FUSE: Lightweight Guaranteed Distributed Failure Notification

John Dunagan* Nicholas J. A. Harvey‡ Michael B. Jones* Dejan Kostić†
Marvin Theimer* Alec Wolman*

"I am not discouraged, because every failure is another step forward." – Thomas Edison

Abstract

FUSE is a lightweight failure notification service for building distributed systems. Distributed systems built with FUSE are guaranteed that failure notifications never fail. Whenever a failure notification is triggered, all live members of the FUSE group will hear a notification within a bounded period of time, irrespective of node or communication failures. In contrast to previous work on failure detection, the responsibility for deciding that a failure has occurred is shared between the FUSE service and the distributed application. This allows applications to implement their own definitions of failure. Our experience building a scalable distributed event delivery system on an overlay network has convinced us of the usefulness of this service. Our results demonstrate that the network costs of each FUSE group can be small; in particular, our overlay network implementation requires no additional liveness-verifying ping traffic beyond that already needed to maintain the overlay, making the steady state network load independent of the number of active FUSE groups.

1 Introduction

This paper describes FUSE, a lightweight failure notification service. When building distributed systems, managing failures is an important and often complex task. Many different architectures, abstractions, and services have been proposed to address this [5, 10, 13, 28, 30, 38, 42, 43, 44]. FUSE provides a new programming model for failure management that simplifies the task of agreeing when failures have occurred in a distributed system, thereby reducing the complexity faced by application developers. The most closely related prior work on coping with failures has centered around *failure detection services*. FUSE takes a somewhat different approach, where detecting failures is a shared responsibility between FUSE and the application. Applications create a FUSE group with an immutable list of participants. FUSE monitors this group until either FUSE or the application decides to terminate the group, at which point all live participants are guaranteed to learn of a "group failure" within a bounded period of time. This focus on delivering failure notifications leads us to refer to FUSE as a *failure notification service*. This is a very different approach than prior systems have adopted, and we will argue that it is a good approach for wide-area Internet applications.

Applications make use of the FUSE abstraction as follows: the application asks FUSE to create a new group, specifying the other participating nodes. When FUSE finishes constructing the group, it returns a unique identifier for this group to the creator. The application then passes this FUSE ID to the applications on all the other nodes in this group, each of which registers a callback associated with the given FUSE ID. FUSE guarantees that every group member will be reliably notified via this callback whenever a failure condition affects the group. This failure notification may be triggered either explicitly, by the application, or implicitly, when FUSE detects that communication among group members is impaired.

Applications can create multiple FUSE groups for different purposes, even if those FUSE groups span the same set of nodes. In the event that FUSE detects a low-level communication failure, failure will be signalled on all the FUSE groups using that path. However, on any individual FUSE group the application may signal a failure without affecting any of the other groups.

FUSE provides the guarantee that notifications will be delivered within a bounded period of time, even in the face of node crashes and arbitrary network failures. We refer to these semantics as "distributed one-way agreement". One-way refers to the fact that there is only one transition any group member can see: from "live" to "failed". After failure notification on a group, detecting future failures requires creating a new group.

By providing these semantics, FUSE ensures that *failure notifications never fail*. This greatly simplifies failure handling among nodes that have state that they want to handle in a coordinated fashion. FUSE efficiently handles all the corner cases of guaranteeing that all members will be notified of any failure condition affecting the group. Applications built on top of FUSE do not need to worry

[*] Microsoft Research, Microsoft Corporation, Redmond, WA. {jdunagan, mbj, theimer, alecw}@microsoft.com

[†] Department of Computer Science, Duke University, Durham, NC. dkostic@cs.duke.edu

[‡] Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA. nickh@mit.edu

that a failure message did not get through or that orphaned state remains in the system.

The primary target for FUSE is wide-area Internet applications, such as content delivery networks, peer-to-peer applications, web services, and grid computing. FUSE is not targeted at applications that require strong consistency on replicated data, such as stock exchanges and missile control systems. Techniques such as virtual synchrony [6], Paxos [29], and BFT [10] have already proven to be effective in such environments. However, these techniques incur significant overhead and therefore have limited scalability. Furthermore, FUSE does not provide resilience to malicious participants, though it also does not preclude solving this problem at a higher layer.

Previous work on failure detection services uses "membership" as the fundamental abstraction: these services typically provide a list of each component in the system, and whether it is currently up or down. Membership services have seen widespread success as a building block for implementing higher level distributed services, such as consensus, and have even been deployed in such commercially important systems as the New York Stock Exchange [5]. However, one disadvantage of the membership abstraction is that it does not allow application components to have failed with respect to one action, but not with respect to another. For example, suppose a node is engaged in an operation with a peer and at some point fails to receive a timely response. The failure management service should support declaring that this operation has failed without requiring that a node or process be declared to have failed. The FUSE abstraction provides this flexibility: FUSE tracks whether individual application communication paths are currently working in a manner that is acceptable to the application.

The scenario described above is more likely to occur with wide-area Internet applications, which face a demanding operating environment: network congestion leads to variations in network loss rate and delay; intransitive connectivity failures (sometimes called partial connectivity failures) occur due to router or firewall misconfiguration [27, 31]; and individuals network components such as links and routers can also fail. Under these conditions, it is difficult for applications to make good decisions based solely on the information provided by a membership service. A particular scenario illustrating why applications need to participate in deciding when failure has occurred is delivery of streaming content over the Internet: failure to achieve a certain threshold bandwidth may be unacceptable to such an application even though other applications are perfectly happy with the connectivity provided by that same network path.

Our FUSE implementation is scalable, where scaling is with respect to the number of groups: multiple failure notification groups can share liveness checking messages. Other implementations of the FUSE abstraction may sacrifice scalability in favor of increased security. Our FUSE implementation is designed to support large numbers of small to moderate size groups. We do not attempt to efficiently support very large groups because we believe very large groups will tend to suffer from too-frequent failure notification, making them less useful.

Our FUSE implementation is particularly well-suited to applications using scalable overlay networks. Scalable overlay networks already do liveness checking to maintain their routing tables and FUSE can re-use this liveness checking traffic. In such a deployment, the network traffic required to implement FUSE is independent of the number of groups until a failure occurs; only creation and teardown of a group introduce a per-group overhead. FUSE can be implemented in the absence of a pre-existing overlay network – the implementation can construct its own overlay, or it can use an alternative liveness checking topology.

We have implemented FUSE on top of the SkipNet [25] scalable overlay network, and we have built a scalable event delivery application using FUSE. We evaluated our implementation using two main techniques: a discrete event simulator to evaluate its scalability, and a live system with 400 overlay participants (each running in its own process) running on a cluster of 40 workstations to evaluate correctness and performance. Both the simulator and the live system use an identical code base except for the base messaging layer. Our live system evaluation shows that our FUSE implementation is indeed lightweight; the latency of FUSE group creation is the latency of an RPC call to the furthest group member; the latency of explicit failure notification is similarly dominated by network latency; and we show that our implementation is robust to false positives caused by network packet loss.

In summary, the key contributions of this paper are:

- We present a novel abstraction, the FUSE failure notification group, that provides the semantics of distributed one-way agreement. These are desirable semantics in our chosen setting of wide-area Internet applications.
- We used FUSE in building a scalable event delivery service and we describe how it significantly reduced the complexity of this task.
- We implemented FUSE on top of a scalable overlay network. This allowed us to support FUSE without adding additional liveness checking. We experimentally evaluated the performance of our implementation on a live system with 400 virtual nodes.

2 Related Work

Failure detection has been the subject of more than two decades of research. This work can be broadly classified into unreliable failure detectors, weakly consistent membership services, and strongly consistent membership services. Unreliable failure detectors provide the weakest semantics directly, but they are a standard building block in constructing membership services with stronger semantics. Both weakly-consistent and strongly-consistent membership services are based on the abstraction of a list of available or unavailable components (typically processes or machines). In contrast, a FUSE group ID is not bound to a process or machine and hence can be used in many contexts. For example, it can correspond to a set of several processes, or to related data stored on several different machines. This abstraction allows FUSE to provide novel semantics for distributed agreement, a subject we will elaborate on in our discussion of weakly consistent membership services.

Chandra et al. formalized the concept of unreliable failure detectors, and showed that one such detector was the weakest failure detector for solving consensus [11, 12]. Such detectors typically provide periodic heartbeating and callbacks saying whether the component is "responding" or "not responding." These callbacks may be aggregate judgments based on sets of pings. Unreliable failure detectors provide the following semantic guarantee: fail-stop crashes will be identified as such within a bounded amount of time. There has been extensive work on such detectors, focusing on such aspects as interface design, scalability, rapidity of failure detection, and minimizing network load [1, 17, 21, 23, 38].

FUSE uses a similar lightweight mechanism (periodic heartbeats) to unreliable failure detectors, but provides stronger distributed agreement semantics. Unreliable failure detectors are typically used as a component in a membership service, and the membership service is responsible for implementing distributed agreement semantics.

Weakly consistent membership services have also been the subject of an extensive body of work [2, 19, 42, 44]. This work can be broadly classified as differing in speed of failure detection, accuracy (low rate of false positives), message load, and completeness (failed nodes are agreed to have failed by everyone in the system). Epidemic and gossip-style algorithms have been used to build highly scalable implementations of this service [19, 42]. Previous application areas that have been proposed for such membership services are distributed collaborative applications, online games, large-scale publish-subscribe systems, and multiple varieties of Web Services [5, 19]. These are the same application domains targeted by FUSE, and FUSE has similar overhead to a weakly consistent membership service in these settings. Typical characteristics of such applications are that many operations are idempotent or can be straightforwardly undone, operations can be re-attempted with a different set of participants, or the decision to retry can be deferred to the user (as in an instant messaging service).

One novel aspect of the FUSE abstraction is the ability to handle arbitrary network failures. In contrast, weakly consistent membership services provide semantic guarantees assuming only a fail-stop model. One kind of network failure where the FUSE abstraction is useful is an intransitive connectivity failure: A can reach B, B can reach C, but A cannot reach C. This class of network failures is hard for a weakly consistent membership service to handle because the abstraction of a membership list limits the service to one of three choices, each of which has drawbacks:

- Declare one of the nodes experiencing the intransitive connectivity failure to have failed. This prevents the use of that node by any node that *can* reach it.
- Declare all of the nodes experiencing the intransitive failure to be alive because other nodes in the system can reach them. This may cause the application to block for the duration of the connectivity failure.
- Allow a persistent inconsistency among different nodes' views of the membership list. This forces the application to deal with inconsistency explicitly, and therefore the membership service is no longer reducing the complexity burden on the application developer.

FUSE appropriately handles intransitive connectivity failures by allowing the application on a node experiencing a failure to declare the corresponding FUSE group to have failed. Other FUSE groups involving the same node but not utilizing a failed communication path can continue to operate. Application participation is required to achieve this because FUSE may not have detected the failure itself; like most weakly consistent membership services, FUSE typically monitors only a subset of all application-level communication paths.

FUSE would require application involvement in failure detection even if it monitored all communication paths. Consider a multi-tier service composed of a front-end, middle-tier, and back-end. Suppose the middle-tier component is available but misconfigured. FUSE allows the front-end to declare a failure, and to then perform appropriate failure recovery, such as finding another middle-tier from some pool of available machines. This difference in usage between membership services and FUSE reflects a difference in philosophy. Membership services try to proactively decide at a system level whether or not nodes and processes are available. FUSE provides a mechanism that applications can use to declare failures when application-level constraints (such as configuration) are violated.

Another contrast between the two approaches is that the FUSE abstraction enables "fate-sharing" among distributed data items. By associating these items with a single FUSE group, application developers can enforce that invalidating any one item will cause all the remaining data items to be invalidated. Weakly consistent membership services do not explicitly provide this tying together of distributed data.

Strongly consistent membership services share the abstraction of a membership list, but they also guarantee that all nodes in the system always see a consistent list through the use of atomic updates. Such membership services are an important component in building highly available and reliable systems using virtual synchrony. Notable examples of systems built using this approach are the New York and Swiss Stock Exchanges, the French Air Traffic Control System, and the AEGIS Warship [2, 5, 6]. However, a limitation of virtual synchrony is that it has only been shown to perform well at small scales, such as five node systems [6].

Some network routing protocols, such as IS-IS [8], OSPF [33], and AutoNet [35], use mechanisms similar to FUSE. One similar aspect of AutoNet is its use of teardown and recreate to manage failures. Any link-state change causes all AutoNet switches to discard their link-state databases and rebuild the global routing table. OSPF and IS-IS take local link observations and propagate them throughout the network using link-state announcements. They tolerate arbitrary network failures using timers and explicit refreshes to maintain the link-state databases. FUSE also uses timers and keep-alives to tolerate arbitrary network failures. However, FUSE uses them to tie together sets of links that provide end-to-end connectivity between group members, and to provide an overall yes/no decision for whether connectivity is satisfactory.

A more distantly related area of prior work is black-box techniques for diagnosing failures. Such techniques use statistics or machine learning to distinguish successful and failed requests [9, 13, 14, 15, 16]. In contrast, FUSE assumes application developer participation and provides semantic guarantees to the developer. Another significant distinction is that black-box techniques typically require data aggregation in a central location for analysis; FUSE has no such requirement.

Distributed transactions are a well-known abstraction for simplifying distributed systems. Because FUSE provides weaker semantics than distributed transactions, FUSE can maintain its semantic guarantees under network failures that cause distributed transactions to block. Theoretical results on consensus show that the possibility of blocking is fundamental to any protocol for distributed transactions [22, 24].

Two of the design choices we made in building FUSE were also recommended by recent works dealing with the architectural design of network protocols. Ji et al. [26] surveyed hard-state and soft-state signaling mechanisms across a broad class of network protocols, and recommended a soft state approach combining timers with explicit revocation: FUSE does this. Mogul et al. [32] argued that state maintained by network protocol implementations should be exposed to clients of those protocols. As described in Section 6, we modified our overlay routing layer to expose a mechanism for FUSE to piggy-back content on overlay maintenance traffic.

3 FUSE Semantics and API

We begin by describing one simple approach to implementing the FUSE abstraction. Suppose every group member periodically pings every other group member with an "are you okay?" message. A group member that is not okay for any reason, either because of node failure, network disconnect, network partition, or transient overload, will fail to respond to some ping. The member that initiated this missed ping will ensure that a failure notification propagates to the rest of the group by ceasing to respond itself to all pings for that FUSE group. Any individual observation of a failure is thus converted into a group failure notification. This mechanism allows failure notifications to be delivered despite any pattern of disconnections, partitions, or node failures. This specific FUSE implementation guarantees that failure notifications are propagated to every party within twice the periodic pinging interval. Our implementation uses a different liveness checking topology, discussed in Section 5. It also uses a different ping retry policy; the retry policy and the guarantees on failure notification latency are discussed in Section 3.3.

The name FUSE is derived from the analogy to "laying a fuse" between the group members. Whenever any group member wishes to signal failure it can light the fuse, and this failure notification will propagate to all other group members as the fuse burns. A connectivity failure or node crash at any intermediate location along the fuse will cause the fuse to be lit there as well, and the fuse will then start burning in every direction away from the failure. This ensures that communication failures will not stop the progress of the failure notification. Also, once the fuse has burnt, it cannot be relit, analogous to how the FUSE facility only notifies the application once per FUSE group.

Many different FUSE implementations are possible. All implementations of the FUSE abstraction must provide distributed one-way agreement: failure notifications are delivered to all live group members under node crashes and arbitrary network failures. Different FUSE implementations may use different strategies for group creation, liveness checking topology, retry, programming interface, and persistence, with consequent variations in performance. In this section, we describe the choices that we made in our FUSE implementation, the resulting semantics that application developers will need to understand, and some of the alternative strategies that other FUSE implementations could use.

3.1 Programming Interface

We now present the FUSE API for our implementation. FUSE groups are created by calling *CreateGroup* with a desired set of member nodes. This generates a FUSE ID unique to this group, communicates it to the FUSE layers on all the specified members, and then returns the ID to the caller. Applications are subsequently expected to explicitly communicate the FUSE ID from the creator to the other group members. Applications learning about this FUSE ID register a handler for FUSE notifications using the *RegisterFailureHandler* function. In this design, the FUSE layer is not responsible for communicating the FUSE ID to applications on nodes other than the creator.

```
// Creates a FUSE notification group containing
// the nodes in the set
FuseId CreateGroup(NodeId[] set)

// Registers a callback function to be invoked
// when a notification occurs for the FUSE group
void RegisterFailureHandler
    (Callback handler, FuseId id)

// Allows the application to explicitly cause
// FUSE failure notification
void SignalFailure(FuseId id)
```

Figure 1. *The FUSE API*

We believe the most likely use of FUSE is to allow fate-sharing of distributed application state. Applications should learn about FUSE IDs with sufficient context to know what application state to associate with the FUSE ID. Though failure handlers can simply perform garbage collection of the associated application state, a handler is also free to attempt to re-establish the application state using a new FUSE group, or to execute arbitrary code. For brevity, we refer to all of these permissible application-level actions using the short-hand "garbage collection."

The application handler is invoked whenever the FUSE layer believes a failure has occurred, either because of a node or communication failure or because the application explicitly signalled a failure event at one of the group members. If *RegisterFailureHandler* is called with a FUSE ID parameter that does not exist, perhaps because it has already been signalled, the handler callback is invoked immediately. Applications that wish to explicitly signal failure do so by calling the *SignalFailure* function.

3.2 Group Creation

Group creation can be implemented in one of two ways: it can return immediately, or it can block until all nodes in the group have been contacted. Returning immediately reduces latency, but because FUSE has not checked that all group members are alive, the application may perform expensive operations, only to have FUSE signal failure a short time later. In contrast, blocking until all members have been contacted increases creation latency, but decreases the likelihood that the FUSE group will immediately fail. We chose to implement blocking create; this provides application developers the semantic guarantee that if group creation returns successfully, all the group members were alive and reachable. A high enough rate of churn amongst group members could repeatedly prevent FUSE group creation from succeeding. However, based on the low latency of FUSE group creation (Section 7), such a high churn rate is likely to cause the system to fail in other ways before FUSE becomes a bottleneck.

When *CreateGroup* is called, FUSE generates a globally unique FUSE ID for the group. Each node is then contacted and asked to join the new FUSE group. If any group member is unreachable, all nodes that already learned of the new FUSE group are then notified of the failure. Group members that learned of the group but subsequently become unreachable similarly detect the group failure through their inability to communicate with other group members. If the FUSE layer is successfully contacted on all members, the FUSE ID is returned to the *CreateGroup* caller.

FUSE state is never orphaned by failures, even when those failures occur just after group creation. An application may receive a FUSE ID from the group creator, and then attempt to associate a failure handler with this FUSE group, only to find out that the group no longer exists because a failure has already been signalled. This causes the failure handler to be invoked, just as if the notification had arrived after the failure handler was registered.

3.3 Liveness Monitoring and Failure Notification

Once setup is complete, our FUSE implementation monitors the liveness of the group members using a spanning tree whose individual branches follow the overlay routes between the group creator and the group members. Each link in the tree is monitored from both sides; if either side decides a link has failed, it ceases to acknowledge pings for the given FUSE group along all its links. When this occurs, one could immediately signal group failure, but our implementation instead attempts repair, as will be explained in more detail in Section 6.

This mechanism allows FUSE to guarantee that any member of the group can cause a failure notification to be received by *every* other live group member. FUSE invokes the failure notification handler exactly once on a node before tearing down the state for that FUSE group. A node hearing the notification does not know whether it was due to crash, network disconnect or partition, or if the notification was explicitly triggered by some group member. Explicit triggering is a necessary component of FUSE because FUSE does not guarantee that it will notice all persistent communication problems between group members automatically; it only guarantees that a communication failure noticed by any group member will soon be detected by all group members.

FUSE only guarantees delivery of failure notifications, and only to nodes that have already been contacted during group creation. Note that FUSE clients cannot use this mechanism to implement general-purpose reliable communication. Therefore, well-known impossibility results for consensus do not apply to FUSE [22, 24]. A concrete example illustrating this limitation is a network partition. FUSE members on both sides of the partition will receive failure notifications, but it is not possible to communicate additional information, such as the cause of the failure, across the partition.

FUSE will sometimes generate a notification to the entire group even though all nodes are alive and the next attempted communication would succeed. We refer to such a notification as a false positive. It is easy to see how false positives can occur – transient communication failures can trigger group notification. The possibility of false positives is inherent in building distributed systems on top of unreliable infrastructure. One can tune the timeout and retry policy used by the liveness checking mechanism, but there is a fundamental tradeoff between the latency of failure detection and the probability that timeouts generate false positives. We do not provide an API mechanism for applications to modify the FUSE timeout and retry policy. Providing such a mechanism would add complexity to our implementation, while providing little benefit: applications already need to implement their own timeouts, as dictated by their choice of transport layer. FUSE requires this participation from applications because FUSE does not necessarily monitor every link.

Because we implement liveness checking using overlay routes, the maximum notification latency is proportional to the diameter of the overlay: successive failures at adversarially chosen times could cause each link to fail exactly one failure timeout after the previous link. However, we expect notification after a failure to rarely require more than a single failure timeout interval because our FUSE implementation always attempts to aggressively propagate failure notifications. Also, application developers do not need to know the maximum latency in order to specify their timeouts. As mentioned above, sends should be monitored using whatever timeout is appropriate to the transport layer used by the application. If the sender times out, it can signal the FUSE layer explicitly. If a node is waiting to receive a message, specifying a timeout is difficult because only the sender knows when the transmission is initiated. In this case, if the sender has crashed, developers should rely on the FUSE layer timeout to guarantee the failure handler will be called.

FUSE failure notifications do not necessarily eliminate all the race conditions that an application developer must handle. For example, one group member might signal a failure notification, and then initiate failure recovery by sending a message to another group member. This failure recovery message might arrive at the other group member before it receives the FUSE failure notification. In our experience, version-stamping the data associated with a FUSE group was a simple and effective means of handling these races.

3.4 Fail-on-Send

FUSE does not guarantee that all communication failures between group members will be proactively detected. For example, in wireless networks sometimes link conditions will allow only small messages – such as liveness ping messages – to get through while larger messages cannot. In this case, the application will detect that the communication path is not working, and explicitly signal FUSE. We call this reason for explicitly signalling FUSE *fail-on-send*.

There are two categories of failures that require fail-on-send. The first is a communication path that successfully transmits FUSE liveness checking messages but which does not meet the needs of the application. The second is a failed communication path the application is using, but which FUSE is not monitoring.

An example from this second category is an intransitive connectivity failure. If two applications cannot communicate directly, but are both responding to FUSE messages from a third party, they may only experience a failure upon attempting to exchange a message. FUSE still guarantees that if either party triggers a notification at this point, all live group members will hear a notification.

Some applications may generate mixed acknowledged and un-acknowledged traffic. For example, an application might send streaming video over UDP alongside a control stream over TCP. In this case, it is up to the application to decide which delivery failures warrant a notification. Fail-on-send allows this failure case to be handled in the same manner as the previous cases.

3.5 Failure Model and Security

FUSE is designed to handle node crashes and arbitrary network failures, but not malicious behavior. The application we built using FUSE handles malicious behavior through redundancy above the FUSE layer by using multiple content distribution trees (see Section 4).

FUSE assumes a network failure model consisting of any pattern of packet loss, duplication or re-ordering. This includes simultaneous network partitions and even an adversary dropping packets based on their content. For any network failure, FUSE guarantees that all parties agree whether or not a failure has occurred. Our FUSE implementation routes all FUSE and overlay messages over TCP connections. Our implementation handles arbitrary packet loss and re-ordering, but only handles duplication to the extent that TCP does. It would be straightforward to extend our implementation to handle arbitrary duplication by incorporating digital signatures and timestamps, though we have not yet done so. This extension would also prevent tampering with message contents. FUSE's ability to handle packet loss is not dependent on using a reliable transport layer, such as TCP. Alternative FUSE implementations could use unreliable transport layers, such as UDP. Using a different transport would present different performance characteristics that many application developers would want to be aware of.

Our model for node failures is fail-stop. Software failures that are recognized by the application (e.g., misconfiguration is detected) can be handled by explicitly signalling the FUSE group. FUSE also handles software failures that result in a process exit, such as unhandled exceptions. FUSE does not handle nodes that behave maliciously, either due to explicit compromise or due to software faults that are not appropriately contained.

Malicious nodes can attack FUSE in one of two ways: by dropping legitimate failure notifications or by unnecessarily generating failure notifications. Dropping a failure notification, and then continuing to generate ping messages for the failed group, can delay the notification indefinitely for certain group members. This violates the FUSE notification semantics. Generating unnecessary failure notifications can prevent the use of otherwise functional FUSE groups, thus leading to a Denial-of-Service (DoS) attack. Of course, if the application response to failure notification is to re-attempt the failed operation with a different set of nodes, sustaining a DoS attack may be quite difficult.

3.6 Crash Recovery

Our implementation of FUSE does not use stable storage, and so crash recovery is trivial. The implementation performs the same actions during crash recovery as during any other initialization.

A recovering node does not know whether a failure notification was propagated to other group members. FUSE handles this case and several other corner cases by having nodes actively compare their lists of live FUSE groups as part of liveness checking. We will discuss the details of our implementation more in Section 6, but the effect is that disagreements about the current set of live FUSE groups are detected within one failure timeout interval. Disagreements are resolved by triggering a notification on any groups already considered to have failed by some group member.

An alternative FUSE implementation could use stable storage to attempt to mask brief node crashes. A node recovering from a crash could assume that all the FUSE groups in which it participates are still alive; the active comparison of FUSE IDs would suffice to reliably reconcile this node with the rest of the world. Furthermore, there is no compatibility issue: Nodes employing stable storage could co-exist with nodes not employing stable storage without any change to the FUSE semantics. It is still the case that a persistent communication failure on the node recovering from crash would cause all the FUSE groups it participates in to be notified. Applications can also make use of volatile-state FUSE groups to guard state stored in stable storage, but this requires additional application-level complexity.

4 Applications

As part of the Herald [7] project to build a scalable event notification service, we have been exploring the construction of scalable, reliable application-level multicast groups using a scalable peer-to-peer overlay network. Grappling with the complexities of implementing failure handling and automatic re-configuration led us to invent a new abstraction for failure notification.

The deciding factor for inventing FUSE was our design of multicast groups. A well-known technique for implementing application-level multicast on an overlay is to construct the multicast trees using reverse path forwarding (e.g., Scribe [37]). One major drawback of this approach is that nodes on an overlay routing path between a subscriber and the root node must forward a potentially large amount of traffic for the multicast group, even if they have no interest in it. To remove this potential obstacle to deployment, we designed Subscriber/Volunteer (SV) trees [20]. SV trees route content around non-interested parties by establishing separate content-forwarding links among subscribers and volunteers. This leads to two interrelated data structures: a content forwarding tree overlaid on a reverse path forwarding (RPF) tree.

The content-forwarding tree is straightforward to construct in the absence of failure. However, repairing the tree without introducing distributed routing cycles proved difficult in the face of arbitrary and possibly simultaneous node failures, link failures, message loss, and routing changes in the overlay. To manage this complexity, we adopted a simple design pattern: garbage collect out-of-date state using FUSE and retry by establishing a new FUSE group and installing new application-level state. FUSE allowed us to tie together all the distributed state that needed to be garbage collected.

This design pattern drastically reduced the state space that we had to consider and was instrumental in achieving a working SV tree implementation. For example, a single FUSE group ties together the endpoints of a content-forwarding link and all the RPF tree nodes bypassed by that link. Failure notification on this group garbage collects all the relevant state. After failure notification, the subscriber that requested creation of the now-failed content-forwarding link is responsible for creating a replacement FUSE group and forwarding link. If this subscriber is dead, then no replacement is needed. Indeed, there was a natural choice for the FUSE group creator everywhere we used FUSE, obviating the need for a voting mechanism to manage group creation or re-creation.

As mentioned in Section 3.3, FUSE did not eliminate all race conditions in SV trees, but the remaining ones were trivial to handle. For example, subscribers add version stamps to each subscription request to prevent late-arriving FUSE notifications from acting on new content-forwarding links.

FUSE also reduced the amount of code required to implement SV trees. Without FUSE, we would have had to include a large amount of additional context in each message to allow the recipient to garbage-collect now-invalid state and we found it difficult to reason about the correctness of this non-FUSE alternative design. We also used FUSE in one important non-failure case: when a multicast group participant voluntarily leaves the tree, we explicitly signal the FUSE group that would have been signaled if the node had failed. This causes the appropriate repairs to occur, removing the node from the content-forwarding tree.

Our desire to support large multicast trees does not require that we support individual FUSE groups with a large number of members. We designed SV trees to use a large number of small to medium size FUSE groups, and this determined the scalability requirements for FUSE. For example, simulating a 2000 subscriber tree on a 16,000 node overlay required an average of 2.9 members per FUSE group with a maximum size of 13. We also verified that the maximum and mean FUSE group sizes depend very little on the size of the multicast tree, and increase slowly with the size of the overlay.

4.1 Other Applications

From our experience implementing an event delivery service with FUSE, we believe that many applications built on top of scalable overlay networks can benefit from the use of FUSE. Many of these applications construct a large number of trees, and then monitor parent-child links in these trees using application-level heartbeats. Using FUSE for these application-level heartbeats would allow liveness checking traffic to be shared across all the trees.

Another type of application where FUSE would be useful is a Content Delivery Network (CDN) that replicates a large number of documents and pushes updates to them. If the replication topology varies on a per-document basis, this will entail a large number of replication multicast trees. A common strategy for reliability on these trees mirrors the approach discussed above for peer-to-peer applications: heartbeats ensure that each replica site for a given object can track whether it is receiving all updates correctly, or is instead somehow disconnected from the tree. FUSE can replace the per-tree heartbeat messages with a more efficient and scalable means of detecting when the trees need to be reconfigured due to node or network failures.

Peer-to-peer storage systems such as TotalRecall [4] and Om [45] could also benefit from the efficiencies of using FUSE to implement liveness checking. For example, TotalRecall relies on the overlay for liveness-checking of eager replicas, but must separately implement liveness-checking of lazy replicas. The substitution of FUSE groups would be straightforward. Om implements its own failure detection and timeout scheme using leases; these leases could be replaced by FUSE groups. FUSE would also be a good fit for Om because every replica in Om can regenerate the entire replica set, and therefore should monitor the liveness of all other replicas. This symmetric responsibility exactly corresponds to the semantics of FUSE group notifications. Lastly, the potential for false positives using FUSE does not compromise Om's consistency guarantees; Om's failure-induced reconfiguration protocol is already designed to be robust to failure-detection false positives.

In addition, FUSE may also be useful in some of the application areas targeted by weakly consistent membership services. For example, Vogels and Re [44] argue that weakly consistent membership services would benefit many Web Services, ranging from scientific computing to federated business activities. FUSE may be a more suitable choice for some of these emerging applications.

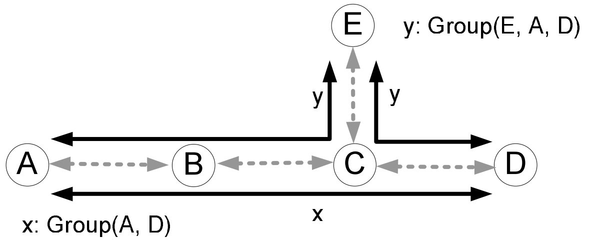

Figure 2. *Two FUSE groups being monitored by overlay pings. The black lines denote end-to-end checking, while dashed gray lines denote active overlay pings.*

5 Liveness Checking Topologies

Many different liveness checking topologies can be used to implement FUSE. In this Section, we describe in detail the topology we chose: per-group spanning trees on an overlay network. We then discuss other topologies and their security/scalability tradeoffs.

Our overlay design requires a content-addressable overlay (e.g., DHTs such as CAN, Chord, Pastry, SkipNet, or Tapestry [25, 34, 36, 39, 46]). Figure 2 depicts an overlay topology with two live FUSE groups, x and y. The spanning tree for FUSE group y contains the group *members* A, D, and E, and two additional nodes, B and C, that we refer to as *delegates*. Delegates arise because the FUSE liveness checks are routed along overlay paths between members, and these paths may contain nodes that are not members. If overlay routes change or delegates fail, delegates may be added to or deleted from the liveness checking tree; this repair process is explained in detail in Section 6.

Building liveness checking trees on top of an overlay lets us reuse the liveness checks that the overlay uses to maintain its routing tables. The liveness checking tree for a given FUSE group is the union of the overlay routes between the group creator (the *root*) and the group members. When multiple FUSE groups have overlapping trees, each overlay ping message monitors all FUSE groups whose liveness checking trees include that overlay link. Figure 2 illustrates this sharing for the FUSE groups x and y.

In the absence of failures, our FUSE implementation requires no additional messages beyond the overlay pings to monitor FUSE groups. Only group setup, teardown, and repair incur per-group costs. This allows one to build systems that require very large numbers of FUSE groups.

5.1 Alternative Topologies

In this section, we present three alternative topologies for FUSE liveness monitoring that provide better security guarantees at the cost of worse scalability. Certain implementations of these topologies can be simpler and provide stronger guarantees for worst-case failure notification latency.

As mentioned in Section 3.5, malicious nodes can mount two kinds of attacks against FUSE: the dropped notification attack and the unnecessary notification attack. In the overlay topology used by our FUSE implementation, these attacks can be mounted by malicious group members or delegates. The SV tree application handles the dropped notification attack above the FUSE layer by using redundant content-distribution trees. It handles the unnecessary notification attack by re-creating FUSE groups with different sets of members.

The first alternative topology we consider is per-group spanning trees *without* an overlay. By routing liveness checking traffic directly between members, this topology eliminates the threat of delegates launching attacks on FUSE. The scalability tradeoff for this additional security is that the overhead of liveness checking traffic may be additive in the number of FUSE groups – the opportunities to share liveness checking traffic will depend on the degree of overlap in FUSE group membership.

The second alternative topology we consider is per-group all-to-all pinging (again, without an overlay). This improves security even further; all-to-all pinging is robust to dropped notification attacks from members because no member relies on any other node to forward failure notifications. However, this topology requires n^2 messages for a group of size n – significantly more than the per-group spanning tree topology. An added benefit of the all-to-all topology is that worst-case failure notification latency is reduced to twice the pinging interval.

The final topology we consider is using a central server to ping all nodes. This may be an appropriate topology for using FUSE within a data center environment. From a security standpoint, this server represents a single point of trust, which may be easier to secure than a larger collection of machines. If the server is compromised, attacks can be launched against any FUSE group in the system. If not, the security guarantees are the same as in the all-to-all pinging topology. For settings than span multiple administrative domains, the use of a single trusted server may not be appropriate. The scalability of this topology is limited: all FUSE traffic passes through the server, which can be a bottleneck for a large number of FUSE participants. However, the load on each group member is minimal: each group member only pings the central server during each ping interval.

6 Implementation

The key architectural choice we faced in implementing FUSE was whether to route all FUSE messages using the overlay paths, or to route certain messages directly between the group members. In the topology we used, spanning trees along overlay routes, path failures involving delegates can be dealt with in one of two ways. One option is to signal a failure on all FUSE groups using that path. This has the advantage of implementation simplicity, but can be a significant source of false positives. Instead, we chose the second option: to attempt to repair the liveness monitoring topology for the group.

Repair will succeed if the members of the group can still communicate with each other directly, and therefore repair routes around all failures involving delegates. We chose to implement repair by routing repair messages directly from the root to the group members. The overriding factor for this choice was rapidity of failure detection: relying solely on overlay routes would require waiting for the overlay to attempt to repair itself before signaling a failure. Using direct root to member communication allows failures involving group members to be detected more rapidly. This direct communication also results in better latencies for group creation and application-signalled failure notifications. When overlay routing paths are working, we still get the scalability benefits of shared spanning trees using the overlay. In the remainder of this section, we first describe the functionality exposed by the SkipNet overlay, and we then discuss the details of implementing each FUSE operation.

6.1 Overlay Functionality

Our implementation of FUSE on top of the SkipNet overlay required two features that SkipNet provides to client applications: messages routed through the overlay result in a client upcall on every intermediate overlay hop, and the overlay routing table is visible to the client. This functionality is standard for many overlays [18].

Our FUSE implementation re-uses the overlay routing table maintenance traffic by piggybacking a SHA1 hash (20 bytes) on ping requests. This hash encodes all the FUSE groups that use this overlay link. FUSE could have sent its own messages across these same links, but the piggybacking approach amortizes the messaging costs. SkipNet pings cause a client upcall at their destination, so the destination FUSE layer can examine the piggybacked contents. Because all SkipNet links are monitored from both sides, we did not need to add additional pinging at the FUSE layer to ensure this. Implementing FUSE on an overlay that does not have these properties would require FUSE to perform additional pinging itself.

All FUSE and overlay messages in our system are delivered over TCP, and therefore inherit TCP's retry and congestion control behaviors. When a TCP connection breaks, or a liveness checking message fails to get through before the timeout, we interpret that to mean that the node at the other end is unavailable.

6.2 Group Creation

We implement group creation as follows: Group creation does not finish until every member node has a timer installed that will signal failure in the event of future communication failures. These timers are only reset by the receipt of liveness checking messages. Thus, any future communication failures will be converted into failure notifications.

To achieve low creation latencies, the creating node directly contacts every other member node in parallel,

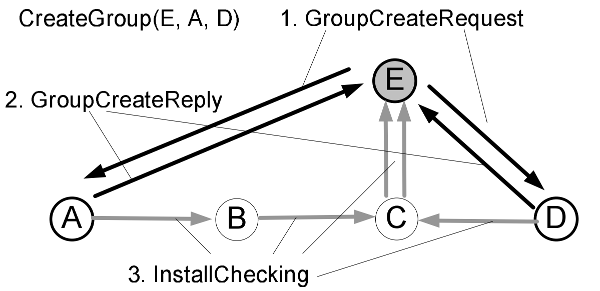

Figure 3. *Group Creation Example. Root node E sends GroupCreateRequest messages directly to group members A and D. Nodes A and D reply directly with GroupCreateReply messages, and also route InstallChecking messages though the overlay towards E.*

declaring creation to have succeeded when all of them have replied. When *CreateGroup* is called on a node, that node (referred to as the root) generates a unique ID for the group. The root also creates an entry in its list of groups being created, and associates a timeout with this group creation attempt. The entry contains the FUSE ID, the list of group members, and which members the root has received a reply from. The root then sends *GroupCreateRequest* messages to the other member nodes. An example message sequence that could result from group creation is shown in Figure 3.

On receiving a *GroupCreateRequest*, a member node installs FUSE member state for the group: the unique ID, a sequence number that is initially 0 (and which is incremented by group repair), and the identity of the root. Concurrently to sending the *GroupCreateReply* directly to the root, the member node routes an *InstallChecking* message towards the root using overlay routing. The *InstallChecking* message will set a timer on every node it encounters to ensure that liveness checks are heard.

If the root receives a *GroupCreateReply* from every member within the group creation attempt timeout, it installs the FUSE root state for the group: the unique ID, the sequence number, the identities of all the other group members, and a timer for checking that *InstallChecking* messages have arrived from every member. The root then removes this group from its list of groups being created and returns the unique ID to the FUSE client application.

If the *GroupCreateReply* is not received from every node within the group creation attempt timeout, the group creation fails and the root returns failure to the FUSE client application. The root also attempts to send a failure notification for this FUSE group to each group member. This notification is a *HardNotification* – we elaborate on the different types of notifications in Section 6.4. Finally, the root removes this group from its list of groups being created. This prevents *GroupCreateReply* messages received later from causing installation of state for the failed group creation.

When an *InstallChecking* message arrives at a delegate node, it installs the FUSE delegate state for the group: the FUSE ID, sequence number, and current time are associated with both the previous hop and the next hop of the *InstallChecking* message, and timers are associated with both hops as well. The node then forwards the message towards the root. If the timer for receiving all the *InstallChecking* messages fires on the root, the root attempts a repair.

6.3 Steady-State Operation

Whenever an overlay node initiates a ping to a routing table neighbor, it piggybacks a hash of the list of FUSE IDs that this node believes it is jointly monitoring with its neighbor. When the neighbor receives this message, if the hash matches, the neighbor resets the timers for all the (FUSE ID, neighbor) pairs represented by the hash. There can be more than one timer per FUSE ID because a node may have more than one neighbor in the liveness checking tree. If one of these timers ever fires, the node sends a *SoftNotification* message to every neighbor in the liveness checking tree for this FUSE group, and then it cleans up the FUSE delegate state for the group. Additionally, if the timer is firing on a member, a repair is initiated.

If a node receives a non-matching hash of FUSE IDs from a neighbor, both nodes attempt to reconcile the difference by exchanging their lists of live FUSE IDs. If they can communicate, they only remove the liveness checking trees on which they disagree, and the timers are reset on the others. If they cannot communicate, the relevant checking state is removed, and *SoftNotification* messages are sent.

During group creation, a race condition exists that can cause hash mismatches: a node that has just received an *InstallChecking* message may receive a ping from the next hop of the *InstallChecking* message. We resolve this race condition using a brief grace period. A node only removes a liveness checking tree that its neighbor does not believe exists if that tree has existed for longer than the grace period; in our implementation, this period is 5 seconds.

6.4 Notifications

To achieve the simultaneous goals of low notification latency and resilience to delegate failures, our FUSE implementation distinguishes between different classes of failures. Failures of the steady-state liveness checking trigger a *SoftNotification*. This message is distributed throughout the liveness checking tree, which alerts the root that a repair is needed and prevents a storm of *SoftNotifications* from being sent to the root by the rest of the tree. Members receiving a *SoftNotification* also initiate repair directly with the root as described in Section 6.5.

Failures of group creation or group repair trigger a *HardNotification*. Because both create and repair use direct root-to-member communication, delegate failures do not incur false positives. Note that *SoftNotifications* do not cause failure notifications at the application layer. In-

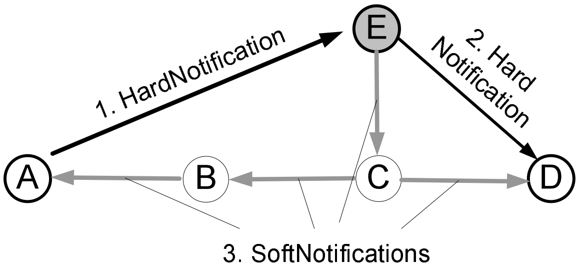

Figure 4. *Explicitly Signalled Notification Example. SignalFailure() is called on node A. Node A sends a HardNotification to the root, E. E forwards the HardNotifcation to the remaining group member, D. E also generates a SoftNotification to clean up the liveness checking tree.*

stead, they trigger repair actions. The failure of these repair actions will lead to a *HardNotification*, which is reflected at the application layer. To achieve low latency for explicitly signalled notifications, *HardNotifications* are also used to convey them.

A member generating a *HardNotification* sends it to the root, which in turn forwards it to all other group members. A node receiving a *HardNotification* immediately invokes the application-installed failure handler. The root node additionally sends *SoftNotifications* to proactively clean up the liveness checking tree. An example of such a message sequence is shown in Figure 4.

A node receiving a *SoftNotification* message first checks to see that the sequence number is greater than or equal to its recorded sequence number for the specified group; recall that the sequence number is incremented during the repair process. If not, the message is discarded. If the sequence number is current, the node forwards the message on to all neighbors in the liveness checking tree other than the message originator, and removes its delegate state for the group. If the node is a member or the root, it also initiates repair.

6.5 Group Repair

When a member initiates a repair, it sends the root a *NeedRepair* message, and installs a timer for hearing back from the root. If the timer fires, it signals a failure notification to the FUSE client application, sends a *HardNotification* message to the root, and cleans up the state associated with this group.

The root can be signalled that a repair is needed through either of two paths: a *NeedRepair* directly from a member or a *SoftNotification* spreading through the liveness checking tree. The *NeedRepair* message is needed to remove a potential source of variability in the latency of repair. When a member receives a *SoftNotification*, it does not have a good estimate for how soon the root will similarly receive a *SoftNotification*; these notifications are routed through the overlay, and breaks in overlay routing require a timeout on the other side to continue the progress of the *SoftNotification*. An example of a sequence of messages leading to a *NeedRepair* is shown in Figure 5.

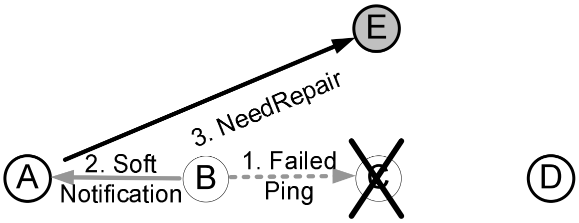

Figure 5. *Messages Triggering Group Repair Example. Delegate B sends a ping to delegate C, and the ping is not acknowledged. B then sends a SoftNotification to member A. A then sends a NeedRepair to root E. Once E has been alerted, E coordinates repair just as it coordinated creation.*

When the root attempts a repair, it sends out *GroupRepairRequest* messages to every member. Members answer these with *GroupRepairReply* messages, and they send *InstallChecking* messages just as in group creation. State management at the root during repair is similar to creation, involving a repair attempt table where open repairs are recorded. State management at member nodes is different: if a repair message ever encounters a member that no longer has knowledge of the group, it fails and signals a *HardNotification*. This guarantees that repairs will not suppress any *HardNotification* that has already reached some members. Such notifications garbage collect all group state at the node.

Nodes receiving a *GroupRepairRequest* increment the group sequence number so that late-arriving *SoftNotification* messages will not trigger a redundant repair. If the root decides that repair has failed (using the same criterion as for create failing), the root sends *HardNotifications* to all members, and signals the application.

As we discussed in Section 5, using overlay paths allows us to achieve low steady-state message overheads. We believe a repair scheme that better localizes repair traffic is possible, but we did not consider this a case worth optimizing. During transient overlay routing failures, repairs may be quite frequent; a node consulting its routing table may learn that there is no next hop for an *InstallChecking* message. To reduce the message volume under such circumstances, we implement per-group exponential backoffs (capped at 40 seconds) for the frequency of repairs. Although repair generates additional network traffic shortly after a failure has been detected, our use of TCP serves to regulate this additional network load.

7 Experimental Evaluation

We evaluate FUSE running on top of the SkipNet [25] overlay network using two main techniques: a scalable discrete event simulator and a live implementation with up to 400 virtual nodes running on a cluster of 40 workstations. Our SkipNet and FUSE implementations running on the live system and in the simulator use an identical code base, except for the base messaging layer.

7.1 Methodology

We configured the SkipNet overlay to employ a 60 second ping period, a base of size 8, and a leaf set of size 16. For a 400 node overlay, this yielded an average of 32.3 distinct neighbors per node.

For the cluster evaluation our router uses ModelNet [41] to emulate wide-area Internet-like network characteristics. We ran 10 processes on each of the 40 physical nodes, for a total of 400 virtual nodes. In order to emulate nodes running on physically separate machines, there is no explicit state sharing between these processes, and all communication between processes is forced to pass through ModelNet.

The motivating scenario for our choice of router topology and our assignment of link latencies and bandwidths was small and large corporations on multiple continents with direct Internet connections. Both our live and simulator experiments were run on a Mercator topology [40] with 102,639 nodes and 142,303 links. Each node is assigned to one of 2,662 Autonomous Systems (ASs). There are 4,851 links between ASs in the topology. We assigned 97% of links to be OC3 and 3% to be T3. For each OC3 link, we assigned the link latency uniformly between 10 and 40 milliseconds, and we assigned a bandwidth of 155 Mbps. For each T3 link, we assigned the link latency uniformly between 300 and 500 milliseconds, and we assigned a bandwidth of 45 Mbps. This led to round-trip latencies with a median value of 130 milliseconds, and a significant heavy-tail. In Figure 6, the curve labeled Simulator shows a CDF of end-to-end latencies; paths crossing one or more T3 links are in the heavy-tail.

We also used this topology in our simulator, where we ran experiments with both 400 and 16,000 nodes, to model how our system would scale to a much larger deployment. The simulator used the same latency values, but did not model bandwidth constraints.

Our event notification service workload (Section 4) creates a large number of groups with an average group size of less than 3. Even scaling up to a 16,000 node overlay in our simulator, the maximum group size we observed was 13. Just as these results informed our FUSE design, they also determined our choice of evaluation parameters: we evaluated FUSE on a workload of groups ranging from 2 to 32 members.

7.2 Calibration of Simulator and ModelNet

We used an experiment that performed RPC message exchanges between randomly chosen nodes on a 400-node overlay network to calibrate the wide-area network topology model used in our experiments and to make sure that results obtained through simulation were comparable to those obtained through running on the live cluster with ModelNet.

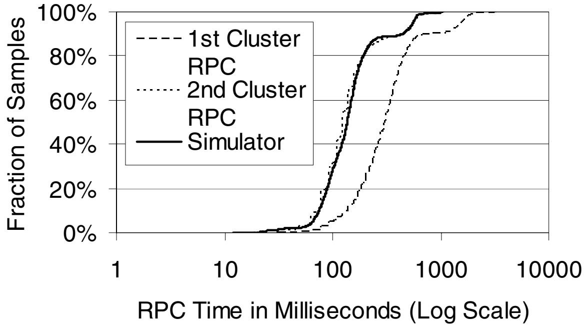

Figure 6. *RPC Latencies*

Towards that end, we measured 2400 RPCs on both our cluster and our simulator. Figure 6 shows a Cumulative Distribution Function (CDF) of the RPC times measured for three sets of RPCs: those obtained in the simulator, and two kinds of RPC times obtained on the cluster. Because the cluster code caches TCP connections between pairs of nodes, the first communication between a pair of nodes takes longer than subsequent communications, due to the additional time required for connection establishment. Our experiment performs two back-to-back RPCs between pairs of nodes on the cluster and reports the durations of both the first RPC, which is likely to incur connection setup overhead, and the second one, which will not.

As can be seen in Figure 6, the values for the second RPC on the cluster closely track those for the simulator. This gives us confidence that both the simulator and ModelNet are faithfully modeling the chosen Mercator topology.

7.3 FUSE Group Creation Latencies

We measured the time on the cluster required to create a FUSE group as a function of group size when group members are uniformly distributed throughout the system. We used group sizes 2, 4, 8, 16, and 32 and created 20 groups of each size. Figure 7 shows the results. While group members were contacted in parallel, the more members there are, the greater the chance of including nodes at a significant network distance from the root node of the group. Note, for instance, that for groups of size 8, the 75th percentile time was significantly larger than the median, whereas for groups of size 32, the 25th percentile, median, and 75th percentile are all relatively close, as with this many members the chance of encountering one or more slow communication paths is quite high.

In the simulator, we evaluated both a 400 node and a 16,000 node system. The simulated creation times followed the same pattern as those on the cluster, except that they tended to be about half as long, for the same reasons that the simulated and actual RPC times in Figure 6 for new connections also differed by about a factor of two. The group creation times for the simulated 16,000 node system were essentially identical to those for the 400 node

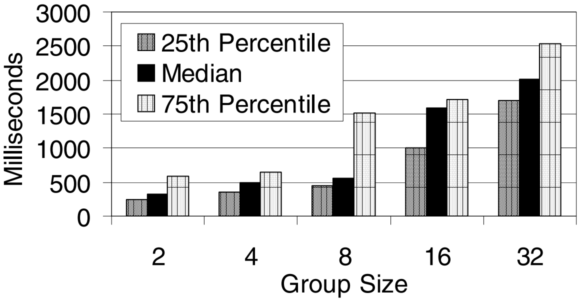

Figure 7. *Latency of Group Creation*

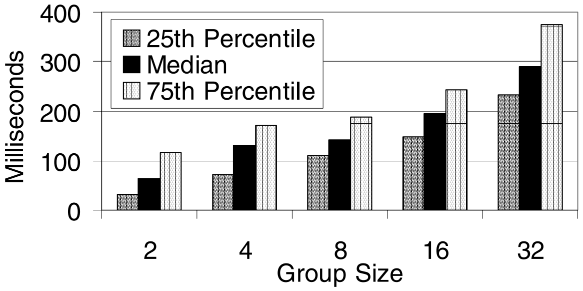

Figure 8. *Latency of Signaled Notification*

simulated system. This is to be expected, since creation messages are routed directly between the root and members, and therefore are not affected by the length of overlay routes.

7.4 Failure Notification Latencies

The latency of failure notification for an actual failure is comprised of two parts: the time for a node in the system to decide that a failure has occurred, and the time for FUSE to propagate that information to all group members. The time to detect a failure depends on the type of failure, and on how node and link failures are monitored. We perform two experiments to characterize these costs: our first experiment investigates the latency of explicitly signaled failures, and our second investigates the latency of failure notification when nodes crash.

To measure the notification latency of explicitly signaled failures, we chose a group member at random from the same set of groups used for the group creation experiments, and had it explicitly signal failure. Figure 8 shows the notification times over 20 such create/notify cycles. As expected, the notification latencies are significantly lower than the group creation latencies. This improvement is a result of three factors. First, our messaging layer maintains a cache of recently used TCP connections rather than opening a new TCP connection each time a message is sent. During this experiment, the failure notifications travel over cached TCP connections because these connections were recently used to perform group creation. Second, failure notifications only require a one-way message, not a round-trip. Third, creation blocks at the root until all members have been contacted, and thus a single node in the group that is far away in the network will delay the entire create operation. In contrast, notification takes effect at each member as soon as the notification has arrived. The maximum notification time observed for any FUSE group was 1165 ms. Our simulation results at 400 nodes matched the results obtained on the cluster. We also investigated scaling behavior in the simulator, and we found the expected result that notification times did not increase in a 16,000 node overlay.

Even though failure notifications take effect at each member upon arrival, in Figure 8 we see that the median notification latency shows a dependence on the group size. The rise in the curve at group sizes 2, 4, and 8 is due to the extra forwarding hop needed when notifications are generated by a non-root node. At size 2, these notifications just travel from the member to the root, whereas at sizes 4 and 8 notifications travel from the member to the root and then to all other members. The additional increase in notification latency for groups of size 16 and 32 reflects the latency added by our messaging layer at the root. Our implementation uses an XML-based messaging system with high message serialization overhead: reducing this overhead would be straightforward but this was not the focus of our work. We ran micro-benchmarks and determined that running 10 virtual nodes on each physical machine adds approximately 1.1 ms of overhead per message, and the base overhead of a message send including XML serialization is 2.8 ms.

To measure the latency of failure notification when nodes crash, we performed the following experiment: we created 400 FUSE groups of size 5 and then disconnected the network on one of the 40 physical machines, disconnecting 10 of the 400 virtual nodes. Of the 400 FUSE groups, 42 contained one or more disconnected virtual nodes as members. All remaining members of these groups received failure notifications – a total of 163 notifications.

Figure 9 shows the distribution of these notification times. These times have several components: the time until a ping of the failed node is attempted, the timeout for this ping, the time for a member to learn of the failed ping, the time for the subsequent repair attempt to fail, and the actual notification time after repair has failed. We used a ping interval of one minute, and a ping timeout of 20 seconds, so the total ping timeout latency should be uniformly distributed between 20 and 80 seconds. If a member has failed, the root times out after 2 minutes with no repair response. If a root has failed, the members time out after 1 minute with no repair response, leading to shorter overall failure notification times. Figure 8 shows that the notification times are less than a second. We deduce from this that the ping and repair timeouts dominate the failure notification times in the event of a node crash.

7.5 Steady State Load and Churn

One set of experiments we performed measured the amount of background network traffic present due to the overlay network and due to FUSE groups that are using

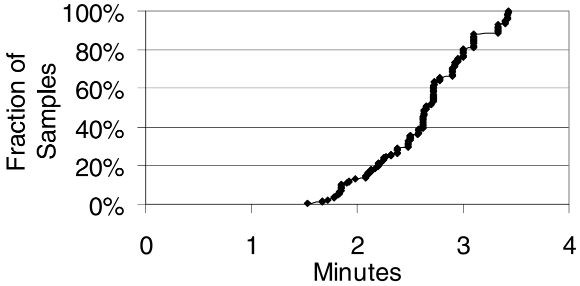

Figure 9. *Combined Latency of Ping Timeout, Repair Timeout, and Failure Notification. Ping and Repair Timeouts dominate other factors.*

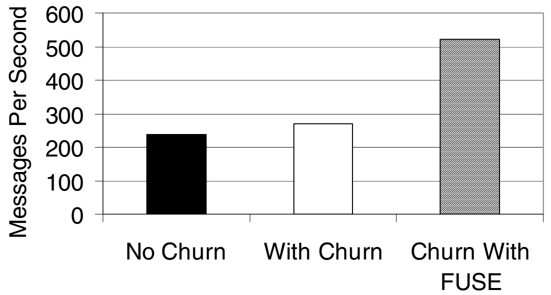

Figure 10. *Costs of Overlay Churn*

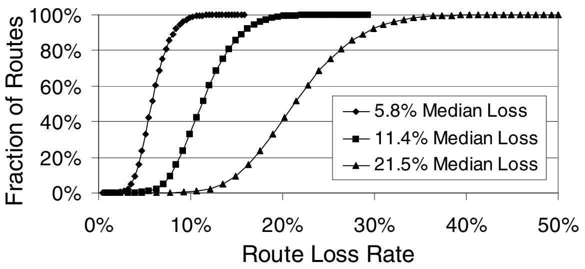

Figure 11. *CDFs of Per-Route Loss Rates for Three Different Per-Link Loss Rates*

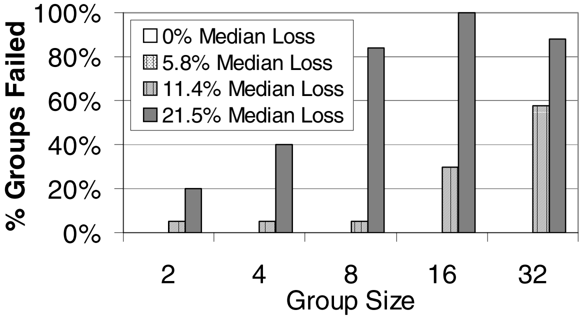

Figure 12. *Group Failures Due to Packet Loss. No failures occurred for 0% and 5.8% loss rates.*

the overlay network for liveness checking. On the cluster, we observed background network traffic loads of 337 messages per second over a 10 minute interval when no FUSE groups were present and 338 messages per second over a subsequent 10 minute interval when 400 FUSE groups of 10 members each were present. These experiments verified that, in the absence of node failures, FUSE groups imposed no additional messages beyond that already imposed by the overlay itself; the only additional cost was a 20 byte hash piggybacked on each ping.

When nodes are entering and leaving the overlay network (often referred to as "churn"), the overlay paths used for liveness checking between nodes in a FUSE group may change, causing the liveness checking state to have to be reconstructed. Overlay churn does not cause false positives in FUSE, but it does cause FUSE to generate higher network load. We experimentally quantified the network load imposed by a high churn rate with FUSE groups.

We designed our churn experiment to use a very aggressive rate of arrivals and departures. We used 200 stable nodes that remained alive for the duration of the experiment and 200 nodes that were killed and restarted such that an average of 100 of the churning nodes were alive at any given time. The rate of churn resulted in a system-wide average half-life of 30 minutes. This is more than a factor of 7 higher rate of churn than was observed in a 2003 study of the OverNet peer-to-peer system [3]. We created a total of 100 FUSE groups of size 10 on the 200 stable nodes, so that on average each stable node was a member of five FUSE groups.

We measured both CPU loads and network message traffic. CPU loads did not show noticeable increases during overlay churn. As a basis of comparison, a stable 300-node overlay with no FUSE groups generates a load of 238 messages per second. A 400-node overlay network with churn as described above (which results in an average of 300 live nodes at any given time) generates a load of 270 messages per second – a 13% increase due to the costs of repairing the overlay. Adding the 100 10-member FUSE groups to the churning overlay results in a total of 523 messages per second – a 94% increase over the cost of the churning overlay without FUSE groups. These results are displayed in Figure 10. This additional load is caused by the group repair traffic described in Section 6, and is proportional to the number of groups times the average group size. While the overlay routes are in flux, new liveness checking paths cannot be installed and thus the repair mechanism will be triggered repeatedly. One could reduce the FUSE overhead during churn by employing a more proactive repair strategy at the overlay level.

A high enough rate of churn may cause overlay routing to fail entirely. In this case, the FUSE liveness checking traffic will still be proportional to the number of groups times the average group size. After reaching steady state, each root node will periodically ping each group member directly with a *GroupRepairRequest* message.

7.6 False Positives

We studied the robustness of our implementation to false positives stemming from two different sources, delegate failures and unreliable communication links. In the previously described churn and node crash experiments, many groups experienced delegate failures and had to perform repairs. In both these experiments, notifications were only delivered for groups where a member crashed: delegate failures never led to a false positive.

To understand the impact of potentially unreliable links on FUSE, we ran a set of experiments with ModelNet configured to probabilistically drop packets on a per-link basis. Routes in our topology ranged from 2 to 43 hops with a median of 15. Figure 11 shows the CDFs of per-route loss rates for three different experiments where we varied the per-link loss rates over 0.4%, 0.8%, and 1.6%. The CDFs are labeled by their median end-to-end loss rates. We created 20 FUSE groups each of sizes 2, 4, 8, 16, and 32. We then enabled losses, and allowed the system to run for an additional 30 minutes.

Figure 12 shows the number of FUSE group failures we observed at each loss rate. No false positives occur at the 0% or 5.8% per-route median loss rates because TCP masks drops at the lower loss rates through retransmissions. At higher loss rates some groups did fail; TCP sockets will break under such adverse network conditions. If one desired a FUSE implementation that continued to monitor links under these conditions, an alternative messaging layer should be employed.

8 Conclusions

This paper has presented FUSE – a lightweight distributed failure notification facility. FUSE provides a novel abstraction, the FUSE group, that is targeted at wide-area Internet applications. The FUSE group abstraction provides distributed application developers with a simple programming paradigm for handling failure: failure notifications never fail, and failures are with respect to groups, not individual members. One significant advantage of the FUSE abstraction is that detecting failures is a shared responsibility between FUSE and the application. This allows applications to implement their own definitions of failure, extending the applicability of failure management services.

We implemented FUSE using a peer-to-peer overlay network and evaluated its behavior on a cluster of workstations under a variety of conditions, including node failures, packet loss, and overlay churn. Our evaluation showed that our FUSE implementation is lightweight and can scale to large numbers of moderate size FUSE groups. Our implementation scales by reusing overlay maintenance traffic to also perform liveness checking of FUSE groups, thereby imposing no additional traffic in the absence of node failures. Because the FUSE abstraction can be implemented efficiently, it may find application in domains where consensus-style protocols have proven to be too heavyweight.

FUSE simplifies the complex task of handling failures in distributed applications. We described our experiences building scalable, reliable application-level multicast groups using FUSE, and how FUSE made this task easier. We believe that the use of FUSE will likewise simplify the construction of other distributed systems.

Acknowledgements

We thank Ken Birman, Jon Howell, Patricia Jones, Keith Marzullo, Stefan Saroiu, Amin Vahdat and Geoff Voelker for their comments on earlier drafts of this paper. We thank Ashwin Bharambe for his insight on integrating ModelNet. We also thank our shepherd, Greg Ganger, and the anonymous reviewers for their insightful feedback.

References

[1] M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *Workshop on Distributed Algorithms*, pages 126–140, 1997.

[2] T. Anker, D. Breitgand, D. Dolev, and Z. Levy. Congress: CONnection-oriented Group-address RESolution Service. Technical Report TR 96-23, Hebrew University of Jerusalem, 1996.

[3] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *2nd Intl. Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[4] R. Bhagwan, K. Tati, Y.-C. Cheng, S. Savage, and G. M. Voelker. TotalRecall: System Support for Automated Availability Management. In *1st NSDI*, 2004.

[5] K. Birman, R. van Renesse, and W. Vogels. Adding high availability and autonomic behavior to web services. In *Intl. Conference on Software Engineering (ICSE)*, 2004.

[6] K. P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer-Verlag, Scheduled for 2004.

[7] L. F. Cabrera, M. B. Jones, and M. Theimer. Herald: Achieving a Global Event Notification Service. In *HotOS*, 2001.

[8] R. Callon. RFC 1195. Use of OSI IS-IS for Routing in TCP/IP and Dual Environments. Dec. 1990.

[9] G. Candea, M. Delgado, M. Chen, F. Sun, and A. Fox. Automatic Failure-Path Inference: A Generic Introspection Technique for Software Systems. In *IEEE Workshop on Internet Applications (WIAPP)*, 2003.

[10] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *3rd OSDI*, 1999.

[11] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. In *PODC*, 1992.

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

[13] M. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based Failure and Evolution Management. In *1st NSDI*, 2004.

[14] M. Chen, E. Kiciman, A. Accardi, A. Fox, and E. Brewer. Using Runtime Paths for Macroanalysis. In *HotOS*, 2003.

[15] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Systems. In *DSN*, 2002.

[16] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer. A Statistical Learning Approach to Failure Diagnosis. In *Intl. Conference on Autonomic Computing (ICAC)*, 2004.

[17] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. In *DSN*, 2000.

[18] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. Towards a common API for structured peer-to-peer overlays. In *2nd Intl. Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[19] A. Das, I. Gupta, and A. Motivala. SWIM: Scalable Weakly consistent Infection-style process group Membership protocol. In *DSN*, 2002.

[20] J. Dunagan, N. J. A. Harvey, M. B. Jones, M. Theimer, and A. Wolman. Subscriber/Volunteer Trees: Polite, Efficient Overlay Multicast Trees. http://research.microsoft.com/sn/Herald/papers/SVTree.pdf, *Submitted for publication*, 2004.

[21] P. Felber, X. Defago, R. Guerraoui, and P. Oser. Failure Detectors as First Class Objects. In *Intl. Symposium on Distributed Objects and Applications*, 1999.

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

[23] I. Gupta, T. Chandra, and G. Goldszmidt. On Scalable and Efficient Distributed Failure Detectors. In *PODC*, 2001.

[24] J. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37:549–587, 1990.

[25] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Fourth USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[26] P. Ji, Z. Ge, J. Kurose, and D. Towsley. A Comparison of Hard-state and Soft-state Signaling Protocols. In *SIGCOMM*, 2003.

[27] C. Labovitz and A. Ahuja. Experimental Study of Internet Stability and Wide-Area Backbone Failures. In *Fault-Tolerant Computing Symposium (FTCS)*, 1999.

[28] L. Lamport. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM TOPLAS*, 6:254–280, 1984.

[29] L. Lamport. The Part-Time Parliament. *ACM TOCS*, 16:133–169, 1998.

[30] B. Liskov. Distributed Programming with Argus. *CACM*, 31:300–312, 1988.

[31] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *SIGCOMM*, 2002.

[32] J. Mogul, L. Brakmo, D. E. Lowell, D. Subhraveti, and J. Moore. Unveiling the Transport. In *HotNets*, 2003.

[33] J. T. Moy. *OSPF: Anatomy of An Internet Routing Protocol*. Addison-Wesley, 1998.

[34] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *SIGCOMM*, 2001.

[35] T. L. Rodeheffer and M. D. Schroeder. Automatic Reconfiguration in Autonet. In *SOSP*, 1991.

[36] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[37] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Third Intl. Workshop on Networked Group Communications*, 2001.

[38] P. Stelling, C. Lee, I. Foster, G. von Laszewski, and C. Kesselman. A Fault Detection Service for Wide Area Distributed Computations. In *High Performance Distributed Computing*, 1998.

[39] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *SIGCOMM*, 2001.

[40] H. Tangmunarunkit, R. Govindan, S. Shenker, and D. Estrin. The Impact of Routing Policy on Internet Paths. In *Infocom*, 2001.

[41] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *5th OSDI*, 2002.

[42] R. van Renesse, Y. Minsky, and M. Hayden. A Gossip-Based Failure Detection Service. In *Middleware*, 1998.

[43] W. Vogels. World wide failures. In *ACM SIGOPS European Workshop*, 1996.

[44] W. Vogels and C. Re. Ws-membership - failure management in a web-services world. In *Intl. World Wide Web Conference (WWW)*, 2003.

[45] H. Yu and A. Vahdat. Consistent and Automatic Replica Regeneration. In *1st NSDI*, 2004.

[46] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-Resilient Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, UC Berkeley, 2001.