Improving availability in distributed systems with failure informers

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

*The University of Texas at Austin †Microsoft Research Silicon Valley

Abstract. This paper addresses a core question in distributed systems: how should applications be notified of failures? When a distributed system acts on failure reports, the system's correctness and availability depend on the granularity and semantics of those reports. The system's availability also depends on coverage (failures are reported), accuracy (reports are justified), and timeliness (reports come quickly). This paper describes *Pigeon*, a failure reporting service designed to enable high availability in the applications that use it. Pigeon exposes a new abstraction, called a *failure informer*, which allows applications to take informed, application-specific recovery actions, and which encapsulates uncertainty, allowing applications to proceed safely in the presence of doubt. Pigeon also significantly improves over the previous state of the art in the three-way trade-off among coverage, accuracy, and timeliness.

1 Introduction

Availability is now a paramount concern of distributed applications in data centers and enterprises (distributed storage systems, key-value stores, replication systems, etc.); for such applications, even seconds of downtime can affect millions of users. A critical factor in availability is failure handling. Specifically, for optimal availability, distributed applications need to learn of failures quickly, so that they can recover, and they need information *about* the failure, so that they can take the best recovery action.[1]

This paper proposes *Pigeon*, a service for reporting host and network failures to highly available distributed applications. Pigeon provides a new abstraction, called a *failure informer*. This abstraction hides the messy details of failures; it reports a small number of conditions that each represent a class of problems that affect the application similarly. The conditions are differentiated by the failure certainty, or lack thereof, which gives enough information for applications to improve their recovery, in application-specific ways.

For example, if a lease server [13, 30] is informed of the certain crash of a process holding a lease, the server can bypass the lease delay and reissue the lease immediately; without this information, the lease server would have to wait until the lease times out. As another example, consider a primary-backup system [4]. If Pigeon reports to the backup that the primary has certainly stopped, the backup takes over immediately; if Pigeon reports that the primary is (possibly intermittently) unreachable, the backup must decide whether to fail over the primary, based on the expected problem duration (which Pigeon reports) and the cost of failover; and if Pigeon reports that the primary is expected to crash soon, the backup can provision a new replica without failing over the primary yet.

In the above example, notice that the different reports from Pigeon are qualitatively different and allow qualitatively different failure responses. Consider, by contrast, existing mechanisms for reporting failures, such as ICMP, TCP connection reset, and failure detectors [17] built on tuned timeouts [11, 18, 34, 62] or on layer-specific monitors [46]. These mechanisms not only cannot distinguish between various failure conditions but also have other shortcomings (as argued in Section 2.1). These shortcomings are rooted in the network's design:

> [At] the top of transport, there is only one failure, and it is total partition. The architecture was to mask completely any transient failure. ... the Internet makes very weak assumptions about the ability of a network to report that it has failed. [The] Internet is thus forced to detect network failures using Internet level mechanisms, with the potential for a *slower and less specific error detection* [emphasis added] [21].

The rationale was simplicity. Since the network was to be designed for survivability above almost everything else [21, §3–§4], and hence would recover from failures, the benefit of exposing failures to applications was not worth the cost of a mechanism. Yet, availability of distributed applications—a more pressing concern now than it was then—calls for additional design: we want *faster and more specific error detection*!

What should such a failure reporting service look like? Answering this question requires addressing several challenges. First, there are many failure indicators (e.g., monitors reporting crashed processes, status of network links, hardware error status), each with its own idiosyncrasies, but what details should be exposed to applications? Second, these indicators may report uncertain information, leading to wrong conclusions. Addressing these two challenges requires finding the right abstraction for failure reporting—one that is simple but conveys

[1] By *failure*, we mean a problem that is visible end-to-end, not masked; by *recovery*, we mean actions in response to such failures (failover, etc.). Techniques such as microreboot and component restart [14, 15] are failure *prevention*, which is orthogonal to (in the sense that it does not obviate) our concern of failure *reporting*.

the information that lets applications recover effectively. The third challenge is in implementing the abstraction: to improve application availability, the implementation must provide full *coverage* (failures are reported), but also provide *accuracy* (reports are justified), and *timeliness* (failures are reported quickly). Meanwhile, these considerations are in a three-way trade-off.

Our response, Pigeon, classifies failures into four types: whether the problem certainly occurred versus whether it is expected and imminent, and whether the target is certainly and permanently stopped versus not. Observe that a report of certain occurrence and certain permanence abstracts "process crash" (among other things), and a report of certain occurrence and uncertain permanence abstracts "pending timeout expired" or "network partition" (among other things). Furthermore, observe that applications can benefit from even the uncertain reports: they can consider the cost-benefit trade-offs of waiting versus recovery (for problems of uncertain permanence) and of waiting versus precautionary actions (for problems of uncertain occurrence). Pigeon includes other information too, such as expected problem duration, and the resulting abstraction is what we refer to as a failure informer. To summarize the abstraction, it knows what it knows, it knows what it doesn't know, and applications benefit from hearing the difference.

Pigeon manages the conflict among coverage, accuracy, and timeliness by relying on an end-to-end timeout as a backstop (achieving full coverage) and then using low-level information from throughout the system to significantly improve the accuracy-timeliness tradeoff. The use of low-level information is inspired by Falcon [46]. However, Falcon has limited coverage (network failures cause it to hang), a coarse interface (it reports crashes only), and adverse collateral effects (it kills components, sometimes gratuitously). We elaborate on these points in Section 2.1 and compare the two systems in Section 7.

Our implementation of Pigeon has several limitations and operating assumptions. First, Pigeon assumes a single administrative domain (but there are many such networks, including enterprise networks and data centers). Second, Pigeon requires the ability to install code in the application and network routers (but doing so is viable in single administrative domains). Third, for Pigeon to be most effective, the administrator or operator must perform environment-specific tuning (but this needs to be done only once).

Before continuing, we emphasize that the challenges of Pigeon are mostly in architecture and design, as opposed to low-level mechanism; the mechanisms in Pigeon are largely borrowed from previous work [36, 37, 46, 59, 60]. The contributions of this work are:

- The thesis that network and host failures should be exposed to applications (§2). Though simple, this thesis has apparently not been advanced in previous work (§7).
- The failure informer abstraction for exposing failures (§3.1–§3.2) and a service, Pigeon, that implements it (§3.4–§3.5). As is often the case with concise but powerful abstractions, this one may appear "easy", yet identifying it was not, judging by our own repeated attempts.
- The uses of Pigeon (§3.3, §5.2). Our confidence in the abstraction is bolstered by concrete use cases.
- The evaluation (§5) of our prototype (§4). For a minor price in resources, Pigeon quickly (sub-second time) and accurately reports common failure types. Pigeon quantitatively and qualitatively outperforms other mechanisms (including Falcon), and we demonstrate that it allows real applications to make better, faster, application-specific recovery decisions.

2 Motivation, challenges, and principles

We now explain the status quo's shortcomings (§2.1) and the principles that Pigeon is based on (§2.2).

2.1 Failure reporting today

Existing mechanisms for reporting failures are coarse-grained, lack coverage, lack accuracy, or do not handle latent failures. We give specifics below and demonstrate some of them experimentally in Section 5.1.

As an example of a coarse-grained mechanism, consider ICMP "destination unreachable" messages, which the network delivers to sources [54]. This signal conflates different failure cases (whether the failure resulted from a problem in the host or network, whether the condition is transient, etc.), requiring that applications react to each failure identically or ignore the notifications altogether.

Other mechanisms do not have good coverage. For example, consider the "connection reset" error in TCP. This signal reports to the application that a remote process has exited—but only if the remote TCP stack and the network are both working.

Other mechanisms have good coverage but lack accuracy. For example, end-to-end timeouts eventually trigger if a failure occurs, but they sometimes trigger prematurely, without any failures.

Some mechanisms do not detect *latent* failures: they report failure only if and when the application tries to use the network. For example, the network generates an ICMP error packet only when a host attempts to send data.[2] As another example, consider timeouts again: they are often set *on* some pending event (e.g., a request issued to a peer). If an application has no such event out-

[2]The `select()` and `epoll()` system calls, which report errors on particular file descriptors, are simply interfaces to this behavior.

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

standing but later generates one, it must then wait for the timeout interval to expire before learning of the failure.

Falcon [46] detects latent failures and is accurate, but it sacrifices coverage and gives coarse-grained reports. Falcon monitors a remote process with a network of spies deployed at different layers of the system (operating system, application, etc.). If a layer is unresponsive, a spy sometimes kills that layer (e.g., by terminating a virtual machine) so that clients can make progress; this requires network communication so that the Falcon client can request and confirm the kill. As a result, Falcon hangs if there is a network failure. Moreover, Falcon can report applications only as crashed or not crashed.

2.2 Design challenges and principles

As noted in the introduction, there are three top-level challenges in designing Pigeon: identifying what failure details to provide; handling uncertain information safely; and managing a three-way trade-off among coverage, accuracy, and timeliness. At a high level, the root cause of these challenges is the difficulty of determining why a remote process does not respond: is it crashed? or slow? or is the problem in the network? We confront these challenges with the principles below.

Renounce killing. Consider techniques that provide perfect accuracy, such as Falcon [46], watchdogs [1, 27], and virtual synchrony [12]. What would be required for them *not* to hang on network failures? Since their accuracy comes from killing (when they are uncertain), they would have to kill network elements and intentionally create network partitions. This seems like a bad idea. In fact, even targeted killing is not ideal: taking live components offline impairs availability! Pigeon shall not kill.

Provide full coverage. Availability requires that the failure informer report all failures (full coverage). However, two issues result. First, full coverage implies that perfect accuracy is unattainable: if an informer must report all failures (and do so without killing), but is uncertain about whether a failure occurred, then the informer will sometimes report some failures incorrectly. Second, the three-way conflict among coverage, accuracy, and timeliness means that full coverage causes a trade-off between accuracy (already imperfect) and timeliness. Our next two principles address these issues in turn.

Expose uncertainty. How can the failure informer ensure safety, despite occasional mistakes? Our approach is for the failure informer to provide certainty when possible and to flag the reports that may be wrong as uncertain. (This is different from the notion of confidence in failure detectors [34]; see Section 7.) This allows applications to take qualitatively different recovery actions, as stated in the introduction (see also Section 5.2). Note that handling uncertainty is not a burden, as applications do so already when, for example, end-to-end timeouts expire.

Leverage local information. The timeliness-accuracy tradeoff can be improved by local knowledge that reveals the state of components. For example, if a host's cable disconnects from a network switch, the switch quickly learns, and the informer can thus tell the application quickly. For the same accuracy, then, a failure informer with access to lower layers can be more timely, because the local information is visible sooner than if it had to bubble up to higher layers. We borrow the idea of using local information from Falcon [46] (see Section 7).

Design for extensibility. We are not going to get a perfect implementation, so we design for extensibility: Pigeon accommodates add-on modules that provide better information and indicate different kinds of faults, ideally improving the accuracy-timeliness trade-off. 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.

3 Design of Pigeon

This section presents the interface exposed by Pigeon (§3.1), describes the guarantees (§3.2), explains how Pigeon can be used (§3.3), describes its architecture (§3.4), and explains errors and their effects (§3.5).

3.1 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 1.

(1) In a *stop*, the target process has stopped executing and lost its volatile state. The problem has already

occurred, and it is certainly permanent. This condition abstracts process crashes, machine reboots, etc.

(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 occurs it is permanent. This condition abstracts cases such as a report about an imminent disk failure [33, 53, 63].

(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: whether the problem is certainly permanent (stop vs. unreachability) and whether the problem certainly occurred (actual vs. warning).

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.[3]) We describe how this property is set in Section 4.4. A property of the 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 see the following programmatic interface.

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
setTimeout(h, timeout)	set/reset timeout
clearTimeout(h)	cancel timeout

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 to be used in other functions. The init() function takes as a parameter a callback function, which the implementation calls as new failure conditions emerge.

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

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

The setTimeout() and clearTimeout() functions set/reset and clear end-to-end timeouts. Clients use timeouts as a catch-all: after the client installs a timer, if the client does not cancel or reset it before the timeout period, then the interface reports an unreachability.

3.2 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 the network and Pigeon itself, as described in Section 3.5.

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

Accuracy. By accuracy, we mean "reported failures are justified" (§1); we address the correctness of duration estimates in Section 5.1. 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 (a 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.

3.3 Using the interface

We now give a general description of how applications might use Pigeon; Section 5.2 considers specific applications (RAMCloud [52], Cassandra [43], lease-based replication [30]). 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 a quantitative implication and a qualitative one. 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 [13], ZooKeeper [35], or Paxos [45], 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 must 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.
- 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.

3.4 Architecture of Pigeon

As stated in the introduction, Pigeon works within a single administrative domain: an enterprise, a data center, a campus network, etc. Pigeon's architecture is geared toward extracting and exploiting the information about failures already available inside the system. For example, the failed links in a network collectively yield information about a network partition. To use this 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 2). We describe their abstract function below and their instantiations in our prototype in Section 4.

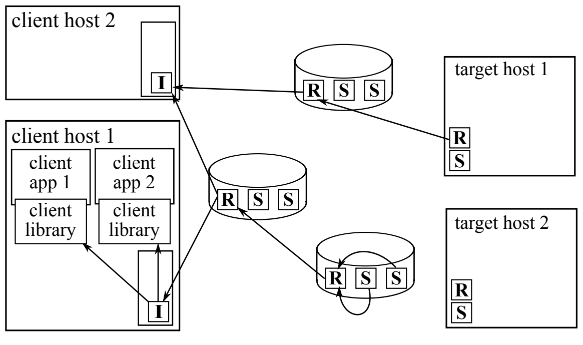


Figure 2—Architecture of Pigeon. Pigeon has sensors (S), relays (R), and interpreters (I). Sensors are component-specific. Sensors and relays are shared by multiple clients and end-hosts; an interpreter is shared by all client applications on its host. The client library presents the client API (§3.1) to applications.

A *sensor* is component-specific and tailored; it is embedded in the component and detects faults in it. A *fault* is a local event, possibly a malfunction, that may contribute to one of the four failure conditions (§3.1). 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 (§3.1). Interpreters also determine which sensors are relevant to the client-supplied (IP, port) pair that identifies a target (§4.4).

3.5 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 3 summarizes the effect of errors.

Before continuing, we note *non*-effects. First, Pigeon does not compromise on coverage: its coverage derives from the end-to-end timeout, which is implemented in

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 3—Effect of errors on Pigeon's guarantees. Errors in duration estimates are covered in Section 5.1.

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; while *inaccuracy* is possible under Pigeon, the only threat to *safety* is a report of a stop that did not happen (§3.2), which Pigeon is designed to avoid (§4).

If a sensor, relay, or interpreter crashes or is disconnected from the network, Pigeon loses access to local information, which affects accuracy and timeliness (§2.2). Loss of local information also causes missed opportunities to report some failures as stop conditions (e.g., remote process exit) rather than an unreachability condition triggered by the end-to-end timeout.

If a sensor does not detect a fault, then Pigeon may need to rely on the end-to-end timeout, 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 (see above). 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 (see above). Errors in the interpreter's duration estimates are covered in Section 5.1.

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

4 Prototype of Pigeon

We describe our target environment (§4.1), and the implementations of sensors (§4.2), relays (§4.3), and the interpreter (§4.4) 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).

4.1 Target environment

Our prototype targets networks that use link-state routing protocols, which are common in data centers and enterprises [31, 39]. Currently, the prototype assumes the Open Shortest Path First (OSPF) protocol [51] with a single OSPF *area* or routing zone. This assumption may raise scalability questions, which we address in Section 5.3. We discuss multi-area routing and layer 2 networks in Section 6.

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.

4.2 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 as F-$\langle$type$\rangle$ (critical ones 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 [46], 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. This fault can be due, for example, to 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 Falcon [46, §4]).

F-disk-vulnerable. A disk used by the target process has failed or is vulnerable to failure (based on vendor-specific reporting data, e.g., SMART [63]). 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 (the 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. 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 runs as a sensor that detects two faults:

F-suspect-stop. The end-host is no longer responding to network probes. This fault can occur, for example, 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) and T_{util} to 1 second (which corresponds to the maximum rate at which this fault can be reported; see Section 4.3).

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) [38].

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.

4.3 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 3.5.

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 elsewhere [20, 36, 46].

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 [57]. 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 highest rate at which routers should accept LSAs [10]). Similarly, a buggy client could deplete the resources of this relay (and the router relay), since they are shared; mitigating such behavior is outside our current prototype's scope, but standard techniques should apply (rate-limiting, etc.). Note that the concern is buggy clients, not malicious ones, because Pigeon targets a single administrative domain (§4.1).

4.4 The interpreter

The interpreter gathers information about faults and outputs the failure conditions of §3.1. The interpreter must (1) determine which sensors correspond to the client-specified target process, (2) determine 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) never falsely report a stop condition. 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 [56] 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 (§3.1). Our implementation sets $T_{probe\text{-}to}$ to 50 ms.[4] 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 elsewhere [36, 59, 60]. 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 5.3.

(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 report 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 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 (§3.1).

(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).

Compared to existing failure reporting services, Pigeon improves, either in coverage, accuracy, timeliness, or quality	§5.1
Pigeon's richer information enables applications to react quickly or prevent costly recoveries	§5.2
Pigeon uses negligible CPU and moderate network bandwidth	§5.3

Figure 4—Summary of main evaluation results.

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 [63]

Figure 5—Panel of modeled faults. The three groups should generate stop, unreachability, and warning reports, respectively.

5 Experimental evaluation

We evaluate Pigeon by assessing its reports (§5.1), its benefit to applications (§5.2), and its costs (§5.3). Figure 4 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 paper in its own right [28]. 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.
3. A set of Linux system calls (§2.1): `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).[5] Our

[4] 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.

[5] We do not expect much loss of fidelity in network performance from using VMs. The peak throughput achieved by a benchmark tool, net-

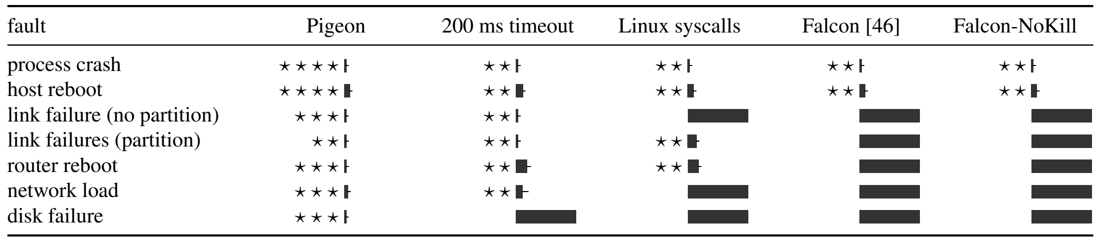
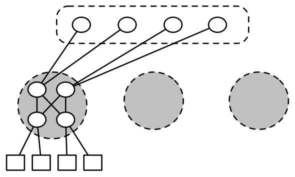


Figure 6—Pigeon compared to baseline failure reporters under our fault panel. **More stars and smaller bars are better.** Stars indicate the quality of a report; bars indicate the detection time. A maximum of four stars are awarded for detecting a failure, giving a certain report, giving more information 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.

testbed looks like this:



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.[6] Our topology has the same size 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) [25], and use the Quagga networking suite [55] patched to detect link failure with BFD [38]. 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.

Figure 5 lists the panel of faults in our experiments. Although the *faults* are synthetic, the resulting *failures* model a class of actual problems.

5.1 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 6 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 so. And none of the baselines gives proactive warnings, as Pigeon does for the final two faults in the panel. In Section 5.2, 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 delay between then and when the reboot affects processes.)

Pigeon has full coverage, at least in our experiments. Finally, we come to accuracy (recall that Pigeon has to balance coverage, timeliness, and accuracy). In our experiments, Pigeon is accurate: we never observe Pigeon incorrectly reporting 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 4). 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 ad-

perf [2], is the same for a virtual and physical machine in our testbed, and in our experiments, VMs do not contend for physical resources.

[6]A non-assumption is using layer 3: there are data center architectures, based on fat-tree variants, that use OSPF at layer 2 [31].

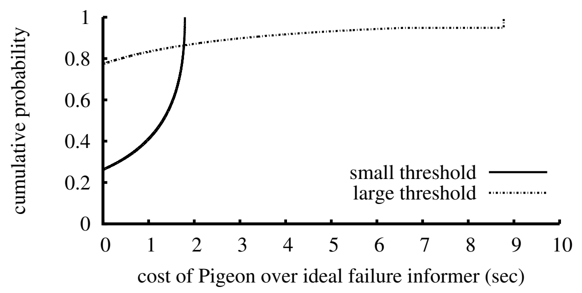


Figure 7—CDF of Pigeon's cost over the ideal failure informer for two sample applications, with availability thresholds (§3.3) smaller and larger than Pigeon's duration estimate.

ditional unavailability that Pigeon causes in two applications: one that always recovers when using Pigeon because its unavailability threshold (§3.3) is smaller than Pigeon's estimate (which is static; see Section 4.4), 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 7 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. Additionally, we find that these simulated applications incur lower costs from using Pigeon compared to choosing "recover" or "wait" uniformly at random.

5.2 Does Pigeon benefit applications?

We consider three case study applications that use Pigeon differently: RAMCloud [52], Cassandra [43], and lease-based replication [30]. For each, we consider the unmodified system, the system modified to use Pigeon, and the system modified to use one or more baselines.

RAMCloud [52]. 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 [46]	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 8—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 saves RAMCloud the cost of recovery sometimes (under link failure (no partition) and network load faults). Falcon [46] 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 20MB 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 8 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 4.4), so it recovers.

Cassandra [43]. Cassandra [43] is a distributed key-value storage system used broadly (e.g., at Netflix, Cisco,

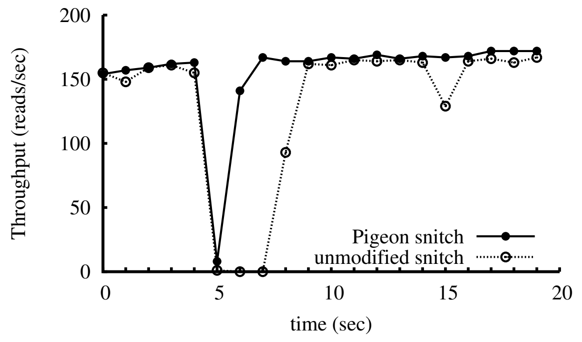


Figure 9—Cassandra's read throughput with and without Pigeon, after a network link fails 5 seconds into the run, temporarily disrupting a single server. Using Pigeon, the Cassandra snitch avoids using an unreachable replica; without Pigeon, Cassandra waits for the server to become reachable again. This 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.

and Reddit [16]). 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 (significant) benefit over no snitch, as the unmodified snitch's decisions are based on latencies—but only if the network is working. This brings us to Figure 9, which depicts the link failure case: here, Pigeon's report to the snitch allows the server to quickly choose a better primary, resulting in higher throughput. Compare to 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 (see above).

Lease-based replication [30]. A common approach to replication is to use a *lease server* [13, 30], 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.[7] 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 there is 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.

5.3 What are Pigeon's costs?

Implementation costs. Pigeon has 5.4K 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 10 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 (§4.2). 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. And this many LSAs are 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

[7]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 (§4)	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 10—Resource overheads of our Pigeon implementation.

presumably use multiple areas; we briefly discuss extending Pigeon to that setting in the next section.

6 Discussion, limitations, and future work

We now consider assumptions and limitations of the failure informer abstraction (§3.1–§3.2), the Pigeon architecture (§3.4), and our prototype implementation (§4).

The abstraction. How do we know if we got the abstraction right? 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 (§2.2); thus, this critique is really a requirement that the implementation have few false positives (§5.1).

The architecture. Our architecture assumes a single administrative domain. This scenario has value (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, 61, 68].

The prototype. Our prototype assumes OSPF, runs on layer 3, and monitors only end-hosts and routers (not middleboxes). We could extend our prototype to other routing protocols, by implementing appropriate relays and sensors (§4.2–§4.3). We could also extend to layer-2 networks, either with OSPF (some layer-2 architectures run OSPF for routing [31]), 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 5.3: 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.

7 Related work

Pigeon borrows low-level mechanisms from prior work in network monitoring and failure handling. We describe these two areas, and also present an extended comparison with Falcon [46], which is Pigeon's closest relative.

Network monitoring and intelligence. Many works in network monitoring [7, 9, 23, 26, 29, 40, 41, 67, 69] are complementary to Pigeon. Broadly speaking, these works extract intelligence from network elements to aid diagnosis, and Pigeon could use these techniques. Indeed, Pigeon's OSPF monitoring technique is borrowed from Shaikh et al. [59, 60] (see Section 4.4). However, the goal of these works is to help network operators perform diagnosis while Pigeon's is to provide an online failure reporting abstraction to distributed applications.

Providing a comprehensive service to distributed applications, using global information about the state of a network, is the goal of *information planes* [19, 65]. Works in this area include the Knowledge Plane [22], Sophia [58, 65] (which provides a distributed computational model for queries), iPlane [49, 50] (which helps end-host applications choose servers, peers, or relays, based on link latency, link loss, link capacity, etc.), and NetQuery [61] (which instantiates a Knowledge Plane under adversarial assumptions). These works are more flexible than Pigeon (they usually expose an interface to arbitrary queries), while Pigeon is more focused: its goal is to report failure conditions to applications, a capability that these papers do not discuss.

More targeted works include Meridian [66] (a node and path selection service), King [32] (a latency estimation service), and Network Exception Handlers [36] (NEHs), which proactively delivers information from the network to the end-host operating system, so end-hosts can participate in traffic engineering. The goals of these systems are different from Pigeon's goal of exposing failures. But again, Pigeon could be extended to use similar techniques, and in fact, Pigeon's callback-based architecture is reminiscent of the delivery mechanism in NEH.

While there are works that do report network failures and errors to end-hosts [5, 42, 64], they do not provide a comprehensive abstraction or full coverage, in contrast to Pigeon's goal. For example, Packet Obituaries [5] (POs) proposes that each dropped packet should generate a report about which AS dropped it. Their credo ("keep the host informed!") is similar to ours, and information about POs would be useful for Pigeon, but POs do not obviate Pigeon. First, under POs, the network generates reports

	Falcon [46]	Pigeon
interface	failure detector (crashes)	failure informer (§3.1–§3.2)
accuracy	always accurate	usually accurate
timeliness	fast	fast
domain	host failures	host+network failures
coverage	incomplete	full
blocks	yes	no
kills	yes	no

Figure 11—Pigeon compared to its most closely related system, Falcon [46]. Falcon has better accuracy, which simplifies the layers over it, but Pigeon is superior in the other respects and in particular leads to higher availability.

proactively but only when the host sends a packet, so this mechanism has the limitation discussed in Section 2.1, of allowing latent failures to persist. Second, POs provide low-level information about individual packets, in contrast to Pigeon's higher-level goal. Third, POs do not provide information about host failures.

Failure recovery and detection. Handling failures requires *recovery* and *detection*. *Host* failure recovery (see [24] and citations therein) is complementary to our work. (From our vantage, strategies such as microreboot [14, 15] are about masking and containing faults; for us, recovery is about what to do when faults cannot be masked.) *Networks*, of course, are designed for recovery, but there are techniques for making them even more robust: Failure Carrying Packets [44] and SafeGuard [47] mask failures by carrying control plane information inside data packets, and in Data-Driven Connectivity [48], data plane packets trigger limited routing state changes. Though these works are orthogonal to ours, an open question is whether applications can benefit from knowing that fault masking is underway.

The other aspect of handling failures is *detection*. Chandra and Toueg [17] gave a theory of failure detection, in the context of a client monitoring a remote process. Since then, a number of failure detectors (FDs) based on end-to-end timeouts have been proposed, including by Bertier et al. [11], Chen et al. [18], and So and Sirer [62]. The ϕ-accrual FD, by Hayashibara et al. [34], extends the FD interface with a measure of *confidence*. This notion of confidence contrasts with Pigeon's notion of "certain crash": the confidence is probabilistic, so even when the ϕ-accrual failure detector reports a crash with high confidence, the monitored process may be up. The failure detection literature, particularly the Falcon failure detector [46], influenced the design of Pigeon; we compare the two systems immediately below.

Pigeon vs. Falcon [46]. Falcon observed the power of low-level information, and Pigeon borrows this observation, but the two have different goals, different properties, and different designs. Figure 11 shows the differences. Falcon is an accurate failure detector [17]—an existing abstraction that reports crash or up; by contrast, Pigeon presents a new abstraction (the failure informer) that exposes more information but with less accuracy. Furthermore, Falcon does not have full coverage of hosts or any coverage of the network; in the non-covered cases, it hangs. In terms of design, Falcon (a) uses low-level information only from hosts, (b) relies on the layered structure of end-host system software, and (c) relies on killing. Pigeon faces a bigger problem (network *and* host failures, and a richer interface), in a landscape that does not admit a layered structure or a license to kill. Thus, Pigeon needs a different design, one that has intelligence from the network and better local knowledge. Furthermore, there is a philosophical distinction in the knowledge provided: Falcon reports the things that it knows it knows, while Pigeon *in addition* gives timely reports of the things that it knows it doesn't know, and eventual reports of the things that it does not know it does not know.

8 Summary and conclusion

The Internet is transparent to success but opaque to failure [5].

Pigeon's top-level contributions are architectural: a thesis that applications should get information about failures, and a proposal to encapsulate that information in a new abstraction that conveys the degree of certainty. Of course, there is much about Pigeon to object to: its ultimate goal (better availability) is shared by all, its design is unsurprising, its mechanisms are borrowed, and its implementation is limited. Nevertheless, this derivative system in fact enables higher application availability, and it does so by enabling new behavior and functionality in applications. Specifically, applications can use the information provided by Pigeon to take the most appropriate action for the failure at hand: to initiate recovery more quickly, to execute a simpler recovery strategy, to recover proactively, or to simply wait it out by not recovering yet. As demonstrated in our experimental evaluation, this freedom leads to qualitatively and quantitatively better behavior, for a modest price in resources. Pigeon, then, is like its namesake: in the wrong environment, it is a homely nuisance; in the right one, it is a key tool with surprisingly powerful functionality.

Acknowledgments

This paper was improved by the helpful comments of Lorenzo Alvisi, Sebastian Angel, Mahesh Balakrishnan, Russ Cox, Alan Dunn, James Grimmelmann, Rodrigo Rodrigues, Srinath Setty, Scott Shenker, and Edmund L. Wong. We thank the anonymous reviewers, and our shepherd Katerina Argyraki, for their suggestions. This research was supported in part by AFOSR grant FA9550-10-1-0073 and NSF grants 1055057 and 1040083.

References

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

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

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

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

[14] 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.

[15] 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.

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

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

[18] 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.

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

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

[37] D. Katz, K. Kompella, and D. Yeung. Traffic enginnering (TE) extensions to OSPF Version 2. RFC 3630, Network Working Group, Sept. 2003.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

[44] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson,

S. Shenker, and I. Stoica. Achieving convergence-free routing using failure-carrying packets. In *ACM SIGCOMM*, pages 241–252, Aug. 2007.

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

[46] 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.

[47] A. Li, X. Yang, and D. Wetherall. SafeGuard: Safe forwarding during route changes. In *ACM Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, pages 301–312, Dec. 2009.

[48] J. Liu, S. Shenker, M. Schapira, and B. Yang. Ensuring connectivity via data plane mechanisms. In *Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2013.

[49] 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.

[50] 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.

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

[52] 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.

[53] 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.

[54] J. Postel. Internet control message protocol. RFC 792, Network Working Group, 1981.

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

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

[57] A. Retana, L. Nguyen, R. White, A. Zinin, and D. McPherson. OSPF stub router advertisement. RFC 3137, Network Working Group, June 2001.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

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

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

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

[65] 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.

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

[67] 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.

[68] 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.

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