, this take-away is an unsurprising instantiation of a general rule in systems design: lower-level interfaces often perform better than higher-level interfaces—but with more difficulty in their use. The more important take-aways from this dissertation are the considerations in using inside information, such as the costs of interference and the granularity at which inside information is presented to applications. Thus, we hope that the lasting contribution of this dissertation is motivating infrastructure providers (data centers, cloud services, etc.) to implement fast failure reporting.

Bibliography

[1] M. K. Aguilera, G. L. Lann, and S. Toueg. On the impact of fast failure detectors on real-time fault-tolerant systems. In *International Conference on Distributed Computing (DISC)*, pages 354–370, Oct. 2002.

[2] M. K. Aguilera and M. Walfish. No time for asynchrony. In *Workshop on Hot Topics in Operating Systems (HotOS)*, May 2009.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM*, pages 63–74, Aug. 2008.

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

[6] K. Argyraki, P. Maniatis, and A. Singla. Verifiable network-performance measurements. In *ACM Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, Dec. 2010.

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

[8] B. Barak, S. Goldberg, and D. Xiao. Protocols and lower bounds for failure localization in the Internet. In *EUROCRYPT*, Apr. 2008.

[9] 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.

[10] L. Berger, I. Bryskin, A. Zinin, and R. Coltun. The OSPF opaque LSA option. RFC 5250, Network Working Group, July 2008.

[11] 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.

[12] K. P. Birman. Replication and fault-tolerance in the ISIS system. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 79–86, Dec. 1985.

[13] 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.

[14] British Association for the Advancement of Science. *Laughlab: The Scientific Search for the World's Funniest Joke (Humour)*. Arrow Books, 2002.

[15] 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.

[16] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive microreboots: A soft-state system case study. *Performance Evaluation Journal*, 56(1–4):213–248, Mar. 2004.

[17] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot—a technique for cheap recovery. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–44, Dec. 2004.

[18] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol (SNMP). RFC 1157, Network Working Group, 1990.

[19] The Apache Cassandra project. `http://wiki.apache.org/cassandra/ArchitectureInternals#Failure_detection`.

[20] The Apache Cassandra Project. `http://cassandra.apache.org/`.

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

[22] 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.

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

[24] B. Chun, J. M. Hellerstein, R. Huebsch, P. Maniatis, and T. Roscoe. Design considerations for Information Planes. In *Workshop on Real, Large, Distributed Systems (WORLDS)*, Dec. 2004.

[25] 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.

[26] D. D. Clark. The structuring of systems using upcalls. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 171–180, Dec. 1985.

[27] D. D. Clark. The design philosophy of the DARPA Internet protocols. In *ACM SIGCOMM*, pages 106–114, Aug. 1988.

[28] 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.

[29] 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.

[30] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 161–174, Apr. 2008.

[31] 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.

[32] DD-WRT firmware. `http://www.dd-wrt.com`.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

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

[40] 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.

[41] Forrester. eCommerce website performance today. `http://www.edataworld.com/files/pdf/ecommerce_website_perf_wp.pdf`, 2009.

[42] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–43, Oct. 2003.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *ACM SIGCOMM*, pages 51–62, Aug. 2009.

[48] 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.

[49] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary Internet end hosts. In *SIGCOMM Workshop on Internet Measurement (IMW)*, pages 5–18, Nov. 2002.

[50] G. Hamerly and C. Elkan. Bayesian approaches to failure prediction for disk drives. In *International Conference on Machine Learning (ICML)*, pages 202–209, June 2001.

[51] 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.

[52] 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.

[53] 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.

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

[55] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, pages 59–72, Mar. 2007.

[56] 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.

[57] 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.

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

[59] D. Katz and D. Ward. Bidirectional forwarding detection (BFD) for IPv4 and IPv6 (single hop). RFC 5881, Network Working Group, June 2010.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] R. Krishnan, J. P. G. Sterbenz, W. M. Eddy, C. Partridge, and M. Allman. Explicit transport error notification (ETEN) for error-prone wireless and satellite networks. *Computer Networks*, 46(3):343–362, 2004.

[65] A. Lakshman and P. Malik. Cassandra – A decentralized structured storage system. In *International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, Oct. 2009.

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

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

[68] 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.

[69] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Taming uncertainty in distributed systems with help from the network. In *European Conference on Computer Systems (EuroSys)*, Apr. 2015.

[70] 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.

[71] libvirt: The virtualization API. `http://libvirt.org/`.

[72] DomUClusters – Linux-HA. `linux-ha.org/wiki/DomUClusters`.

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

[74] Linux kernel dump test module. `http://kernel.org/doc/Documentation/fault-injection/provoke-crashes.txt`.

[75] 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.

[76] 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.

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

[78] J. Moy. OSPF version 2. RFC 2328, Network Working Group, Apr. 1998.

[79] Netperf, the network performance benchmark. `www.netperf.org`.

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

[81] 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.

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

[83] 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.

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

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

[86] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 17–28, Feb. 2007.

[87] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Symposium on Networked Systems Design and Implementation (NSDI)*, May 2015.

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

[89] The Quagga routing software suite. `http://www.nongnu.org/quagga/`.

[90] K. K. Ramakrishnan, S. Floyd, and D. Black. The addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, Network Working Group, Sept. 2001.

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

[92] A. Retana, L. Nguyen, R. White, A. Zinin, and D. McPherson. OSPF stub router advertisement. RFC 3137, Network Working Group.

[93] A. M. Ricciardi and K. P. Birman. Using process groups to implement failure detection in asynchronous environments. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 341–353, Aug. 1991.

[94] T. Roscoe, R. Mortier, P. Jardetzky, and S. Hand. InfoSpect: Using a logic language for system health monitoring in distributed systems. In *ACM SIGOPS European Workshop*, pages 31–37, Sept. 2002.

[95] N. Schiper and S. Toueg. A robust and lightweight stable leader election service for dynamic systems. In *International Conference on Dependable Systems and Networks (DSN)*, pages 207–216, June 2008.

[96] N. Schiper, S. Toueg, and D. Ivan. Leader elector source code. `https://github.com/nschiper/LeaderElection`.

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

[98] A. Shaikh, M. Goyal, A. Greenberg, R. Rajan, and K. K. Ramakrishnan. An OSPF topology server: design and evaluation. *IEEE JSAC*, 20(4):746–755, May 2002.

[99] 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.

[100] A. Shieh, S. Kandula, and E. G. Sirer. SideCar: building programmable datacenter networks without programmable switches. In *ACM Workshop on Hot Topics in Networks (HotNets)*, Oct. 2010.

[101] 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.

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

[103] C. E. Stevens. AT attachment 8 - ATA/ATAPI command set. Technical Report 1699, Technical Committee T13, Sept. 2008.

[104] J. Stone and C. Partridge. When the CRC and TCP checksum disagree. In *ACM SIGCOMM*, pages 309–319, Aug. 2000.

[105] 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.

[106] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *International Middleware Conference (Middleware)*, pages 55–70, Sept. 1998.

[107] 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.

[108] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *European Conference on Computer Systems (EuroSys)*, Apr. 2015.

[109] Project Voldemort: a distributed database. `http://www.project-voldemort.com/`.

[110] 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.

[111] D. A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`.

[112] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: A lightweight network location service without virtual coordinates. In *ACM SIGCOMM*, Aug. 2005.

[113] 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.

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

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

[116] 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.

[117] X. Zhang, A. Jain, and A. Perrig. Packet-dropping adversary identification for data plane security. In *ACM Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, Dec. 2008.

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

[119] GSoC 2010: ZooKeeper Failure Detector model. `http://wiki.apache.org/hadoop/ZooKeeper/GSoCFailureDetector`.

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