Detecting failures in distributed systems with the FALCON spy network

Joshua B. Leners* Hao Wu* Wei-Lun Hung* Marcos K. Aguilera† Michael Walfish*
*The University of Texas at Austin †Microsoft Research Silicon Valley

ABSTRACT

A common way for a distributed system to tolerate crashes is to explicitly detect them and then recover from them. Interestingly, detection can take much longer than recovery, as a result of many advances in recovery techniques, making failure detection the dominant factor in these systems' unavailability when a crash occurs.

This paper presents the design, implementation, and evaluation of Falcon, a failure detector with several features. First, Falcon's common-case detection time is sub-second, which keeps unavailability low. Second, Falcon is reliable: it never reports a process as down when it is actually up. Third, Falcon sometimes kills to achieve reliable detection but aims to kill the smallest needed component. Falcon achieves these features by coordinating a network of *spies*, each monitoring a layer of the system. Falcon's main cost is a small amount of platform-specific logic. Falcon is thus the first failure detector that is fast, reliable, and viable. As such, it could change the way that a class of distributed systems is built.

Categories and Subject Descriptors: C.2.4 [Computer-Communication Networks]: Distributed Systems—Client/Server; Distributed applications; D.4.5 [Operating Systems]: Reliability—fault-tolerance

General Terms: Algorithms, Design, Experimentation, Performance, Reliability

Keywords: Failure detectors, high availability, reliable detection, layer-specific monitors, layer-specific probes, STONITH

1 INTRODUCTION

Many distributed systems must handle crash failures, such as application crashes, operating system crashes, device driver crashes, application deadlocks, application livelocks, and hardware failures. A common way to handle crashes involves two steps: (1) Detect the failure; and (2) Recover, by restarting or failing over the crashed component. Failure recovery has received much attention. For instance, using periodic checkpoints, an entire VM can be failed over in one second [22]; finer-grained components such as processes or threads can be restarted even faster [15, 16]. Interestingly, failure detection has received less attention, perhaps because it is a hard problem. The fundamental difficulty is that uncertain communication delay and execution time make it hard to distinguish a crashed process from one that is merely slow.

Given this difficulty, current approaches to failure detection use a blunt instrument: an end-to-end timeout set to tens of seconds. As a result, after a crash, a system can be unavailable for a long time, waiting for the timer to fire. Indeed, we (and everyone else) are personally familiar with the hiccups that occur when a distributed system freezes until a timeout expires. More technically, examples of timeouts in real systems include 60 seconds for GFS [29], at least 12 seconds for Chubby [14], 30 seconds for Dryad [32], and 60 seconds for NFS. Of course, one could set a shorter timeout—and thereby increase the risk of falsely declaring a working node as down. We discuss end-to-end timeouts further in Section 2.2 and for now just assert that there are no good end-to-end timeout values.

This paper introduces Falcon (Fast And Lethal Component Observation Network), a failure detector that leverages internal knowledge from various system layers to achieve a new combination in failure detection: sub-second crash detection time, reliability, and little disruption. With these features, Falcon can (1) improve applications' availability and (2) reduce their complexity. The target applications are those in data centers and enterprise networks.

A failure detector is a service that reports the status of a remote process as UP or DOWN. A failure detector should ideally have three properties. First, it should be a *reliable failure detector (RFD)*: when a process is up, it is reported as UP, and when it crashes, it is reported as DOWN after a while. Second, the failure detector should be *fast*: the time taken to report DOWN, known as the *detection time*, should be short (less than a second), so as not to delay recovery. Third, the failure detector should cause *little disruption*.

The above properties are in tension with each other and with other desired properties. For instance, a short detection time based on timeouts would compromise reliability, since the detector would report as DOWN a process that is up. As an alternative, a detector could ensure reliability and a short detection time by killing processes [6, 27] at the slightest provocation, but that would be disruptive. Also, short detection times often require probing the target incessantly, which is costly. Another challenge is comprehensiveness: how can the detector maximize its coverage of failures?

The starting point in the design of Falcon is the observation that many crash failures can be observed readily—by looking at the right layer of the system. As examples, a process that core dumps will disappear from the process table; after an operating system panics, it stops scheduling processes; and if a machine loses power, it stops communicating with its attached network switch. In fact, if the failure detector infiltrates various layers in the system, it can provide reliable failure detection using local instead of end-to-end timeouts and sometimes without using any timeouts.

To infiltrate the system, Falcon relies on a network of *spy modules* or *spies*. At the cost of a small amount of platform-specific logic, spies use inside information to learn whether layers are alive. If a layer seems crashed, the spies kill it so that Falcon can report DOWN with confidence. However, killing is a last resort and is *surgical*: Falcon aims to kill the smallest possible layer.

A challenge that we address in Falcon is to provide a careful, thorough, and general design for the collection of spies, to maxi-

SOSP '11, October 23-26, 2011, Cascais, Portugal.

mize detection coverage and to avoid disruption. Spies are arranged 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 application 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. Thus, Falcon includes a backstop: a large end-to-end timeout to cover (the ideally rare) cases that the spies missed. Second, to report DOWN reliably, Falcon must be able to communicate with the remote system. Thus, if a network partition happens, Falcon pauses until the network heals, which we think is acceptable since a partition likely disrupts most services anyway.

We have implemented and evaluated Falcon. In its current implementation, Falcon deploys spies on four layers: application, OS, virtual machine monitor (VMM),[1] and network 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 baseline approaches. This yields higher availability: adding Falcon to ZooKeeper [31] (which provides configuration management, naming, and group membership) and to a replication library [44] reduces unavailability after some crashes by roughly 6×. Falcon's CPU overhead and per-platform requirements are small, and it can be integrated into an application with tens of lines of code. Finally, Falcon can simplify applications that use a failure detector: with RFDs, such applications can shed complex logic that handles failure detector errors (e.g., a replicated state machine can be implemented with primary-backup [9] instead of Paxos [35], thereby using 21% less code, in our rough estimate).

The contributions of this work are as follows:

- *The first viable and fast RFD.* Previous RFDs (§2.2, §7) have drawbacks that make them impractical: large timeouts (to avoid killing aggressively) or disruption from small timeouts. Perhaps for this reason, the conventional wisdom is that a viable RFD cannot be built (§6.1), and indeed, most current failure detectors are unreliable (i.e., not RFDs). Yet, a viable RFD could change the way that we build a class of distributed systems (§6.5).
- *Spies, a spy network, and their composition with existing techniques (§2.3).* Many of Falcon's elements are not new; for instance, killing to achieve reliability has been proposed before and so, for that matter, have end-to-end timeouts, which Falcon uses as a backstop. The new aspects of Falcon are (a) layer-specific monitors (spies); (b) a network of chained spies, where a spy monitors the spy in the next higher layer; and (c) composing these two with existing techniques. We note that the purpose of (a) and (b) is not just fast failure detection; they also reduce false suspicion and kill surgically.
- *The design of Falcon (§3).* We provide a concrete, complete, and sound design for Falcon, based on the key high-level ideas above.
- *The implementation and evaluation of Falcon (§4, §5).*

2 PROBLEM, PERILS, AND PRINCIPLES

2.1 Problem statement and setting

A reliable failure detector (RFD) is a service that, upon being queried about the operational status of a (possibly remote) process p, reports p as UP or DOWN, such that [19]:

- if the RFD reports p as DOWN, then p has crashed;
- if p crashes, then the RFD eventually reports p as DOWN (and does so ever after).

If p crashes, the second property above allows the RFD to report p as UP for some time—called the *detection time*—before it reports DOWN. A *fast* RFD is one with short detection time. We wish to build a fast RFD that is *viable*, meaning that it uses few resources, and that *minimizes disruption*, meaning that it kills only if necessary and, when it does so, kills only the smallest needed component.

Our target setting is a data center or enterprise system. The target applications range from small-scale Web applications that use primary-backup replication [9]; to large-scale storage systems like GFS [29] and Dynamo [25]; to distributed systems that perform batch computations (e.g., MapReduce [24], Dryad [32], and Hadoop [1]); to services, such as Chubby [14] and ZooKeeper [31], that provide common distributed systems functions (group membership, leases, locks, etc.) to other applications.

We assume that (limited) modifications to the software stack are permissible; this assumption holds in our target setting, in which there is a single administrative domain, and may hold in other controlled settings as well. Likewise, we assume that users are trustworthy; access control is orthogonal and could be added to our design. Our approach handles crash failures of any kind; handling Byzantine failures is future work. Also, we design for monitoring within a single data center (though our solution could be used across data centers, with some drawbacks, as discussed in Section 6.4).

2.2 Why is failure detection vexing?

The fundamental difficulty in failure detection is that it is hard to make judgments that are both quick and accurate—a problem that exists in many intelligence contexts. This difficulty leads to a choose-two-of-three situation, in which it is hard to achieve all three of the goals of fast detection, reliability, and little disruption but straightforward to achieve any two of them.

For instance, a failure detector (FD) can achieve accuracy and little disruption by dithering in its reply until there is no question of failure. Alternatively, an FD can achieve a fast detection time if it is willing to jump to conclusions, sometimes producing inaccurate suspicions of failure, at which point there are two ways to handle the inaccuracy. First, the FD can back up its misjudgments by killing the target; however, in converting bad calls into needless kills, this approach sacrifices the goal of little disruption. Second, the FD can give wrong answers, sacrificing reliability; such FDs are *unreliable failure detectors* (UFDs) and force applications—if they are to be responsible—to deal with added complexity, as we elaborate below.

We now highlight the above trade-offs in the context of existing approaches to failure detection; Section 2.3 describes the high-level ideas that we use to break the impasse. The prevalent approach to failure detection uses end-to-end timeouts. The problem is: how does one choose the timeout value? Small values lead to premature timeouts, while large timeouts lead to large detection times. In fact, there may not *be* a perfect timeout value: the difference in latency between normal and delayed requests in data center applications can be several orders of magnitude (e.g., [24]). And while adaptive timeouts (e.g., [11, 21, 30]) might seem promising, adaptation requires time; thus, if system responsiveness changes rapidly (e.g., from bursty load), one does not obtain an RFD.

To get an RFD, the failure detector can kill the process's machine (or virtual machine [5]) before reporting the process as DOWN (e.g., [6, 27]); this killing-based discipline is known as STONITH (for Shoot The Other Node In The Head).[2] Unfortunately, this approach causes disruption: what used to be too-short timeouts convert to

[1] Our current implementation is geared to a system with virtualization, but Falcon can be applied to a system with no virtual machines (§6.3).

[2] STONITH is folklore knowledge that appears to have been around since the 1970s but not in published form.

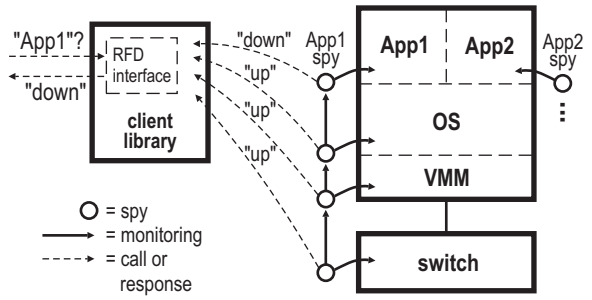

Figure 1—Architecture of Falcon. The application spy provides accurate information about whether the application is up; this spy is the only one that can observe that the application is working. The next spy down provides accurate information not only about its layer but also about whether the application spy is up; more generally, lower-level spies monitor higher-level ones.

needless killing. Other RFD approaches include special hardware (e.g., [52, 53]) or real-time synchronous systems built to bound delays in every case. Such systems are expensive and inappropriate for large data centers, where cost is a key consideration.

Why not give up on RFDs and instead implement an unreliable failure detector (UFD), which is explicitly allowed to make mistakes? UFDs require applications to implement distributed algorithms that handle the case that the UFD reports DOWN when a process is up (and just slow). Unfortunately, such algorithms carry added complexity. An example is Paxos-based consensus [35], used in various systems [13, 14, 18, 31, 33, 39, 43, 49]. Under Paxos, replicas never diverge, even if the system incorrectly detects a crash of the current leader and thereby obtains multiple leaders. Yet Paxos's complexity is well known, as evidenced by the many published papers that try to explain it [18, 34, 36, 37, 40, 45].

Developers have embraced UFDs because of the conventional wisdom that it is impossible to implement a fast RFD that is viable (§6.1). In this paper, we demonstrate that this wisdom is misleading, at least in the context of data centers.

2.3 Design principles

The design principles underlying Falcon are as follows.

Make it reliable. With a *reliable* failure detector, other layers need not handle failure detector mistakes and the resulting complexity.

Avoid end-to-end timeouts as the primary detection mechanism. End-to-end timeouts can serve as a catch-all to detect unforeseen failures, but they take too long to detect common failures.

Peek inside the layers. Layer-specific knowledge can indicate crashes accurately and quickly. For example, if a process disappears from the OS's process table, it is dead, or if a key thread exits, the process is as good as dead. Extracting this information requires a module, which we call a *spy*, at each layer. A spy may use timeouts on internal events (e.g., the main loop has not executed in 1 second), but those timeouts are better informed and shorter than end-to-end timeouts, as they reflect local, more predictable behavior.

Kill surgically, if needed. A spy may not always observe failures correctly, but it must be reliable. Thus, it may kill when it suspects a crash (e.g., the layer is acting erratically or a local timeout has fired). Killing is expensive, so the RFD should kill the smallest necessary component, rather than the entire machine, as in [27, 51, 53]. Such surgical killing conserves resources (e.g., a process is killed while others in the same machine are not) and improves recovery time (e.g., only the process must be restarted, not the machine). A similar argument was made by [15, 16] in the context of reboot.

function	description
init(*target*)	register with spies
uninit()	deregister with spies
query()	query the operational status
set_callback(*callback*)	install callback function
clear_callback()	cancel callback function
start_timeout(*timeout*)	start end-to-end timeout timer
stop_timeout()	stop end-to-end timeout timer

Figure 2—Falcon RFD interface to clients.

Monitor the monitors. Spies are embedded in layers and can crash with them, so spies too should be monitored. This calls for a *spy network*, in which lower-level spies monitor higher-level ones.

3 Design of Falcon

Figure 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 RFD interface to the client, and it coordinates the spies. Roughly speaking, the client library takes as input the identifier of a *target*, which specifies a process whose operational status the client would like to know, and returns UP or DOWN. A spy is a layer-specific monitor. A spy is 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, virtual machine monitor (VMM), and network. Falcon assumes that lower layers enclose higher ones, meaning that if a lower layer crashes, the layers above it also crash or stop responding. This assumption holds by design. As an example, if the VMM crashes, then both the OS and application crash; as another example, if the network crashes, then the higher layers become unresponsive.

The high-level difficulty in realizing Falcon out of spies is how it should interact with them and use their knowledge to meet the desired properties. Our experience is that ad-hoc approaches lead to erroneous designs or ones that do not simultaneously achieve reliability, fast detection, and minimal disruption (§6.2). Achieving these properties together requires carefully addressing the following questions: what interfaces are exposed by the RFD and the spies, what spies do and how, how to orchestrate spies, and how to handle various corner cases. The next sections address these questions in turn, focusing on aspects common to all spies. Section 4 describes the details of the spies in our implementation.

3.1 RFD interface

The RFD interface that Falcon presents to clients is shown in Figure 2. Function *init* indicates the target to be monitored, which identifies each layer (process name, VM id, VMM 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 *set_callback* installs a callback function to be called when a spy reports LAYER_DOWN or the application spy reports LAYER_UP. Function *clear_callback* 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 *start_timeout* and *stop_timeout*.

3.2 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, for example, activity may mean receiving HTTP requests or an indication that there are no requests; for a map-reduce task, activity may mean reading and processing from the 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 VMM, 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 specialized knowledge—which we sometimes call "inside information". A spy exposes three remote procedures:

- *register*() to register a remote callback (which is distinct from the callback to the client in §3.1: 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 LAYER_DOWN; if the layer is operational, the spy calls back the client library periodically, reporting LAYER_UP.

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 LAYER_DOWN must be correct, always. (No exceptions!) Thus, if the spy is inclined to report LAYER_DOWN but is not sure, the spy resorts to killing: it terminates the layer that it is monitoring and *then* reports LAYER_DOWN. (Section 4 explains how spies at each layer kill reliably; the basic idea is to use a component below the layer to be killed.) Of course, spies should be designed to avoid killing.

Figure 3 gives the pseudocode for our spies. UP-INTERVAL is the minimum duration to wait before a spy indicates that the layer is up, to prevent the spy from wasting resources with too frequent LAYER_UP reports; a reasonable value for UP-INTERVAL is 30 seconds. The value of UP-INTERVAL does not affect detection time: a spy reports that the layer is down as soon as it knows.

Below, in Section 3.3, we describe how the client library coordinates the spies, assuming that (1) spies are ideal and (2) network partitions do not happen. Sections 3.4 and 3.5 back off of these two assumptions in turn.

3.3 Orchestration: spies spying on spies

To report the operational status of the target, the client library uses the following algorithm. On initialization, it registers callbacks at each spy at the target and sets a local status variable to UP. If the client library receives a LAYER_DOWN callback from any of the spies, it sets the status variable to DOWN. When the client library receives a query 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 LAYER_DOWN—because 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 LAYER_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 1. So here, the OS spy is monitoring the application spy, and if the application spy is crashed, the OS spy will eventually return LAYER_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 LAYER_DOWN, causing the client library to report DOWN to the client.

```
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 = CERTAINLY_UP then
      if have not called callback within UP-INTERVAL then
        callback(LAYER_UP)
    if rc = SUSPECT_CRASH then
      kill()
      callback(LAYER_DOWN)

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

Figure 3—Pseudocode for spies.

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.

3.4 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 we 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 4. Error A happens 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 requests. Error B happens when there is a violation in the assumption from Section 3.3 that a layer L is up if and only if the spy on layer $L+1$ is responsive. Error C is a spy's reporting LAYER_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 not handled by Falcon and in fact Falcon is expressly designed *not* to have this error: when a spy reports LAYER_DOWN, it must absolutely ensure that the layer is down, which means disconnected from the outside world. Error D is addressed in Section 3.5.

Figure 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.

3.5 Network partition

We said above that lower-level spies monitor higher-level ones, but no spy monitors the lowest level spy. Is that a problem? No, 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 briefly discuss the case of multiple switches in Section 6.4.)

There are three ways to handle network partition. First, the client library can block until it hears from the switch; this is what our implementation does. This is reasonable because during a network partition, other vital services (DNS, file servers, etc) are likely blocked as well, making the system unusable. Second, the client library can, after the client-supplied timeout expires, call back with "I don't know"; this is an implementation convenience that is 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.

3.6 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 $L-1$, 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 then (b) the counter suddenly stops exactly where it was the last time that the client library checked.

```
function init(target)
  for L ← 1 to N do
    invoke register() at spy in target[L]
  Target ← target
  Status ← UP
  Callback ← dummy_function

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

function query()
  return Status

function set_callback(callback)
  Callback ← callback

function clear_callback()
  Callback ← dummy_function

function start_timeout(timeout)
  start countdown timer with value timeout

function stop_timeout()
  stop countdown timer

upon receiving callback (status) from spy in Target[L] do
  if status = LAYER_DOWN then
    Status ← DOWN
    Callback(DOWN)
  if status = LAYER_UP and L = N then Callback(UP)

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 §3.5.
    if got reply then
      Status ← DOWN
      Callback(DOWN)
      return
```

Figure 5—Pseudocode for the client library. N is the number of monitored layers and the layer number of the application.

tag	error / limiting case	cause	effect
A	layer L is down, layer $L-1$ is up, but spy on layer L reports LAYER_UP	bug in layer-L spy	triggers end-to-end timeout and kills
B	layer L is down, layer $L-1$ is up, but spy on layer L is unresponsive	bug in layer-L spy	triggers end-to-end timeout and kills
C	layer L is up, but spy on layer L or below reports LAYER_DOWN	should not happen	would compromise RFD properties
D	none of the spies responds	network partition	RFD blocks or watchdog timer fires

Figure 4—Errors and limiting cases in Falcon, and their effects.

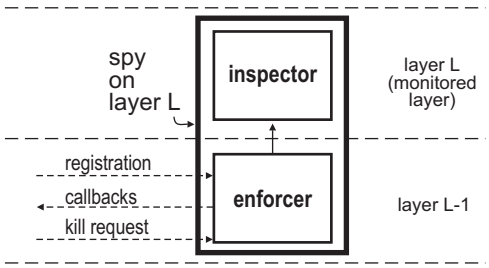

Figure 6—Architecture of spies. A spy has two components: an *inspector* that gathers inside information and an *enforcer* that ensures the reliability of LAYER_DOWN reports (and may also use inside information). The client library communicates with the enforcer.

4 DETAILS OF 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 virtual machine monitor (VMM) 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 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): 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 §3.4), and it must be reliable, meaning that its LAYER_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 7.

- *What are the spy's components, and how do they communicate?* There is a lot of latitude here, but we discuss in Section 6.3 the possibility of a uniform intra-spy interface.
- *How does the spy detect crashes with sub-second detection time?* Although a spy is required to detect crashes of the monitored layer only eventually, it is most useful if it does so quickly.
- *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 meet our goal of little disruption.
- *How does the spy give a reliable answer?* We break this question into two: How does the spy know for sure when its layer is down? If the spy is unsure, how does it kill the layer to become sure?
- *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 6.3 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 layer (e.g., Windows instead of Linux).

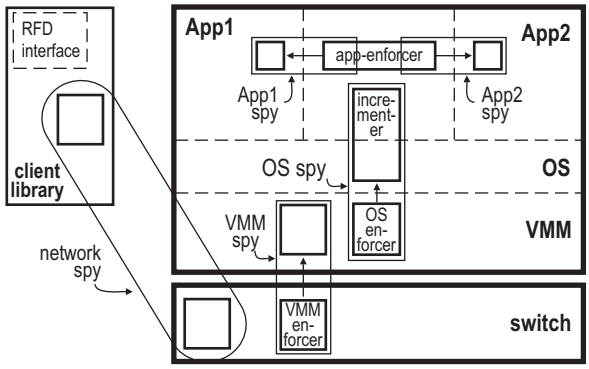

Figure 7—Our implementation of Falcon.

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 `f()`, whose implementation depends on the application. For example, in our primary-backup application spy, `f()` checks whether the main event loop is processing events; in our ZooKeeper [31] spy, `f()` 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 3.3, that "if layer-L is up, then so is the spy on layer-$(L+1)$". As discussed in Section 3.4, if the assumption is violated (which is unlikely), then 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 `f()`. The enforcer infers a crash if `f()` 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 `f()` 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 14, Section 5.4 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 mea-

sures $T_{app\text{-}resp}$ 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 higher-level (human or application) deadlines since those are expressed and enforced by Falcon's end-to-end timeout (§3.4).

A second use of inside information is that $T_{app\text{-}resp}$ is set by the application itself. (Indeed, as mentioned in Section 2.3, timeouts are ideally local and application-specific.) One choice is $T_{app\text{-}resp} = \infty$; in that case, if the app inspector is unresponsive, then Falcon relies on the end-to-end timeout. Or, an application might expect to be able to reply quickly, 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 LAYER_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 LAYER_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 `mlock` it (to prevent swap out). The inspector is implemented in a library; using the library requires only supplying `f()` 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 pid *is* recycled, the end-to-end timeout applies.

OS spy. Our OS spy currently assumes virtualization; Section 6.3 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 VMM. 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, then 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 lack of increments, kill, and report LAYER_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 LAYER_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. If not, 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 would have sufficed to satisfy 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 (§2.3) and does not have the disadvantages of end-to-end timeouts (§2.2).

Reliability. If the VM is no longer being scheduled, the enforcer can verify that case, using its access to the VMM. If the enforcer suspects a crash, it asks the VMM 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 `mlock`. Our VMM is standard Linux; the VMs are QEMU/KVM [46] 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 [41]. We extend this API with a call to check the incrementer's activity. Since all calls into `libvirtd` 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.

VMM spy. Our implementation assumes the ability to deploy new functionality on the switch. We believe this assumption to be reasonable in our target environment of data centers and enterprise networks (§2.1), particularly given the trend toward programmable switches. We also assume that the target is connected to the network through a single interface; Section 6.4 discusses how this assumption could be relaxed.

Components. The inspector is a module in the VMM, while the enforcer is a software module that runs on the switch to which the VMM host is attached. The enforcer infers that the VMM is crashed if, after a period of time in which the switch has not received network packets through the port to which the VMM is connected, the enforcer cannot reach the inspector (this detection method saves network bandwidth, versus more active pinging). The two communicate by RPC over UDP.

Sub-second detection time. Every $T_{VMM\text{-}check}$ time units, the enforcer performs an aliveness check. This check takes one of two forms. Usually, the enforcer checks whether the switch has received network packets from the VMM over the prior interval. If this check fails, or if an interval of $T_{VMM\text{-}check\text{-}2}$ 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

Falcon goal	larger benefit	evaluation result	section
fast detection	availability	• Even simple spies are powerful enough to detect a range of common failures.	§5.1
		• 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.	§5.1
		• Augmenting ZooKeeper [31] and a replication library (PMP) [44] with Falcon (minus killing) reduces unavailability by roughly 6× (or more, for PMP) for crashes below application level.	§5.2
little disruption	availability	• For a range of failures, Falcon kills the smallest problematic component that it can.	§5.3
		• Falcon avoids false suspicions (and kills) even when the target is unresponsive end-to-end.	§5.3
reliability	simplicity	• As an RFD, Falcon enables primary-backup replication [9], which has 50% less replica overhead than Paxos [35], and which requires less complexity (21% less code in our comparison).	§5.5
inexpensive	viability	• Falcon's CPU costs at each layer are single digits (or less) of percentage overhead.	§5.4
		• Falcon requires per-platform code: ≈2300 lines in our implementation. However, the added code is likely simpler than the application logic that can be removed by using an RFD.	§5.5
		• Falcon can be introduced into an application with tens or hundreds of lines of code.	§5.2, §5.5

Figure 8—Summary of main evaluation results.

within $T_{VMM\text{-}resp}$ time units (set to 20 ms in our implementation), it does $N_{VMM\text{-}retry}$ more tries (set to 5 in our implementation), for a total waiting period of $T_{VMM\text{-}resp} \cdot (N_{VMM\text{-}retry} + 1)$ 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 $T_{VMM\text{-}check}$: 100 ms in our implementation.

Avoiding false suspicions. First, our enforcer test is conservative: most of the time, any traffic from the VMM host placates the enforcer. Second, we validate our choice of parameters by running an experiment where 2000 processes on the VMM 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, the operators would need to do more extensive parameter validation for production. Finally, although $N_{VMM\text{-}retry}$ is a constant in our implementation, a better implementation would set $N_{VMM\text{-}retry}$ proportionately to the traffic into the VMM. 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 VMM, by shutting down the network port to which the VMM is connected. The enforcer has no doubt once it has shut down the port, at which point it reports LAYER_DOWN to the client library.

Implementation details. The VMM inspector runs as a process on the VMM (which is standard Linux, as described above). The VMM enforcer is a daemon process that we run on the DD-WRT open router platform [23], 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 as follows. 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.

Localizing network failures via modules in multiple switches is future work (§6.4). For now, we leave the network spy in our design as a placeholder for this extension.

5 Evaluation of Falcon

To evaluate our Falcon implementation, we ask to what degree it satisfies our desired features for a failure detector (FD)—short detection time, reliability, little disruption—and at what cost. We also translate those features into higher-level benefits for the applications that are clients of Falcon. To do so, we experiment with Falcon, with other failure detectors [11, 21, 30] as a baseline, with ZooKeeper, with ZooKeeper modified to use Falcon, with a minimal Paxos-based replication library [44], with that library modified to use Falcon, and with a primary-backup-based replication library that uses Falcon. Figure 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 [42]). 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 9, and the transient condition panel is detailed in Section 5.3. 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 is three hosts connected to a switch. The switch is an ASUS RT-N16. The software on the switch is the DD-WRT v24-sp [23] platform (essentially Linux), extended with our VMM enforcer (§4). 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 VMM. The native (host) OS is 64-bit Linux (2.6.36-gentoo-r5), compiled with the `kvm` module [46], running QEMU (v0.13.0) and a modified `libvirt` [41] (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).

5.1 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 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 [17], static timers are used in many systems, etc.); we borrow the code to implement them from [55]. All of these FDs work as follows: the client pings the target according to a fixed

where injected?	what is the failure?	what does the failure model?
application	forced crash	app. memory error, assert failure, or condition that causes exit
application	app inspector reports LAYER_DOWN	inside information that indicates an application crash
application/ Falcon itself	non-responsive 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
VMM/host	VMM error; causes guest termination	VMM memory error, assert failure, or condition that causes guest exit
VMM/host	`ifdown eth0` on host	hardware crash (machine is separated 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 VMM inspector	bug in Falcon VMM spy

Figure 9—Panel of synthetic failures in our evaluation. The failures are at multiple layers of the stack and model various error conditions.

baseline FD	T: timeout (ms)	error	parameters
Static Timer	10,000	0.0	timer $= 10,000$
Chen [21]	5,001	0.0	$\alpha = 1$ ms
Bertier [11]	5,020	0.0	$\beta = 1, \phi = 4, \gamma = 0.1,$ mod_step $= 0$
ϕ-accrual [30]	4,946	0.01	$\phi = 0.4297$
ϕ-accrual [30]	4,995	0.001	$\phi = 0.4339$

Figure 10—Baseline failure detectors that we compare to Falcon. The implementations are from [55]. 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.

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 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 [14, 29, 32], as noted in the introduction. 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 UFDs) and Falcon (which is an RFD), we also experiment with a UFD 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 §4). 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 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 $[T - p + d, T + d]$;[3] here, T and p are the timeout and ping interval, as defined above and quantified in Figure 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 (§4).

Figure 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 LAYER_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 (§4), 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. Nevertheless, these delays, plus the expected value of 150 ms, explain the observations.

[3]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 $T - p + d$.

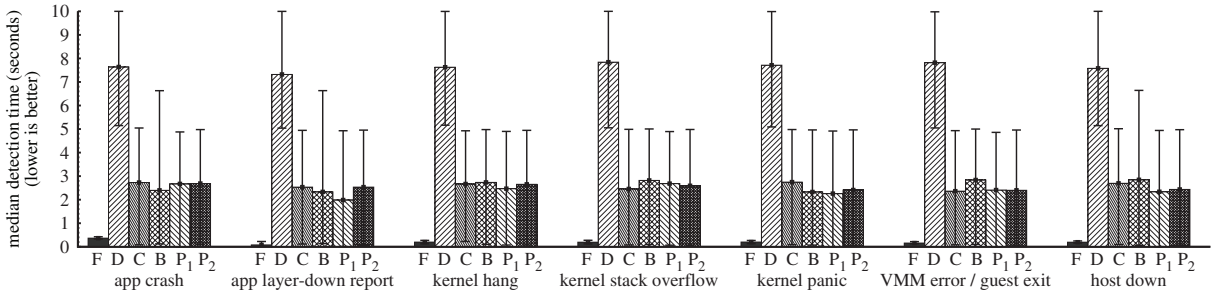

Figure 11—Detection time of Falcon (F) and baseline failure detectors under various failures. The baselines are Static Timer (D), Chen (C), Bertier (B), ϕ-accrual with 0.01 error (P_1), and ϕ-accrual with 0.001 error (P_2); see Figure 10 for details. Rectangle heights depict medians, and the bars depict 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.

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 VMM 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 $T_{VMM\text{-}check} = 100$ ms) plus the 120 ms of waiting (see §4), 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.

5.2 What is Falcon's effect on availability?

We now consider the effect of improved detection time on system availability. We incorporate Falcon into two Paxos-based [35] applications that use failure detectors based on static timers: ZooKeeper [31] (ZK) and a replication library [44] (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 4, and a PMP spy that checks whether the main event loop is running; in both cases, we use Falcon-NoKill, as both systems' unmodified failure detectors are unreliable. The unmodified ZK 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 ZK to use 4 nodes: 3 servers and 1 client (our testbed has 3 hosts, so the client and a server run on the same VMM). ZK partitions the servers into 1 leader and 2 followers. The ZK client sends requests to one of the followers (alternating

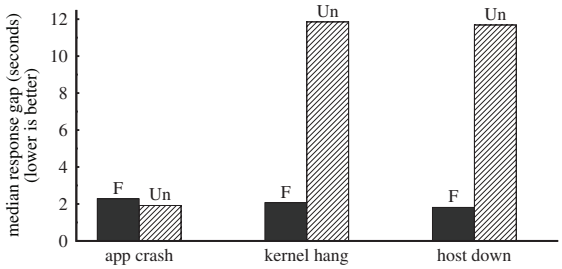

Figure 12—Median response gap (unavailability) of ZooKeeper [31] with Falcon-NoKill (F) and unmodified (Un) under injected failures at the leader. In unmodified ZooKeeper, followers quickly detect application crashes but not kernel- or host/VMM-level crashes. Under the latter types, Falcon reduces median 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.

`create()`s and `delete()`s) when it gets a response to its last one, recording the time of every response. For each of three failure types and the two ZKs, 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:

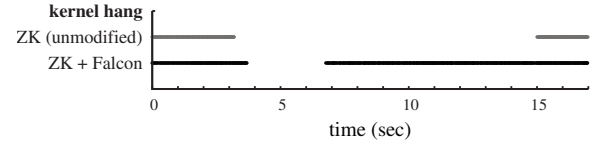

Figure 12 depicts the durations of those response gaps. Under application failures, ZK reacts relatively quickly because the follower explicitly loses its transport session with the leader. Though the median of ZK+Falcon is 350 ms slower than with unmodified ZK, this difference appears due to experimental variation (ZK+Falcon also experiences transport session loss, and the standard deviations are 566 ms for ZK+Falcon and 762 ms for unmodified ZK). Under kernel and VMM/host failures, the ZK 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 ZK 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.

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
VMM error / guest exit	OS enforcer detects failure
host down	VMM enforcer kills VMM/host
crashed app enforcer + app crash	E2E timeout kills guest OS
crashed incrementer	OS enforcer kills guest OS
crashed OS enforcer + OS crash	E2E timeout kills VMM/host
crashed VMM inspector	VMM enforcer kills VMM/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 VMMs	VMM enforcer kills VMM/host

Figure 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 while STONITH, for example, would kill more coarsely. Under the transient conditions, Falcon correctly holds its fire in most cases but sometimes suspects falsely and thus kills.

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.

5.3 How disruptive is Falcon?

We now ask whether Falcon achieves its goal of little disruption, 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 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 4, 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 implementations can target the virtual machine [5]).

For aspect (2), we apply the panel of transient conditions, listed in the bottom part of Figure 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 VMM (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 VMM enforcer and VMM inspector degrades, causing the VMM enforcer sometimes (in 4 out of 15 of our runs) to infer death and kill. As mentioned in Section 4, a better design would set $N_{VMM\text{-}retry}$ adaptively. In the other cases, Falcon's inside information prevents it from killing. For example, the app-enforcer measures $T_{app\text{-}resp}$ based on CPU time (§4), so a long block (e.g., the "hung system call" row) does not cause a kill.

	CPU overhead (percent of a core's cycles)	
component (§4)	**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
VMM inspector	0.39	0.27
VMM total	**8.27**%	**3.07**%
VMM enforcer	0.00	0.00
switch total	**0.00**%	**0.00**%

Figure 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.

5.4 What are Falcon's computational costs?

Falcon's benefits derive from infiltrating the layers of a system. 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 4). 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 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 VMM 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 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 §4). We verified this by separately running the experiment above (Falcon enabled, 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

module (§4)	spy component (§4)	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	VMM inspector	103
DD-WRT extension	VMM enforcer	450
platform-specific total		**2289**
application-specific modules		
`f()` for Paxos (from [44])	app inspector	17
`f()` for primary-backup	app inspector	42
`f()` for ZooKeeper [31]	app inspector	159

Figure 15—The modules in our Falcon implementation and their lines of code. The platform-independent modules assume a POSIX system.

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.

5.5 What is the code and complexity trade-off?

Although we can use Falcon in legacy software (as in §5.2, 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) while the logic shed in applications is complex (tolerate mistakes in an unreliable failure detector).

Figure 15 tabulates the lines of code in our implementation, according to [54]. (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 `f()` (though a production application might wish to embed more intelligence in its `f()`).

Next, we assess the gain to applications that use failure detectors (FDs). Examples of such applications are ZooKeeper, Chubby, state machine replication libraries, and systems that use end-to-end timeouts based on pings of remote hosts.[4] As noted in Section 2.2, if the FD is a UFD, then the application needs complex algorithms that can handle FD mistakes; for example, it might use Paxos [35] for replication. However, if the application has access to an RFD (as provided by Falcon), then it can use simpler approaches; for example it can use primary-backup [9] for replication. Measuring simplicity is difficult, but we compare the lines of code in (1) PMP, which uses a static timer as an FD and Paxos for replication (see §5.2), and (2) a replication library that we implemented, which uses Falcon as an FD and primary-backup for replication. To make the comparison like-to-like, we exclude PMP's FD code from the count.

[4] 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. We discuss ZooKeeper and Chubby further in Section 7.

replication approach	lines of code	# replicas/witnesses
Paxos (from [44])	1759	3
Primary-backup	1388	2

Figure 16—Comparison of two different approaches to replicating state machines: Paxos [35], as implemented in [44], and primary-backup [9], 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.

Figure 16 lists the numbers, again according to [54]. The difference is only 371 lines, but this is 21% of the original code base. And the percentage may be deceptively low: using Paxos in a real system can require intricate engineering [18] whereas primary-backup deployments are not known to suffer similarly. Moreover, primary-backup has lower replication overhead than Paxos: to tolerate a crash, Paxos requires three replicas (or two replicas and a witness), while primary-backup requires just two replicas.

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.

6 Discussion, Extensions, and Outlook

This section discusses how Falcon relates to the conventional wisdom that RFDs cannot be built (§6.1), why we favored Falcon over alternatives (§6.2), how one might apply Falcon to other systems (§6.3), what we see as future work (§6.4), and how Falcon might affect distributed systems more broadly (§6.5).

6.1 Is the conventional wisdom wrong?

The conventional wisdom holds that a viable fast RFD cannot be built, except with specialized hardware. So how did we build Falcon? We explain the arguments for this wisdom by both practitioners and theoreticians, and how Falcon overcomes them.

Practitioners argue that there is an inherent trade-off between detection time and either accuracy or little disruption (§2.2). This trade-off also applies to Falcon: for instance, by reducing the local timeouts of spies, we can get even faster detection and more frequent killing. However, by using inside information, Falcon shifts the trade-off curve to a point where it becomes almost insignificant: even when Falcon is configured to be relatively unaggressive, it often has very fast detection time (§5.1).

Theoreticians argue that RFDs cannot be implemented in asynchronous systems subject to failures because RFDs can be used to solve consensus, and consensus is impossible in such systems [28].[5] Falcon does not contradict this: the theoretical result holds in a model in which processes cannot infer crashes, and part of our point is that processes *can* infer crashes, using inside information. Furthermore, real systems are not asynchronous—a point that we and others have made before [8]. Of course, a system can sometimes experience large delays, thereby behaving like an asynchronous system; this causes Falcon to block temporarily, but that may be tolerable (§3.5).

[5] Even in partially synchronous systems [26], where consensus can be solved, one can prove that RFDs cannot be implemented [38].

6.2 Alternatives to Falcon

Falcon has two backstops: an end-to-end timeout, to catch unexpected conditions, and a chained spy structure, where the spy on a layer monitors the spy on the next layer, to catch the death of spies themselves. An alternate design would be to eliminate the chained structure by not insisting that spies be monitored: there would be a set of ad-hoc spies, each tuned to a particular vulnerability. We did not pursue this design because a problem that both crashed a spy and triggered the vulnerability monitored by the spy would not be detected until the end-to-end timeout expired. Falcon, in contrast, can often detect this case quickly.

Falcon uses local timeouts within each layer. One might wonder if the local timeouts could be replaced with an end-to-end timeout that is the minimum or the sum of the local timeouts. The answer is no: with the minimum, there would be more frequent killing, and with the sum, the detection time would be much larger. In fact, even with the sum, there would be more frequent killing relative to Falcon: spies can avoid killing based on internal signs of life that are not visible end-to-end. One might wonder how this observation relates to the end-to-end argument [47]. The end-to-end argument states that functionality should be implemented at the end hosts (the highest layers), when it is possible to do so completely and correctly. In our context, however, the desired functionality—detecting failures quickly and reliably—can be provided only by infiltrating the layers.

Another design alternative concerns the handling of *intermittent failures*, such as temporary slowness of the target. We designed Falcon to avoid reporting an intermittent failure as a crash whenever possible, but an alternative is to conflate both problems. We eschewed that design for three reasons. First, clients may want to distinguish a crash from an intermittent failure, because the former requires recovery with non-zero cost, while the latter is self-healing. Second, to report an intermittent failure as a crash, an RFD must kill, causing possibly unnecessary disruption. Third, by using Falcon and a timer, a client can infer an intermittent failure, by observing that the target is unresponsive while Falcon deems it operational. However, a service that reports *where* the intermittent failure is, without calling it a crash, might be useful and would be an instance of an FD with richer failure indication (§6.4).

6.3 Applying Falcon to different platforms

Although Falcon's implementation targets a particular platform, we think that its overall design is general. With a different platform, Falcon needs to be tailored for two reasons. (1) The layers may be different: the platform may or may not have VMs, nested VMs [10], Java Virtual Machines (JVMs), etc. (2) A layer may have a different instantiation: the OS layer could be Windows instead of Linux, the VMM could be VMware instead of Linux with QEMU/KVM, etc.

We believe that we can keep small the tailoring from (1). The key is to standardize the communication between enforcer and inspectors, which would let us build different spy networks with minimal changes to the spies. With standardization, we could handle the case of no VMM by moving the OS enforcer to the network driver and leaving the OS inspector unchanged. Or we could insert into the JVM layer a JVM inspector and an enforcer for Java applications and leave unchanged the current app enforcer and app inspectors.

Reason (2), in contrast, requires reimplementing spies. However, because there are few OSs and VMMs, a small number of OS and VMM spies could cover most platforms. And while the application and network spy need to be implemented for each target, this cost is modest (see Section 5.5 for counts of lines of code).

Falcon is only as good as its spies, so how can a developer design useful new spies? Here are three guidelines. First and foremost, do not kill aggressively. Even if the spy monitors few conditions, if it does not kill aggressively, Falcon will fare better than an FD based on end-to-end timeouts alone because Falcon detects the failures monitored by the spy quickly and other failures as fast as the end-to-end timeout. Second, optimize for monitoring the common failures because therein lies the most benefit. Third, design the spy as an iterative process, as the common failures may be unknown at first. That is, the designer should first develop and deploy a simple spy based on some rough knowledge of failures; then observe that the spy fails to detect some common problem quickly (in which case the end-to-end timeout fires); then enhance the spy, redeploy it, and iterate until it catches all common problems. We used this process to design some of the spies in Section 4.

6.4 Future work

Richer failure indication. When a crash occurs, Falcon outputs a simple failure indication, but its spy network has much more information: which layer failed and what problem was observed in that layer. It would be useful to extend the FD interface to expose this data to help applications recover.

Monitoring across data centers. We have been assuming that the client library and the target are in the same physical data center. If they are in separate data centers but in the same administrative domain, our implementation still works, with the proviso that Falcon would block more often, since blocking happens if the client library cannot communicate with the target's switch (§3.5,§4). If they are in different administrative domains, Falcon would need to incorporate access control and permissions.

Scalable monitoring. Our focus has been one process monitoring another, but Falcon also works if $n > 1$ processes monitor each other. However, there will be $O(n^2)$ monitoring pairs, which should give us pause. Nevertheless, the actual resources consumed can be made efficient. When a layer fails, the detecting spy sends only $O(n)$ LAYER_DOWN reports. To avoid $O(n^2)$ messages during healthy times, one option is to eliminate LAYER_UP reports; another is to extend Falcon with techniques such as gossiping [50].

Targets with multiple network interfaces. Falcon currently assumes that the target's host is connected to a single switch, so that the VMM enforcer can kill the VMM by disabling the port of the target's host on the switch. If the target's host is connected to multiple switches, we need to deploy a VMM enforcer at each switch to disconnect all the ports of the target's host.

Network failure localization. Ultimately, we would like to extend Falcon's spy network downward, into the network, to enable failure localization. While one can imagine deploying spies on switches en route from client library to target, this approach raises complex questions related to the algorithms for detection, the approach to remediation (killing will not be viable in many scenarios), and the model for access control and administration.

6.5 Outlook

We finish by considering Falcon's potential effect on distributed systems, based on our expectations, postulations, and speculations.

The key features of Falcon are faster detection and reliability. With faster detection, Falcon may change distributed systems in four ways. First, it can improve availability by removing the periods when the system freezes for several seconds waiting for an end-to-end timeout to expire. Second, because detection is faster,

the system has extra time to recover, so it can try multiple recovery strategies. For example, there may be enough time to restart and retry the failed component before taking the more drastic failover action. Third, with extra time to recover, the system could spend fewer resources during normal operation. For example, there may be no need to keep a warm backup or to checkpoint the state as often. Fourth, the system can afford more frequent failures while maintaining the same availability, which allows for cheaper components and less redundancy.

Besides fast detection, Falcon provides reliable detection, which could simplify the design of some distributed systems and algorithms—a point discussed in Sections 2.2 and 5.5 and which we now briefly elaborate. There are many abstractions to help build distributed systems including atomic registers, atomic broadcast, leader election, group membership, view synchrony, and transactions. However, these abstractions bring difficulty: materializing them has required much thought and work in both theory and practice. The difficulty arises because distributed systems have many sources of uncertainty: failures, slow messages and processes, concurrency, etc. Falcon does not remove all sources of uncertainty, but in its target domain—crash failures in data centers and enterprise networks—it eliminates a vexing one: the ambiguity between slowness and failures.

7 Related work

Before describing other approaches to failure detection, we give context. A formal theory of failure detectors, including definitions for several classes of FDs (reliable, different kinds of unreliable, etc.), was given by Chandra and Toueg [19]. That work established that, with RFDs (as opposed to UFDs), simpler solutions for consensus and atomic (totally-ordered) broadcast were possible. Subsequently, the theoretical advantages of *fast* RFDs were established [7]. Despite this body of theory, it was not known how to build an inexpensive failure detector that is reliable, fast, and minimally disruptive, so we organize related work in terms of the trade-offs among these characteristics.

We begin with unreliable FDs. Chen et al. [21] propose a failure detector based on freshness points and end-to-end timeouts, where the value is chosen adaptively based on delay and loss measurements. Such end-to-end timeouts could be set using other techniques too [11]. These approaches provide a binary indication of failure. Accrual failure detectors [30], in contrast, output a numerical value such that, roughly, the higher the value, the higher the chance that the process has crashed. In practice, applications consider the output to be an indication of failure if it is above a certain threshold. There has also been a strand of work on scaling the failure detector to a large number of processes, with gossiping [50]. This approach also uses end-to-end timeouts, again resulting in a UFD. Each of the above UFDs must trade detection time and accuracy, and none yields an RFD: end-to-end timeouts can be premature, and the guarantees of accrual FDs are probabilistic.

To realize an affordable RFD, one could augment any of the unreliable FDs above by backing up suspicion of failure with killing. In that case, the tradeoff becomes fast detection versus disruption, as what used to be false FD suspicions become needless kills. Such reliable failure detectors can be implemented using watchdogs [27], where the watchdog resets the machine based on an end-to-end timeout. Likewise, the Linux-HA project [6] provides a service called Heartbeat, which provides a failure detection service based on end-to-end timeouts and can be configured to use a hardware watchdog, or STONITH of real or virtual machines. Similarly, with virtual synchrony [12] there is a notion of a process group (which corresponds to the set of operational processes), and if a process becomes very slow, it is excluded from the group via an end-to-end timeout, which is akin to killing. In contrast to all of these approaches, Falcon provides surgical killing and uses fine-grained inside information to detect failures faster than an end-to-end timeout would allow.

Surgical killing and fine-grained monitoring have appeared before but in different contexts. Candea et al. [16] articulated the benefits of surgical killing (faster recovery time, less disruption), and we concur. However, that work focuses on the application layer only, and it solves an orthogonal problem to detection, namely recovery. Fine-grained information is used in cluster monitoring, which collects information about the current condition of hosts in a cluster (e.g., [2–4]), possibly using application-specific data (load, queue lengths, etc.). In contrast to Falcon, these services peek inside only one layer (the application), monitor machines using an end-to-end timeout, and do not have a license to kill (which is needed to get an RFD). Fine-grained information is also used in the leader election service of [48], which enhances a timeout-based failure detector by suspecting a target if its pipe to a local module is broken. Here too, the fine-grained information is limited to one layer, and the failure detector does not kill.

A technique that does involve killing, which is used to increase the availability of Web servers and other services, is to deploy a local script that periodically checks if the application process is running. If not, or if the process has erratic behavior (such as very high CPU usage), the script restarts the application, killing it first if necessary. This technique is limited to one layer (nothing monitors the script) and does not report the failure status to a remote process.

A system that can provide information about failures is ZooKeeper [31], a service for configuration management, naming, and group membership. Its *ephemeral objects*—objects that disappear when the creator is deemed to have crashed—allow other clients to detect the creator's failure status. However, to implement these objects, ZooKeeper internally needs a bona fide failure detector. It uses a UFD (§2.2) for this purpose, so its ephemeral objects provide unreliable detection. However, we replaced its UFD here with Falcon, and though we did not experiment much in this configuration, the change makes ephemeral objects reliable and fast.

Another distributed systems building block is Chubby [14], a lock service with named objects, sessions, and other features. Chubby can address some of the problems that Falcon does. For example, Chubby can avoid two active primaries in some applications. This is done with locks: the primary owns a lock and has a session with Chubby. If the primary fails, Chubby releases the lock only after the primary has lost its session. For this purpose, Chubby uses large end-to-end timeouts and complex session management logic; if incorporated into Chubby, Falcon could replace the former and simplify the latter.

Other production services could also replace their failure detectors, which are based on end-to-end timeouts, with Falcon. For example, GFS [29] uses a timeout of 60 seconds for the primary of a chunk. BigTable [20] uses end-to-end timeouts for sessions with Chubby; it also uses timeouts to expire tablet servers. Dynamo [25] uses end-to-end timeouts in its gossip protocol and between communicating nodes.

8 Summary and conclusion

We began by observing that tolerating crashes requires not only recovering from them—a problem that has been extensively

studied—but also detecting them in the first place, a problem that has received comparatively less attention. This problem brings challenges, whose ultimate cause is the difficulty of quickly and accurately classifying what is truly happening at a remote target. To lift the fog of war, Falcon infiltrates the layers of a remote system with spies, chains spies into a spy network, and combines these with existing techniques (reliability by killing, end-to-end timeouts as a backstop, etc.). To us, the most interesting aspect of Falcon is not any of its individual techniques but rather that it composes them into a system that achieves—as an ad-hoc design would not, judging by our own discarded designs and revised reasoning—sub-second detection, reliability, little disruption, and tolerable expense. This combination is the key contribution of Falcon, and having made it, Falcon now has the chance, we hope, to yield broader benefits: distributed systems that for the user are more responsive and for the designer are more tractable.

Our implementation and experimental configurations are available at: `http://www.cs.utexas.edu/falcon`

Acknowledgments

An early prototype by James Kneeland inspired some of our implementation choices. This paper was improved by careful, detailed comments by Hari Balakrishnan, Allen Clement, Russ Cox, Trinabh Gupta, Carmel Levy, J.-P. Martin, Venugopalan Ramasubramanian, Chao Ruan, Srinath Setty, Sam Toueg, Edmund Wong, Emmett Witchel, the anonymous reviewers, and our shepherd, Marvin Theimer. The research was supported in part by AFOSR grant FA9550-10-1-0073 and NSF grants 1055057 and 1040083.

References

[1] `http://hadoop.apache.org`.

[2] `http://www.managementsoftware.hp.com`.

[3] `http://www.bmc.com/products/brand/patrol.html`.

[4] `http://www.ibm.com/software/tivoli`.

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

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

[7] 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.

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

[9] 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.

[10] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles project: Design and implementation of nested virtualization. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 423–436, Oct. 2010.

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

[14] 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.

[15] 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.

[16] 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.

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

[18] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 398–407, Aug. 2007.

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

[20] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218, Nov. 2006.

[21] 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.

[22] 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.

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

[24] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, Dec. 2004.

[25] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, Oct. 2007.

[26] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Apr. 1988.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] J. Kirsch and Y. Amir. Paxos for system builders: an overview. In *International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, Sept. 2008.

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

[36] L. Lamport. Paxos made simple. *Distributed Computing Column of ACM SIGACT News*, 32(4):51–58, Dec. 2001.

[37] B. Lampson. The ABCD's of Paxos. In *ACM Symposium on Principles of Distributed Computing (PODC)*, page 13, Aug. 2001.
[38] M. Larrea, A. Fernández, and S. Arévalo. On the impossibility of implementing perpetual failure detectors in partially synchronous systems. In *Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pages 99–105, Jan. 2002.
[39] E. K. Lee and C. Thekkath. Petal: Distributed virtual disks. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 84–92, Dec. 1996.
[40] H. C. Li, A. Clement, A. S. Aiyer, and L. Alvisi. The Paxos register. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 114–126, Oct. 2007.
[41] libvirt: The virtualization API. `http://libvirt.org/`.
[42] Linux kernel dump test module. `http://kernel.org/doc/Documentation/fault-injection/provoke-crashes.txt`.
[43] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 105–120, Dec. 2004.
[44] D. Mazières. Paxos made practical. `http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf`, as of Sept. 2011.
[45] R. D. Prisco, B. Lampson, and N. Lynch. Revisiting the Paxos algorithm. *Theoretical Computer Science*, 243(1–2):35–91, July 2000.
[46] Kernel based virtual machine. `http://www.linux-kvm.org/`.
[47] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, Nov. 1984.
[48] N. Schiper, S. Toueg, and D. Ivan. Leader elector source code. `http://www.inf.usi.ch/phd/schiper/LeaderElection`.
[49] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris. Flexible, wide-area storage for distributed systems with WheelFS. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 43–58, Apr. 2009.
[50] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *International Middleware Conference (Middleware)*, pages 55–70, Sept. 1998.
[51] P. Veríssimo. Uncertainty and predictability: Can they be reconciled? In *Future Directions in Distributed Computing (FuDiCo)*, pages 108–113. Springer-Verlag LNCS 2584, May 2003.
[52] P. Veríssimo and A. Casimiro. The Timely Computing Base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930, Aug. 2002.
[53] P. Veríssimo, A. Casimiro, and C. Fetzer. The Timely Computing Base: Timely actions in the presence of uncertain timeliness. In *International Conference on Dependable Systems and Networks (DSN)*, pages 533–542, June 2000.
[54] D. A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`.
[55] GSoC 2010: ZooKeeper Failure Detector model. `http://wiki.apache.org/hadoop/ZooKeeper/GSoCFailureDetector`.