Detecting failures in distributed systems with the FALCON spy network

Josh Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera (Microsoft Research), and Michael Walfish

Abstract. Distributed systems must be resilient to crash failures, by detecting and recovering from them. Interestingly, detecting can take much longer than recovering, due to many advances in recovery techniques, making failure detection the dominant factor in these systems' unavailability when a crash occurs.

We present FALCON, a failure detector with several features. First, FALCON has sub-second detection time, which keeps system unavailability low. Second, FALCON is reliable: it never reports a process as down when it is actually up. This feature can significantly simplify application logic. Third, FALCON sometimes kills to achieve reliable detection but, unlike many other systems, aims to kill the smallest needed component. FALCON achieves these features by coordinating a network of *spies*, each of which monitors a layer of the system. The main cost of FALCON is a small amount of platform-specific logic.

reviewer confused because it looks like end-to-end detection takes many seconds. should clarify this later.

1 Introduction

Distributed systems should be designed to tolerate crash failures, such as application crashes, operating system crashes, device driver crashes, application deadlocks, application livelocks, and hardware failures. A common way to tolerate crashes involves two steps: Step one, detect the crash. Step two, fix the problem, by restarting or failing over the crashed component. There has been a lot of recent work on step two; for instance, using periodic checkpoints, an entire VM can be failed over in a second [21]; finer-grained components such as processes or threads can be restarted even faster [14, 15]. Interestingly, step one has received less attention, perhaps because detecting crashes 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 to respond.

Given this uncertainty, 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 spend a long time unavailable, 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 [27], at least 12 seconds for Chubby [13], and 30 seconds for Dryad [30]. 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.

NOTE: our eval doesn't measure error, only cost. We give the competition a free ride on error.

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: subsecond crash detection time, reliability, and little disruption. With these features, Falcon can (1) improve applications' availability and (2) reduce their complexity. Falcon targets data centers and enterprise networks, and it handles crash failures; Byzantine failures is future work.

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. The time taken to report DOWN is called the *detection time*. Second, the failure detector should be *fast*: the detection time should be short (less than a second), so as not to delay recovery or failover. Third, the failure detector should cause *little disruption*.

It is hard to provide the above properties simultaneously. For instance, a short detection time would compromise reliability, as the detector would report as DOWN a process that is up. Or else, a detector could ensure reliability by killing processes at the slightest provocation, but that would be disruptive. Another difficulty is that short detection times often require probing the target incessantly. A final challenge is comprehensiveness: how can the detector maximize its coverage of failures?

The key idea in our Falcon failure detector is 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 (or enforce) reliable failure detection, using local instead of end-to-end timeouts, and sometimes without using any timeouts at all. 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 the layer is 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 localize the problem and kill the smallest possible layer. If a network partition prevents Falcon from killing, Falcon pauses until the network heals, which is acceptable because a partition likely disrupts most services anyway.

A challenge that we address in Falcon is to provide a careful, thorough, and general design for the collection of spies, to maximize the detection coverage and avoid disruption. To these ends, spies are arranged in a hierarchical network, where the spy in layer X monitors the spy at layer $X+1$, so that every layer (except the bottom-most, the network) is monitored. This interlocked monitoring ensures that, if any layer in the system crashes, some spy will observe it, with the backstop being a large end-to-end timeout to catch imperfections in the spies.

We have implemented and evaluated Falcon. In its current implementation, Falcon deploys spies on four layers: applications, OS, virtual machine monitor (VMM),[1] and network switch. We find that for a range of failures, Falcon has sub-second detection time (one or two orders of magnitude faster than baseline approaches). This yields higher availability: we added Falcon to ZooKeeper [29] (which provides configuration management, naming, and group services) and a replication library [41], finding that Falcon reduces the unavailability after some crashes by 4–7×. 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, we show that Falcon can simplify application logic: without an RFD, replication requires a complex protocol like Paxos, whereas an RFD enables a simpler primary-backup protocol with 25% fewer lines of code.

We suggested layer-specific monitors in a position paper [8], but that paper did not present a design, an implementation, or an evaluation. The specific new contributions of our paper are: (1) A new approach to detecting failures in distributed systems; (2) The design, implementation, and evaluation of Falcon, which embodies that approach and responds to the challenges above; (3) A demonstration that fast and reliable failure detection is viable (contrary to conventional wisdom): it causes little disruption and is affordable.

2 Problem and principles

2.1 Problem statement and setting

A reliable failure detector (RFD) is a service that, upon being queried about a (possibly remote) process p, returns a report on p as UP or DOWN, such that [18]:

- 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 everafter).

If p crashes, the above property 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 minimizes disruption, by using few resources, and killing only the smallest needed component.

Our target setting is a data center or enterprise system. There is a single administrative domain, and users are trustworthy; access control is an orthogonal concern that could be added to our design. The target applications range from small-scale Web applications that employ primary-backup replication [7] to large-scale storage like GFS [27] and Dynamo [24]. Other targets include services, such as Chubby [13] and ZooKeeper [29], that provide common distributed systems functions (leader election, leases, locks, etc.) to other applications.| Given this setting, we assume that (limited) modifications to the software stack are permissible.

anything else that we should cite?

Our approach handles crash failures of any kind. Handling Byzantine failures is future work. Our failure detector is designed for monitoring within a single data center; it could be used across data centers, with some drawbacks, as discussed in Section 6.

2.2 Issues with existing failure detectors

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., [23]).| And while adaptive timeouts (e.g., [10, 20, 28]) might seem promising, adaptation requires time; thus, if system responsiveness changes rapidly (e.g., from bursty load), one does not obtain an RFD.

Josh, team, we should get other citations here if possible.

One way to get an RFD is for the failure detector to kill the process's machine before reporting the process as DOWN (e.g., [25]), to remove any doubt—a discipline 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 needless killing. Other RFD approaches include special hardware (e.g., [48, 49]) 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.

Another approach is to give up on RFDs and instead implement an *unreliable failure detector* (UFD), which can make mistakes. One then designs 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 [33], used in various systems [13, 17, 29, 31, 36, 39, 44]. Under Paxos, replicas never diverge, even if the system incorrectly detects

[1]The current implementation is geared to a system with virtualization; if the system has no virtual machines, the VMM logic could run in a driver in the OS kernel, as we discuss in Section 6.

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

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 [17, 32, 34, 35, 37, 42].

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

2.3 Design principles

The design of Falcon follows the following principles:

Make it reliable. With a *reliable* failure detector, other layers need not deal with failure detector mistakes.

Avoid end-to-end timeouts as the primary detection mechanism. End-to-end timeouts are useful as a catch-all mechanism 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, then it is dead; if a key thread exits, the process is effectively dead. Extracting this information requires embedding 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 [25, 47, 49]. 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 [14, 15] in the context of reboot.

Monitor the monitors. Spies are embedded in layers and can crash along with the layer, so they should be monitored for crashes. We employ a *spy network*, in which broader-scope spies monitor narrower-scope ones.

3 Design of Falcon

Figure 1 depicts Falcon's high-level 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. The client library takes as input the identifier of a *target*, a process whose operational status the client would like to know, and returns

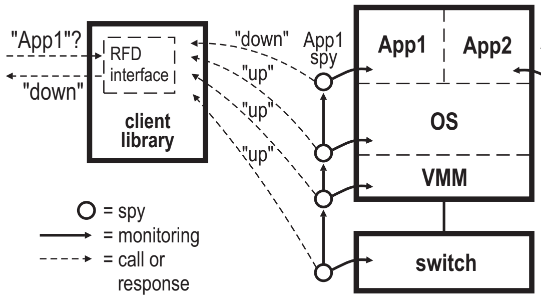

Figure 1—Architecture of Falcon. The application spy provides accurate information about whether the application is up; this spy is the only one that can observe that the application is working. The next spy down provides accurate information not only about its layer but also about whether the application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

UP or DOWN. A spy is a layer-specific monitor, named by the layer monitored (e.g., the OS spy monitors the OS); the spy itself may have parts running at several layers. The layers monitored by the current implementation are application, OS, virtual machine monitor (VMM), and network. Note that lower layers enclose higher ones: if a lower layer crashes, the layers above it also crash or stop responding.| (For example, if the VMM crashes, both the OS and application crash; if the network crashes, the higher layers are unresponsive.) Falcon makes use of this enclosing property. In the rest of this section, we describe the objective, operation, and orchestration of spies, focusing on aspects common to all spies. §4 describes the specific details of each spy in our implementation.

reviewer 75D asks: "what if this wasn't true?". need to clarify the assumptions.

A given layer is supposed to perform some activity, and if the layer is performing it, then the layer is alive by definition. The purpose of a spy is to sense the presence or absence of such activity using specialized knowledge. As examples, for a Web server, 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 number crunching 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.

A spy exposes three remote procedures:

- *register*(*gen*) to register a remote callback;
- *cancel*(*gen*) to cancel it; and
- *kill*(*gen*) to kill the monitored layer.

Parameter *gen* is explained in Section 3.4; for now, we ignore it. If the layer that the spy is monitoring crashes, the spy immediately calls back any registered clients, reporting LAYER_DOWN; if the layer is operational, the spy calls back periodically, reporting LAYER_UP.

A spy is designed to recognize the common case when

tag	error / limiting case	cause	effect
A	spy on layer $L < N$ reports LAYER_UP, but layer L is down	bug in layer-L spy	triggers end-to-end timeout and kills
B	spy on layer $L < N$ reports LAYER_UP, layer L is up, but spy on layer $L+1$ is unresponsive	bug in layer-$L+1$ spy	triggers end-to-end timeout and kills
C	spy on highest layer N reports LAYER_UP, but app is down	bug in layer-N spy	client end-to-end timeout expires and app is killed
D	spy on layer L reports LAYER_DOWN, but a higher-level spy or the application is still up	should not happen	would compromise RFD properties
E	none of the spies respond	network partition	RFD blocks or watchdog timer fires

Figure 2—Errors and limiting cases in Falcon, and their effects. MW: hack latex (bottomfrac, I think): should be *bottom* of page.

the monitored layer is clearly crashed or healthy. What if the spy cannot be certain? To support reliable failure detection, a reply of LAYER_DOWN must be correct, always. (No exceptions!) Thus, if the spy is inclined to reply LAYER_DOWN but is not sure, the spy resorts to killing: it terminates the layer that it is monitoring and *then* reports LAYER_DOWN.| Of course, spies should be designed to avoid killing.

reviewer 75D: "How does it reliably kill the layer?".

Below, in §3.1, we describe how the client library coordinates the spies, assuming that (1) spies are ideal and (2) network partitions do not happen. §3.2 and §3.3 back off of these two assumptions in turn.

3.1 Orchestration: spies spying on spies

To determine the operational status of the target, the client library uses the following algorithm. On initialization, it registers callbacks at each spy at the target. When the client library receives a query from the application, it waits until the application spy reports LAYER_UP or some spy reports LAYER_DOWN, and then accordingly returns UP or DOWN to the application.

To see why this algorithm works, note that if the application spy returns something, the client library reports the status of the target correctly, because we are assuming ideal spies. 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 true purpose of the layer-L spy is to monitor 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 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 return LAYER_DOWN, causing the client library to report DOWN to the client.

We have not yet said how the spy on layer L monitors the spy on layer $L+1$. The spy on layer $L+1$ has a component at layer L, namely the component responsible for responding to queries. Thus, to monitor the spy on layer $L+1$, it suffices for the spy on layer L *to monitor layer L itself*. This avoids the complexity of a signaling protocol among spies. It works because, under the assumption that spies are not buggy, if layer L is down then the spy on layer $L+1$ is down (unresponsive), and vice-versa.

3.2 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 taken seldom.

There are five types of spy errors that we consider, as shown in Figure 2. Error A happens when a spy does not recognize a rare failure condition so 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 3.1 that a layer L is up iff the spy on layer $L+1$ is responsive, which arises because of bugs in the spies. Error C happens when the application spy does not recognize that the application is down. Error D (row 4) is a spy's returning LAYER_DOWN when either the monitored layer is up or any spy above the monitored layer is up. Error E (row 5) occurs when none of the spies responds, because of a network problem such as a partition.

Errors A and B cause a failure detector query of the ideal algorithm in Section 3.1 to hang forever. To address this problem, the actual query interface takes an end-to-end timeout specified by the client, to serve as a backstop. If the end-to-end timeout expires, the algorithm kills the highest possible layer and returns DOWN.

Error C causes a query of the ideal algorithm to always return UP even though the target is down. To address this problem, first note that a client typically uses the RFD by querying the RFD repeatedly until the RFD returns DOWN or some condition is met (e.g., the client receives a message from the target process). Under error C, the client will hang since neither the RFD returns DOWN nor the condition is met. To unhang itself, the client should employ its own end-to-end timeout. If it triggers, the client must ensure that the target is dead; to that end, the client library exposes a function to kill.

Error D is not handled by Falcon and in fact Falcon

```
function initialization(gen)
  for $L \leftarrow 1$ to N do        // $N$ is number of layers
    invoke register(prefix(gen, $L \times 16$)) at spy on layer $L$
      // prefix(word, $i$) returns the first $i$ bits of word
    DeadGens $\leftarrow \emptyset$

upon receiving callback $\langle status, gen \rangle$ from layer $L$ do
  if status = LAYER_DOWN then add $\langle gen, L \rangle$ to DeadGens

function query(e2etimeout, gen)
  start timer with value e2etimeout
  wait for any condition below
    (C1) receive callback $\langle$LAYER_UP, gen$\rangle$ from layer $N$
    (C2) gen matches some generation in DeadGens
        // "matches" means that $\langle prefix(gen, j \times 16), j \rangle$ is in
        // DeadGens for some $j$
    (C3) timer expires
  stop timer
  if condition (C1) then return UP
  if condition (C2) then return DOWN
  if condition (C3) then
    kill-highest-layer(gen)
    return DOWN

function kill-highest-layer(gen)
  for $L \leftarrow N$ downto 1 do
    invoke kill(gen) at spy on layer $L$
    if $L \neq 1$ then wait for reply for SPY_RETRY_INTERVAL
    else wait for reply
    if got reply then return
```

Figure 3—Pseudocode for the client library.

```
remote-procedure kill(gen)
  if gen = curr-gen then
    kill layer we are spying on and wait to confirm kill
  // else layer is already gone
  return ACK

remote-procedure register(gen)
  add caller to clients[gen]
  return ACK

remote-procedure cancel(gen)
  remove caller from clients[gen]
  return ACK

background-task monitor()
  while true
    sense layer and set rc accordingly
    if rc = CERTAINLY_DOWN then
      callback(LAYER_DOWN, curr-gen)
    if rc = CERTAINLY_UP then
      if have not called callback within UP-INTERVAL then
        callback(LAYER_UP, curr-gen)
    if rc = SUSPECT_CRASH then
      kill(curr-gen)
      callback(LAYER_DOWN, curr-gen)

function callback(status, gen)
  for each client $\in$ clients[curr-gen] do
    send $\langle status, gen \rangle$ to client until get ack
```

Figure 4—Pseudocode for spies.

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 E is addressed in Section 3.3.

Figure 3 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 exposes a procedure to kill its layer, which 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, by terminating the highest possible layer. A reasonable value for SPY-RETRY-INTERVAL is 3 seconds; and although it increases the detection time, it only does so when a large end-to-end timeout expires.

reviwer 75D: "does't this violate the "sub-second" failure detection time quoted in the abstract?"

Figure 4 gives the pseudocode for our spies. UP-INTERVAL is the minimum duration to wait before a spy indicates that the layer is up. This is provided for efficiency, so that the spy does not report too frequently that a layer is up. This value does not affect detection time, because a spy reports that the layer is down as soon as it knows, irrespective of UP-INTERVAL. A reasonable value for UP-INTERVAL is 5 seconds. The callback from the spy to the client library need not be communicated over a reliable link but we assume that, in the absence of a network partition, if the spy keeps retransmitting the message, then it eventually gets through.

3.3 Network partition

We said above that lower-level spies monitor higher-level ones. But no spy monitors the lowest level spy, so what happens if that spy does not respond? 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. There are three ways to handle this case. First, the client library blocks until it hears from the switch; this is what our implementation does. Second, the client library can, after the client-supplied timeout expires, return "I don't know"; while this response is conceptually identical to blocking, it is an implementation convenience to avoid blocking the caller inside the client library. Third, the client library reports 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 periodic heartbeats

reviewer asked, "what if a machine has multiple network connections to different switches?", so maybe we should point to the discussion or limitation section where this is discussed?

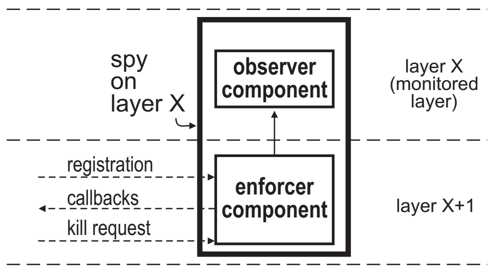

Figure 5—Architecture of spies. A spy has two components: an *observer* that gathers inside information and an *enforcer* that ensures the reliability of LAYER_DOWN reports (and may also use inside information). The client library communicates with the enforcer. MW: figure should should say $X - 1$ where it currently says $X + 1$, right?

from the client library to the switch.

3.4 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 parts of its state in a crash). Therefore, when the application restarts, we treat it as a different instance to be monitored by the RFD, and the original crashed instance is reported DOWN.

For these purposes, the spy on a layer labels the layer with a generation number. Queries of that spy include a generation number; if it is old, a spy returns LAYER_DOWN and its current generation number. 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 16 bits, and sets the low-order 16 bits to a counter that it increments on every restart. (The base case is the generation number of the lowest layer, which can be implemented with a counter.) In particular, at the application level, the generation number contains 64 bits that capture a 16-bit counter for every level below.

The interface to the client library, rather than taking an application name on each query, takes the generation number. The client receives this generation number when it registers with the client library and supplies an application name (this is not shown on the code).

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 not the final word; one can extend spies based on design-time application knowledge or on failures observed in a given system.

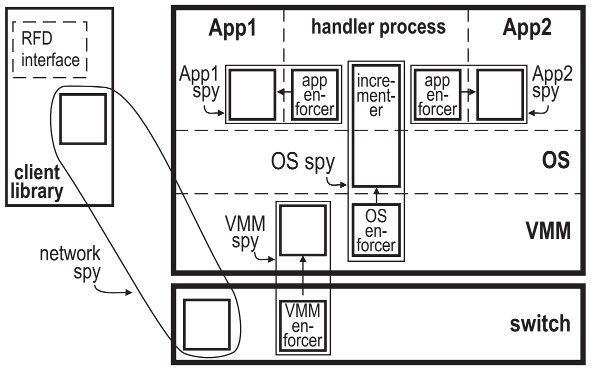

Figure 6—Our implementation of Falcon. MW: if handlerd not split, then say, "A distinguished high-priority process, `handler`, is both the app-enforcer and part of the OS-observer". Also update the text body to that effect, or remove the thing saying that we'll split the two.

Nevertheless, the spies that we present, though mainly illustrative reference designs, should serve as an existence proof that it is possible to react to a large class of failures. As shown in Figure 5, a spy on any given layer has two components, which communicate with each other:

1. *Observer*: 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 is the one that communicates with the client library and is responsible for killing the monitored layer; for those reasons, it typically resides one layer below the monitored layer. This component may also use inside information.

For each class of spy below, we describe the two components. We also describe how a spy detects common crashes with sub-second detection time; how a spy gives a reliable answer; and implementation details. Figure 6 depicts our implementation.

MW: Some things we might want to do in this section: 1. Concentrate the limitations if we have a lot of them (in the submission, it felt like we were frequently apologizing. however, that may happen less with the new implementation). 2. restructure the section around the answers to a set of questions (see the commented-out four questions above).

Application spies All of our application spies have a common organization. The observer is a dedicated thread inside the application that 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 spy, `f()` tests whether the client and replica request queues are being processed. The enforcer is a distinguished high-priority process, the *app-enforcer*, that queries the observers and carries out killing 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 our "if layer-X is up, then so is the spy on layer-$(X + 1)$" assumption, from Sections 3.1–3.2. As discussed, if the assumption is violated (which is unlikely), then Falcon relies on an end-to-end timeout.

The app-enforcer must notify the client library when it detects a crash. To do so, the app-enforcer checks an application every $T_{app\text{-}check}$ time units. If the app-enforcer is certain that the process no longer exists (for example, the process has left the process table), the app-enforcer reports LAYER_DOWN to the client library. Otherwise, the app-enforcer uses IPC to query the observer thread, which invokes `f()`. If the IPC handle returns an error or if `f()` indicates "down", then the app-enforcer kills the process to make sure, waits for confirmation, and then reports LAYER_DOWN to the client library. If the observer thread does not respond within an application-specific $T_{app\text{-}resp}$ time, the app-enforcer kills the application. We measure $T_{app\text{-}resp}$ using the CPU time of the observer, not real time, to avoid timeouts on an overloaded system.|

emphasize that this is an example of inside information? or does eval section make the point enough?

For sub-second detection times, our implementation sets $T_{app\text{-}check}$ to 100 ms. The precise choice is arbitrary, though the order of magnitude (tens or hundreds of milliseconds) is not. Checking does not involve the network, and it is inexpensive—less than 0.17% CPU overhead per-check, according to our experiments (Section 5.4, Figure 13). Thus, we elect to pay a minimal processing cost for rapid detection time in many cases.

An application specifies its own $T_{app\text{-}resp}$ (when its observer registers with the app-enforcer), based on the application's inside information. One choice might be to set an infinite value for $T_{app\text{-}resp}$; in that case, if the app observer is unresponsive, then Falcon relies on the end-to-end timeout, as above. 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 higher availability when the application process is unexpectedly stuck. We note that the observer can also use additional timing considerations, not exposed to the app-enforcer, to make a LAYER_DOWN call. For example, the observer might know, "if a given request is not off an internal queue within 10 ms, then my application is effectively down and I should indicate that", which is different from the use of $T_{app\text{-}resp}$: "if I (the spy) am not heard from after $T_{app\text{-}resp}$ of CPU time, then I am effectively down". Separating these timeouts avoids what would otherwise be a trade-off between disruption and availability.

Implementation details: The observer 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 observer is implemented in a library; the application's job is only to supply an implementation of `f()` and a value of $T_{app\text{-}resp}$. The observer and app-enforcer communicate through a Unix domain socket, the enforcer kills by sending a `SIGKILL`, and the enforcer confirms the kill by checking the process table.

OS spy In our current implementation, the OS enforcer is in the VMM (if the system is not virtualized, we believe it can be elsewhere, as discussed in Section 6). The observer 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. The enforcer is in the VMM and checks a virtual machine every $T_{OS\text{-}check}$ time units. To do so, it first checks whether the VMM indicates that the virtual machine is running. If not, the enforcer reports LAYER_DOWN to the client library. Otherwise, it checks whether the counter has been incremented at least once. If not, the enforcer suspects the OS (or virtual machine) of having crashed, but it cannot be sure. It therefore kills the virtual machine running the OS and reports LAYER_DOWN.

In our implementation, $T_{OS\text{-}inc} = 1$ ms and $T_{OS\text{-}check} = 100$ ms. We validated our choices by running various stress tests. The most stressful is a fork+exec bomb. During this period (modeling a temporary burst), the incrementer still runs: over a 30 minute test (18,000 checks), the enforcer observed, per check, a mean of 100.77 increments, standard deviation of 5, and a minimum of 36 (where 1 increment would have sufficed to satisfy the enforcer). These observations validate our settings for now, but in a production deployment, the operators would obviously need to do a longer "burn-in". Also, the incrementer could conceivably be delayed by a flood of hardware interrupts—which we do not expect to happen frequently. If it did happen, the enforcer would kill the OS, but the spy would not return incorrect information.

An alternate OS spy implementation is for the enforcer to inspect a kernel counter like jiffies, instead of an application-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 application is being scheduled. The cost of this higher-level assurance is the assumption that the high priority application itself does not crash; if that assumption is violated (which is unlikely) then Falcon again relies on the end-to-end timeout.

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`. In fact, as depicted in Figure 6, the incrementer and app-enforcer in our implementation run inside the same process, called `handler` (we plan to separate them in our next version). The OS enforcer leverages the `libvirt` framework [38].

Falcon goal	**larger benefit**	**evaluation result**	**section**
fast detection	availability	• Even simple spies are powerful enough to detect a range of common failures. • For these failure modes, Falcon's 90th percentile detection time is several hundred milliseconds; existing failure detectors take one or two orders of magnitude longer. • Augmenting ZooKeeper [29] with Falcon (minus killing) reduces unavailability by 4–7× in the cases of kernel-level and VMM/host-level crashes.	§5.1, §5.2
little disruption	availability	• For a range of failures, Falcon kills the smallest problematic component that it can. • Falcon can avoid kills when the target is momentarily slow (but would have triggered STONITH).	§5.3
reliable detection	simplicity	• As an RFD, Falcon enables primary-backup replication [7], which has 50% less replica overhead than Paxos [33], and which requires less complexity (25% less code in our comparison).	§5.5
inexpensive	viability	• Falcon's CPU costs are modest: single-digits of CPU percentage at each layer. • Falcon requires per-platform code: ≈1500 lines in our implementation. However, the added code is likely simpler than the shed application logic. • Falcon can be introduced into an application with tens of lines of code.	§5.4, §5.5

Figure 7—Summary of main evaluation results. MW: if time, give better index

The `libvirt` API provides a common interface to different virtualization platforms; the API is exposed through a daemon, called `libvirtd`. Thus, our enforcer, which is an extension to `libvirtd`, works for both unmodified Xen [9] and QEMU/KVM [43] virtual machines, by invoking the `libvirt` API to examine guest virtual memory and kill guest virtual machines.

VMM spy Our implementation assumes that the target is connected to the network through a single interface; we can relax this assumption, as discussed in Section 6. The observer is a module in the VMM, while the enforcer is a module in the switch to which the VMM host is attached. Every $T_{\mathit{VMM\text{-}check}}$ time units, the enforcer checks the observer's aliveness (our implementation sets $T_{\mathit{VMM\text{-}check}}$ to 100 ms; the considerations here are the same as with $T_{\mathit{app\text{-}check}}$ and $T_{\mathit{OS\text{-}check}}$ above). If the enforcer can determine that the link to the VMM host is down (using status information at the switch), then it shuts down the network port to prevent the link from coming back up; it then reports LAYER_DOWN to the client library. Otherwise, the enforcer queries the observer. If the enforcer does not get a response within $T_{\mathit{VMM\text{-}resp}}$ time units, it does $N_{\mathit{VMM\text{-}retry}}$ more tries, after which it shuts down the port and reports LAYER_DOWN to the client library.

To set $T_{\mathit{VMM\text{-}resp}}$, we deployed Falcon and ran an experiment where the VMM had 3 virtual machines, each running 900 CPU-intensive processes. We set the enforcer to query the observer in a closed loop 100,000 times. We observed that the observer always responded within 75 ms. In fact, the mean response time was 287μs, the standard deviation was 191μs, the maximum was 60 ms, and only 4 out of the 100,000 responses took longer than 1 ms. Given these observations, we set $T_{\mathit{VMM\text{-}resp}}$ to 100 ms (though again, in production, the operators would need to do a much longer burn-in).

We need to set $N_{\mathit{VMM\text{-}retry}}$ so that at least one of the $N_{\mathit{VMM\text{-}retry}}$ queries from enforcer to observer gets through. Our current implementation is sub-optimal: it uses TCP, sacrificing detection time in the particular case that the VMM becomes unresponsive while its link to the switch is up; however, this setup does not affect detection time for other failures. A better implementation would use UDP and rely on additional inside information available at the switch, namely the traffic to and from the VMM. The enforcer needs to query the VMM only if it sees no packets *from* the VMM. Moreover, the enforcer can choose $N_{\mathit{VMM\text{-}retry}}$ proportionately to the traffic *to* the VMM: if the traffic is low then retransmissions are not necessary since the VMM is unlikely to drop packets.

change implementation to use UDP. if that item doesn't get done, then see the commented-out latex below.

Implementation details: The VMM observer is implemented alongside the OS enforcer; both are in our extension to `libvirtd`, described above. The VMM enforcer is a daemon process that we run on the DD-WRT open router platform [22], which we modified to map connected hosts to physical ports and to run our software.

Network spy The observer component here is a module on the switch connected to the target's host, and the enforcer component is the client library itself. Every $T_{\mathit{net\text{-}check}}$ time units, the enforcer queries the switch observer, expecting a reply. If the observer does not reply in $T_{\mathit{net\text{-}resp}}$ time units, then we are in the network partition case, as described in Section 3.3. Reasonable values are 1 and 2 seconds for $T_{\mathit{net\text{-}check}}$ and $T_{\mathit{net\text{-}resp}}$, respectively.

Reduce to match E2E ping intervals?

5 Evaluation of Falcon

To evaluate our Falcon implementation, we ask to what degree it satisfies our desired properties for a failure detector—short detection time, reliability, little disruption—and at what cost. We also translate those properties into higher-level benefits for the applications that are clients of Falcon. To do so, we experiment with Falcon, with other failure detectors [10, 20, 28] as a baseline, with ZooKeeper, with ZooKeeper modified to use Falcon, with a minimal Paxos-based replication library [41], and with that library modified to use Falcon.

where injected?	what is the injected failure?	what does the injected failure model?
application	forced crash	models app. memory error, assert failure, or condition that causes exit
application	app observer reports LAYER_DOWN	models inside information that indicates an application crash
application/ Falcon itself	non-responsive app observer	since the app observer is a thread inside the application, this models a buggy application (or app observer) that cannot run but has not exited
kernel	infinite loop	models kernel hangs and liveness problems
kernel	stack overflow	models runaway kernel code
kernel	kernel panic	models 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	models hardware crash (by separating machine from network)
Falcon itself	crash of `handler` (see §4)	models non-responsive or buggy app-enforcer, or buggy OS observer
Falcon itself	crash of `libvirtd` (see §4)	models non-responsive or buggy OS enforcer, or buggy VMM observer

Figure 8—Panel of synthetic failures in our evaluation. The failures are at multiple layers of the stack and model various error conditions. A non-responsive app observer (third row) models either an application failure or a failure in Falcon itself. We expect this failure and those in the last two rows to be caught by local timeouts: $T_{\textit{app-resp}}$, $T_{\textit{OS-check}}$, and $T_{\textit{VMM-resp}}$ respectively (§4).

Figure 7 summarizes our evaluation results.

Failure panel and environment Most of our experiments involve failures from a *failure panel*: a set of ten kinds of failures that we inject into a running system, to model various types of failures. Figure 8 describes the panel.

Our testbed is three hosts connected to a switch. The switch is an ASUS WL-500gP V2. The software on the switch is the DD-WRT v24-sp [22] 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 [43], and runs QEMU (v0.13.0) and a modified `libvirt` [38] (v0.8.6). The virtual machines (guests) run 32-bit Linux (2.6.34-gentoo-r6), extended with a very small patch and kernel module (for the OS observer).

and network observer?

5.1 How fast is Falcon?

Method We compare the detection time of Falcon to that of *baseline* failure detectors (FDs), under the failures in the panel.

Figure 9 describes the baselines. These FDs are used in production or deployed systems (the Cassandra key-value store [16] uses a ϕ-accrual FD, fixed timeouts are ubiquitous, etc.); we borrow the code to implement them from [51]. All of these FDs work as follows: the client pings the target according to a fixed *ping interval* parameter p, and if the client has not heard a response by a *deadline*, the client declares a failure. We define the *timeout* 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 to adjust 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

baseline FD	timeout (ms)	error	parameters
static timer	10,000	0.0	timer $= 10,000$
Chen [20]	5,001	0.0	$\alpha = 1$ ms
Bertier [10]	5,020	0.0	$\beta = 1, \phi = 4, \gamma = 0.1$, mod_step $= 0$
ϕ-accrual [28]	4,946	0.01	$\phi = 0.4297$
ϕ-accrual [28]	4,995	0.001	$\phi = 0.4339$

Figure 9—Baseline failure detectors that we compare to Falcon. The implementations are from [51]. We set their ping intervals as $p = 5$ sec, which is aggressive and makes the comparison pessimistic. For all but 'static timer', the timeout value 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 & Bertier have no observable error.

is pessimistic for Falcon, as this setting allows the baselines to detect failures much more quickly than they would in data center applications, where ping intervals are tens of seconds [13, 27, 30], as noted in the introduction. Likewise, we configure the ϕ-accrual failure detector to yield 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 computed timeout and hence 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 often experiment with a UFD version of Falcon called *Falcon-NoBeak*, which is identical to Falcon except that it does not kill.

Each experiment holds constant the FD and the failure from the panel. Each experiment has 25–50 iterations. In each iteration, for a baseline FD, we choose the failure time uniformly at random in $[t, t + p]$, where t is when the iteration begins. For Falcon, we choose a time uniformly at random in $[t + 4.0, t + 4.1]$ seconds, where t is the time that the client library registers for callbacks;

Clarify closed loop (meaning client issues a request only after getting a reply).

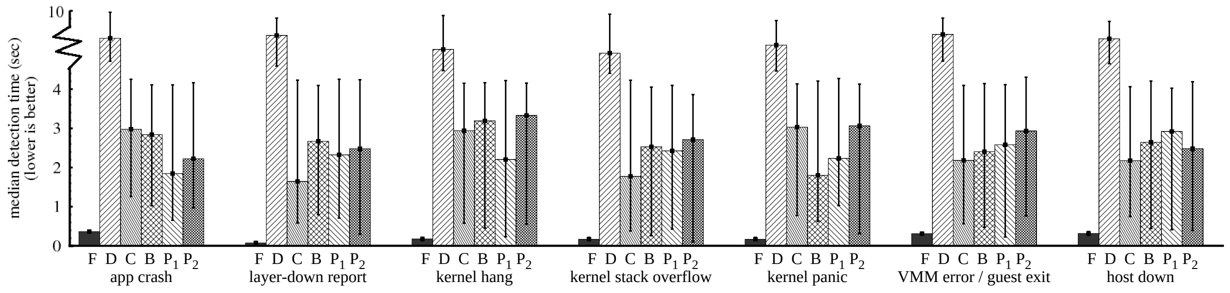

Figure 10—Detection time of Falcon (F) and baseline failure detectors under various failures. The baselines are 'static timer' (D), Chen (C), Bertier (B), ϕ-accrual with 0.01 error (P_1), and ϕ-accrual with 0.001 error (P_2); see Figure 9 for details. Rectangle heights depict medians, bars depict 10th and 90th percentiles, and the scale of 'static timer' is compressed (its medians range from 6.7 to 8.2). 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.

this approach gives the client library 4 seconds to register and sets the failure randomly in any given spy's 100 ms polling interval (§4). To produce a failure, the FD client sends an RPC to one of the *failure servers* that we deploy at different layers on the target.

We are interested in *detection time*: the duration from when the failure happens to when the FD declares the error. 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 detection time. However, we verified (through separate experiments with synchronized clocks) that the measurement error is 2–3 orders of magnitude smaller than the detection times.

This thrust of our evaluation is incomplete in two respects. First, we present numbers only for uncongested network delays; partial results with injected delays indicate similar conclusions. Second, we did not implement end-to-end timeouts as part of Falcon's interface, so for now we simulate this function in the client of Falcon.

Experiments and results We measure the detection times of the baseline FDs and of Falcon-NoBeak, for a range of failures.| Under constant network delay, we expect the baseline FDs' detection times to be uniformly distributed over $[T-p+d, T+d]$, where T is the timeout and d is the network delay (0 in our runs).[3] We hypothesize that Falcon's detection times will be on the order of 100 ms, given spies' periodic checks (§4).

if time: explain what layer-down from app-observer means in the context of the baselines

Figure 10 depicts the 10th, 50th, and 90th percentile detection times. The baselines behave as expected. (We do not report 1st and 99th percentiles because we have not yet gathered hundreds of samples per configuration; for the same reason, the medians vary noticeably.)

For application crashes, Falcon's median detection time is larger than we had expected: 363 ms. However, we realized that the Java Virtual Machine (JVM) has "exit latency", which we verified to be several hundred milliseconds on average (by an experiment that crashes a Java application 1000 times).

For the failure in which the app observer reports LAYER_DOWN, Falcon's median detection time is 73 ms. This is in line with expectations: `handler` polls the app observer 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 175 ms, 167 ms, and 167 ms, respectively. These measurements are also in line with the expected value, which is 150 ms plus processing delays. The reason is that the OS enforcer polls every $T_{OS\text{-}check} = 100$ ms to see if there has been OS activity in the *prior* interval (§4). Since the failure should be uniformly distributed over an interval (during which there would have been *some* activity), the expected detection time is the duration from the failure until the end of the prior interval (50 ms, in expectation) plus the time until the OS enforcer sees no activity (another 100 ms).

For the VMM and host crashes, Falcon's median detection times are 309 and 314 ms, respectively. This was higher than our expectation of 150 ms plus processing delays (as above). On investigation, we learned that the extra time derives from communication between threads in the VMM enforcer, which runs on the switch, which sports a somewhat antique threading package.

Falcon's detection time is an order of magnitude faster than the baselines', 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 (which would be more

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

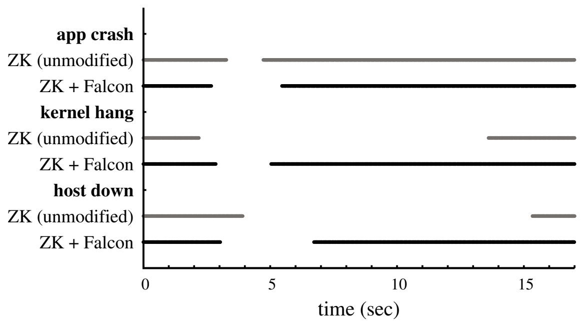

Figure 11—Availability of ZooKeeper [29] (ZK) unmodified and with Falcon under injected failures at the leader. Each dot is a response to a client; the gaps are periods of unavailability. Depicted are the smallest gap for unmodified ZooKeeper and the largest for Falcon, making the comparison pessimistic. In unmodified ZK, followers quickly detect application crashes but not kernel- or host/VMM-level crashes. Under the latter types, Falcon reduces ZooKeeper's unavailability by a factor of 4 to 7, depending on how quickly the client recovers.

realistic), the baselines' detection times would be even worse. Similarly, we expect the network delay and variation experiments to favor Falcon. Under network variation, the baselines, except for 'static timer', ought to increase their timeout, whereas Falcon ought to continue to detect crashes quickly, improving its relative performance. Under constant but non-zero network delay *d*, both the baselines and Falcon should see an increase of *d* in detection time.

5.2 What is Falcon's effect on availability?

We now ask about the effect of improved detection time on system availability. To investigate this effect empirically, we incorporate Falcon into two complex applications: ZooKeeper [29] (ZK) and a Paxos-based [33] replication library [41] (PMP). The modifications are straightforward: tens of lines of Java and 100 lines of C, respectively. We compare these systems' availability, in the case of a crash, with the availability of their unmodified versions. Both ZK and PMP, being Paxos variants, are designed to tolerate failure detector errors, so we run with Falcon-NoBeak.

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 `create()`s and `delete()`s) when it gets a response to its last one. 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. We record the time of each response.

Figure 11 depicts the results for the *shortest* period of unavailability (of the 10) for unmodified ZooKeeper and the *longest* (of 10) for ZooKeeper with Falcon. Under application failures, ZK reacts quickly because the leader's host closes the transport session with the follower, causing an exception at the follower, allowing the follower to initiate leader election. Under kernel and VMM/host failures, however, the ZK follower receives no word that the system is leaderless. Thus, the ZK follower recovers only after 10 seconds (the default ZK configuration) of not having heard from the leader. MW: Need to be clearer about the purpose of the GSoC FDs. Integrate this point: "the FDs implemented in [51] are used for leaders to monitor followers, which is not the current experiment". Under all failures, Falcon notifies the follower quickly. However, even under Falcon, there is noticeable unavailability because the ZK *client*, which is not running Falcon, connects to the follower only periodically.

We run PMP in analogous configurations and subject it to the same faults. For space, we omit the graph, but the results are similar (tens of seconds of unavailability without Falcon; less than a second with Falcon).

injected failure	**component killed by Falcon**
app crash	none
app layer-down report	app enforcer kills application
app observer 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	none
host down	VMM enforcer kills VMM/host
crashed `handlerd`	OS enforcer kills guest OS
crashed `libvirtd`	VMM enforcer kills VMM/host

Figure 12—Components killed by Falcon under various failures. By comparison, STONITH kills entire machines (or virtual machines [4]).

5.3 How disruptive is Falcon?

Next, we evaluate whether Falcon achieves the "little disruption" property, which has two aspects. First, *if* Falcon needs to kill, it should kill the smallest possible component. Second, Falcon should not kill if not required (e.g., if the target is momentarily slow). To evaluate the first aspect, we run Falcon against our failure panel, reporting the component killed, if any. Figure 12 depicts the results. By comparison, a common approach to reliability, STONITH, kills the entire machine (though some implementations can target the virtual machine [4]).

To evaluate the second aspect, we temporarily subject the target to various loads that prevent it from responding to pings, modeling a target that is temporarily slow and thus would have been declared failed by the baselines (or killed by STONITH), but would recover before Falcon's large end-to-end timeout kills. As an example, we temporarily run a process with priority lower than `handler` but higher than a monitored application, preventing the monitored application from running. The baselines declare a timeout. Falcon does not: `handler` correctly infers that the application is starved (and could respond if given CPU cycles). The mechanism for this inference is $T_{app\text{-}resp}$ (§4): a timeout based on CPU, not actual, time—an example of inside information. As another example, we temporarily disconnect a target virtual machine from the network, causing the baselines to declare failure. Not surprisingly, Falcon does not kill (because it waits longer). Of course, under other stresses, such as fork bombs with no end, Falcon's end-to-end timeout (and the baselines) catch the problem. MW: Justify some of these claims by inserting time series from inquiry #7? see commented-out latex.

MW: Two things we might want to address here:

1. What about things that Falcon does NOT correctly pick up on? (False failures that cause Falcon to declare failure.)

2. How often does Falcon wrongly declare a failure when a baseline would not have? There's commented out latex source that says that we didn't see this behavior in our experiments, but that sounds lame. Perhaps we should think about a way to answer this question?

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 following one.

Falcon's main computational cost is CPU cycles, to execute periodic local checks (described in Section 4). To assess this overhead, we run a Falcon-enabled target for 15 minutes, inducing no failures. We then run the target with the Falcon components disabled. In both cases, we take one-second samples of CPU cycles consumed over the preceding second. For each run, we take the average of its one-second samples; we report the difference of the two averages as the CPU overhead of Falcon. To estimate the cost per enforcer check, one can divide the reported cost by 10, since our experiments run with $T_{app\text{-}check} = T_{OS\text{-}check} = T_{VMM\text{-}check} = 100$ ms. MW: Need better stats with standard deviations. multiple 15 minute runs. hypothesis: very little variation. MW: numbers for the switch seem too good to be true.

Figure 13 depicts the results. There is CPU overhead, but it is small. Moreover, the numbers overestimate Falcon's CPU cost: our Falcon-disabled virtual machine (unrealistically) does not run an application so is rarely scheduled,| but the Falcon-enabled virtual machine is scheduled (because of Falcon's checks).

reviewers 75D and F ask us to run a minimal application. reviewer 75C asks something similar so we can see the overhead when running a realistic app, not just the idle VM measurements.

Falcon also has a network cost: every $T_{VMM\text{-}check} = 100$ ms time units, the switch sends a small packet to

spy component(s)	**CPU overhead (% of a core's cycles)**
app observer (in app)	0.02
app-enforcer + OS observer (in `handler`)	1.63
OS enforcer + VMM observer (in VMM)	8.42
VMM enforcer (on switch)	0.00

Figure 13—Background CPU overhead of our Falcon implementation, in which each enforcer performs a local check 10 times per second. In our current implementation, the app-enforcer and part of the OS observer run in the same process (`handler`); similarly, the OS enforcer and VMM observer are implemented at the same layer (see Figure 6). The switch's measured CPU overhead is less than one part in 10,000 so displays as 0. The VMM overhead is an overestimate (see text).

module (§4)	**spy component (§4)**	**lines of code**
platform-independent modules		
thread in app; glue (C++)	app observer	62
thread in app; glue (Java)	app observer	171
`handler`	app enforcer	783
client library	client library	968
client library glue (Java)	client library	301
platform-independent total		**2285**
platform-specfic modules		
incrementer	OS observer	9
kernel module	OS observer	50
`libvirt` extensions	OS enforcer	536
`libvirtd` extensions	VMM observer	156
DD-WRT extension	VMM enforcer	658
platform-specific total		**1409**
application-specific modules		
`f()` for Paxos application	app observer	17
`f()` for primary-backup app.	app observer	40
`f()` for ZooKeeper	app observer	5

Figure 14—The modules in our Falcon implementation. The platform-independent modules assume a POSIX system.

the target and waits for a reply. While this cost is greater than that of the baselines' pings, it is a very small percentage of the network's capacity (12 parts in 100,000 for Gigabit Ethernet, where the minimum frame size is 512 bytes). Moreover, our VMM spy is currently unoptimized. Ideally, it would infer aliveness from network traffic, consuming network bandwidth only if there is no other traffic between the target and the switch (§4).|

issue: what about the pings of the switch spy, assuming we keep the switch spy?

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 can also provide a more radical benefit to the applications that use it: shedding complexity. The cost is some platform-specific logic. However, this is not "moving code around": the platform-specific logic 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).

To understand the trade-off, we first assess the code

replication approach	lines of code	replica overhead
Paxos (from [41])	1822	3×
Paxos (from [41]), with Falcon	2047	3×
Primary-backup, with Falcon	1359	2×

Figure 15—Comparison of two different approaches to replicating state machines: Paxos [33], as implemented in [41], and primary-backup [7], as implemented by us. Numbers for Paxos do not include generated RPC code. The primary-backup approach is fewer lines of code because it is simpler, not being designed to tolerate unreliable failure detection. Primary-backup also has 50% lower replication overhead in the usual case.

introduced by Falcon. Figure 14 tabulates the lines of code (according to [50]) in our implementation (§4). The platform-specific total is fewer than 1500 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. As noted in Section 2.2, if an application does not have reliable failure detection, then it needs complex asynchronous algorithms, such as Paxos [33]. However, if it does have an RFD (as provided by Falcon), it can use primary-backup [7]. Figure 15 lists lines of code (again using [50]) in both PMP [41] (described in §5.2) and a primary-backup replication library that we implemented, which uses Falcon. The difference in lines of code is only 463, but this is 25% of the original code base. And the percentage may be deceptively low: using Paxos in a real system can require intricate engineering [17], 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 while primary backup requires two.

Assessing Falcon's reliability. The benefits in this section result only if Falcon is truly reliable, meaning that it reports DOWN only if the target is down (perhaps because of a kill). Falcon's spies are carefully designed and implemented *not* to violate this property, and in our experience, Falcon has never reported DOWN when a target is up. However, we cannot fully guarantee reliability without a formal verification that our implementation is a refinement of its specification. But this is an instance of a general problem (shared by ZK too!): ensuring that a distributed systems implementation follows its algorithm.

6 Limitations and discussion

MW: add a few lines of preamble to this section.

MW: may want to flesh out this section or split it into a section on limitations and then a discussion section. it feels a bit abrupt right now.

MW: worth adding a few lines as an "alternatives" paragraph in which we explain why there are two backstops in the system? (E2E timeouts, interlocked monitoring.) This is the hyena-vs-falcon discussion. maybe that goes in S3?

Inter data center monitoring. We assumed that the client library and the monitoring target are in the same data center. If not, Falcon will communicate across slower and less-reliable links. This does not affect correctness, but Falcon blocks if the client library cannot communicate with the target's switch, which could happen more frequently than within a data center.

Virtual machines. Falcon currently relies on virtualization, since the OS enforcer is deployed in a VMM. However, Falcon's general architecture does not require virtual machines, and we believe that it would be straightforward to implement the OS enforcer in the network driver, where it would be triggered on packet arrival.

Scalable monitoring. Our focus has been one process monitoring another, but our system also works if $n > 1$ processes monitor each other. However, for large n, the $O(n^2)$ pairs might consume too many network resources. Different techniques may be needed for this case, such as gossiping [45], possibly combined with spies. This combination is new and could be the subject of its own paper.

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.|

a line to address the question, "how do we get all of the VMM enforcers to agree to disconnect the target's host?" note that even if the VMM enforcers are unresponsive to a kill request, the RFD would possibly kill (by disconnecting) the VMM enforcer. But check.

Network failure localization. Falcon currently blocks if network problems prevent the client library from communicating with the target's switch. It might be desirable instead to localize the network problem to an appropriate network element and, if the problem is serious, disable the network element. This could be achievable via SNMP and spies in the network, but a complete solution would again be its own paper.

The end-to-end argument does not apply to Falcon because, to detect failures quickly, one requires inside information that is available only within the layers.

Local vs. end-to-end timeouts. Falcon uses local timeouts within each layer. The rationale is that reasoning about time locally is much easier than doing so over a combination of layers.

Slow processes. MW: this paragraph is optional. may want to flesh out the point and put it in a discussion section. One might wonder, "why distinguish slow from crashed processes, given that slow processes can be disruptive... | Falcon does not report slow processes as DOWN,| because (1) slowness is different from crashes,

MW: rewrote, please check, possibly pull this out, too dangerous

Falcon does sometimes do so.....in terms of striking this...what was the larger point of this paragraph? that Falcon is a finely honed instrument?

as slowness could be *temporary* (due to swapping, load spikes, local timeouts, intense disk or network I/O, etc), and (2) applications should have the choice of what to do when a process is slow, rather than having the choice made for them by Falcon. Note that slowness can be sensed easily using a wall clock.

Falcon and asynchronous systems. In asynchronous systems subject to failures, consensus and hence replication cannot be implemented [26]. Since Falcon can be used to solve consensus, Falcon cannot work in such systems. In practice, that implies that Falcon is susceptible to blocking when delays are unexpectedly large (e.g., the network partitions). This is not a limitation specific to Falcon: any RFD would be subject to such problems.

Spy quality. Falcon is only as good as its spies. A spy monitors for common failures in its layer but may not catch all failures. Falcon can tolerate such mistakes by falling back to an end-to-end timeout, which leads to a large detection time. Nevertheless, Falcon is strictly better than just using end-to-end timeouts because it can detect many common failures much faster. Indeed, our experiments used synthetic failures that represent many real failures, and Falcon detects them quickly.

Postmortem. This paper is about failure detectors; what to do after a failure is a well studied problem beyond our scope. Options include recovery, failover, etc.

7 Related work

Before describing other approaches to failure detection, we begin with 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 [18]. 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 [6]. Despite this body of theory, it was not known how to build an inexpensive failure detector that is reliable, fast, and minimally disruptive (Section 2.1), so we organize related work in terms of the trade-offs among these characteristics.

We begin with unreliable FDs. Chen et al. [20] propose a failure detector based on freshness points and end-to-end timeouts, where the value is chosen adaptively based on measurements of delay and loss. Such end-to-end timeouts could be set using other techniques, as described by Bertier et al [10]. These approaches provide a binary indication of failure. Accrual failure detectors [28], 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 [45]. 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 speed versus disruption, as what used to be false FD suspicions become needless kills. Such reliable failure detectors can be implemented using watchdogs [25], where the watchdog resets the machine based on an end-to-end timeout. Likewise, the Linux-HA project [5] 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, using either messages or shared storage. Similarly, with virtual synchrony [11] 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. [15] articulated the benefits of surgical killing (faster recovery time, less disruption), and we concur.| However, that work focuses on application components, and it solves an orthogonal problem: making failures less likely (via frequent reboot of fine-grained components, transparent to the rest of the system). Our work is complementary; it concerns reacting to observable failures.| Fine-grained information is used in cluster monitoring, which collects information about the current condition of hosts in a cluster (e.g., [1–3]), possibly using application-specific data (load, queue lengths, etc.). Unlike Falcon's spies, these services peek inside only one layer (the application), do not provide a failure detector, and do not have a license to kill (which is needed to get an RFD). Moreover, these services monitor machines using ping or SMTP messages, together with an end-to-end timeout.

clarify, per reviewer 75C.

do we need mary baker's work? is recovery box about detection or recovery? MKA: it's about recovery. Done in 1992. Could recover a DB manager in 6 seconds, and a file server in 26 seconds. We could mention this in the full version

Production services in data centers also use end-to-end timeouts for failure detection. Recent documented examples include GFS [27], Chubby [13], BigTable [19], and Dynamo [24]. MW: explain usage in detail as below; see commented-out latex below. MW: add MapReduce All of these systems might benefit from replacing their failure detectors with Falcon.

MW: Something about Chain replication [46] and CRAQ, as being in the style of primary-backup and hence as possibly gaining utility from our RFD? note that chain replication uses Paxos to replicate a master; the master's job is to tell the clients which node is currently the primary. MKA: seems second order, add if space

MW: other citations, if we want a citations list that appears more recent (we may decide to punt this)

- toueg-schiper. MKA: not extremely relevant. They do have unpublished stuff (a software library), but I'm not sure we need to reference it. FD implementation not documented. Existing paper on leader election.
- mencius [40] MKA: more paxos stuff
- bolosky's Paxos thing [12]
- add year fetched to our URLs, so the citations list "feels" more up-to-date

8 Conclusion

MW: State where to get the code and kernel configurations for the VMs and hypoervisors

References

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

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

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

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

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

[6] M. K. Aguilera, G. L. Lann, and S. Toueg. On the impact of fast failure detectors on real-time fault-tolerant systems. In *Proc. Int. Conf. on Distributed Computing*, pages 354–370, Oct. 2002.

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

[8] Anonymous. Position paper anonymized for blind review.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, Oct. 2003.

[10] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *DSN*, pages 354–363, June 2002.

[11] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP*, pages 123–138, Nov. 1987.

[12] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *NSDI*, Apr. 2011.

[13] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, pages 335–350, Dec. 2006.

[14] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive microreboots: A soft-state system case study. *Performance Evaluation Journal*, 56(1–3), Mar. 2004.

[15] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot—a technique for cheap recovery. In *OSDI*, pages 31–44, Dec. 2004.

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

[17] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *PODC*, pages 398–407, Aug. 2007.

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

[19] 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 *OSDI*, pages 205–218, Nov. 2006.

[20] 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.

[21] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, pages 161–174, Apr. 2008.

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

[23] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, Dec. 2004.

[24] 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 *SOSP*, pages 205–220, Oct. 2007.

[25] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Trans. on Computers*, 52:99–112, Feb. 2003.

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

[27] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, pages 29–43, Oct. 2003.

[28] N. Hayashibara, X. Défago, R. Yared, and T. Katayama. The ϕ accrual failure detector. In *Proc. IEEE Symposium on Reliable Distributed Systems*, 2004.

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

[30] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, Mar. 2007.

[31] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani. Consensus routing: The Internet as a distributed system. In *NSDI*, pages 351–364, Apr. 2008.

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

[33] L. Lamport. The part-time parliament. *TOCS*, 16(2):133–169, May 1998.

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

[35] B. Lampson. The ABCD's of Paxos. In *PODC*, 2001.

[36] E. K. Lee and C. Thekkath. Petal: Distributed virtual disks. In *ASPLOS*, pages 84–92, Dec. 1996.

[37] H. C. Li, A. Clement, A. S. Aiyer, and L. Alvisi. The Paxos register. In *Proc. IEEE Symposium on Reliable Distributed Systems*, pages 114–126, Oct. 2007.

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

[39] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for

storage infrastructure. In *OSDI*, pages 105–120, Dec. 2004.

[40] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for WANs. In *OSDI*, pages 369–384, Dec. 2008.

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

[42] R. D. Prisco, B. Lampson, and N. Lynch. Revisiting the PAXOS algorithm. *Theoretical Computer Science*, 243(1–2):35–91, July 2000.

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

[44] 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 *NSDI*, pages 43–58, Apr. 2009.

[45] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proc. ACM/IFIP/USENIX International Middleware Conference*, pages 55–70, Sept. 1998.

[46] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, pages 91–104, Dec. 2004.

[47] P. Veríssimo. Uncertainty and predictability: Can they be reconciled? In *Future Directions in Distributed Computing*, pages 108–113. Springer-Verlag LNCS 2584, May 2003.

[48] P. Veríssimo and A. Casimiro. The Timely Computing Base model and architecture. *IEEE Trans. on Computers*, 51(8):916–930, Aug. 2002.

[49] P. Veríssimo, A. Casimiro, and C. Fetzer. The Timely Computing Base: Timely actions in the presence of uncertain timeliness. In *DSN*, pages 533–542, June 2000.

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

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