An Experimental Evaluation of Correlated Network Partitions in the Coda Distributed File System

Ryan M. Lefever[*], Michel Cukier[**], and William H. Sanders[*]

[*]Coordinated Science Laboratory and Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{lefever,whs}@crhc.uiuc.edu
[**]Department of Mechanical Engineering
University of Maryland, College Park, MD 20742, USA
mcukier@eng.umd.edu

Abstract

Experimental evaluation is an important way to assess distributed systems, and fault injection is the dominant technique in this area for the evaluation of a system's dependability. For distributed systems, network failure is an important fault model. Physical network failures often have far-reaching effects, giving rise to multiple correlated failures as seen by higher-level protocols. This paper presents an experimental evaluation, using the Loki fault injector, that provides insight into the impact that correlated network partitions have on the Coda distributed file system. In this evaluation, Loki created a network partition between two Coda file servers, during which updates were made at each server to the same replicated data volume. Upon repair of the partition, a client requested directory resolution to converge the diverging replicas. At various stages of the resolution, Loki invoked a second correlated network partition, thus allowing us to evaluate its impact on the system's correctness, performance, and availability.

Keywords: experimental evaluation, correlated network faults, fault injection, Loki fault injector, state-driven fault injection, Coda, distributed file system

1 Introduction

Distributed systems are becoming increasingly common, and are being used in many diverse scenarios, such as web servers, distributed multimedia, distributed operating systems, and mobile and ubiquitous computing environments. We can reason about some of the requirements of these systems using formal methods. However, it is preferable to validate some of their other requirements using experimental techniques. The motivation for using experimental validation could be due to the complexity of the system being validated, or to the nature of the requirements considered. Furthermore, experimental techniques can be used to validate the implementation of a system.

Fault injection is one technique that shows significant promise for distributed system validation. Loki [5, 4, 3] is a fault injector developed at the University of Illinois that has been designed specifically for distributed systems. Its most distinctive feature, relative to other fault injectors, is its ability to trigger faults based on the global state of the system under study (SUS). That allows one to examine experimental scenarios that might not otherwise be possible to examine.

This paper reports on our efforts to use Loki to evaluate Coda [17], a distributed file system developed at Carnegie Mellon University. Coda is interesting because of its elaborate design for maintaining high availability, supporting mobile computing, and managing shared resources. There are two fault models that Coda is designed to address: network partitions and crash failures. Coda claims to achieve high availability by employing server replication and supporting disconnected operation under an optimistic consistency strategy. Due to the difficulty of experimentally evaluating a system as large and elaborate as Coda, our work is focused on the impact of correlated network partitions on the correctness of Coda's data resolution techniques, as well as their impact on Coda's performance and availability. A large body of work exists about Coda, including work on its resolution techniques [11, 12, 13] and an empirical study of its performance over a six-month period [16].

We chose to conduct our study using Loki because 1) its global-state-triggered fault injection mechanism enables us to perform the correlated network partitions and guarantees their correctness, and 2) its global-state-tracking ability and measures support allow us to obtain a wide variety of measures. Several other fault injectors exist for the study of networked systems (e.g., [8, 9, 7, 18]). They are use-

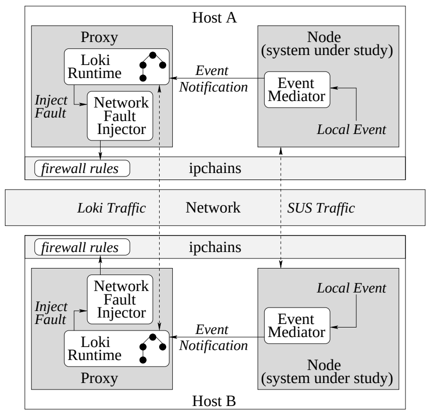

Figure 1. Loki and its fault injection proxy

ful for many studies, but typically do not have the ability to trigger faults based on the global state of the SUS or to compute many fine-grained dependability and performance metrics, such as the ones we wish to compute for Coda. In particular, most fault injection case studies of network systems have been restricted to examination of coverage measures [1, 19, 6]. Other tools exist for the measurement of metrics in distributed systems [14, 2], but they do not have the ability to inject faults. [10] is a good illustration of Loki's ability to inject correlated faults based on state.

The remainder of this paper includes an overview of Loki (Section 2), an overview of the Coda case study (Section 3), a presentation of measures and experimental results (Section 4), and a conclusion (Section 5).

2 Overview of Loki

This section reviews basic Loki concepts. Space does not permit a more comprehensive review. See [5, 4, 3] for more Loki details and [15] for more proxy details.

2.1 Steps to conduct a fault injection study

There are three main steps involved in defining a Loki fault injection study. The first step is to define Loki *state machines* to track the local state of basic components, referred to as *nodes*, in the SUS. A vector of each node's local state defines the *global state* of the system. The second step is to use Loki primitives to identify local events at each node that trigger transitions in the local state machine. The third step is to define faults to inject during the study. The experimenter defines the actions to be taken when the fault is injected as well as a *fault trigger* based on the global state of the SUS; the fault trigger determines when the fault should be injected.

We used Loki with its *lightweight fault injection proxy* to perform experiments. A proxy is set up for each node in the SUS to provide the node with Loki runtime support. Also, an *event mediator* that supports the Loki primitives for signaling local events is attached to each node. Figure 1 illustrates the interaction between Loki and the SUS. Each proxy maintains Loki runtime functionality, including state tracking and fault triggering, for its associated node. When a fault is to be injected, the Loki runtime can invoke a local fault injector like the proxy's network fault injector, or invoke an external command.

After experiments have completed, experiment data is processed. The first step is to run Loki's analysis program to create global timelines and determine the correctness of fault injections from the experiment data. Following the analysis, the experimenter defines measures using Loki's measures language. Finally, measures are computed.

2.2 Loki proxy support

State machine support The Loki proxy provides two features that assist an experimenter in designing state machines. The first feature is an *auxiliary event*, which is generated upon completion of a fault injection. There are three types of auxiliary events. The first type notifies the state machine that the associated fault injection has completed. It is called a *complete event* and is named <fault name>_complete. Closely related to complete events are *returnval events*. They embed a numeric return value from a fault injection into the event's name, using the format <fault name>_ret_<return value>. Complete and returnval events are important because they provide the means for an experimenter to build state machines based on knowledge about faults. The third type of auxiliary event is a *globally conditioned event*. Events of that type are named <fault name>_globalcond, and signal the occurrence of global conditions; thus, the associated faults do nothing.

The second feature is an *auxiliary state*. Auxiliary states represent knowledge that is beyond the scope of the part of the SUS in which the experimenter is interested, such as experiment state. Auxiliary states can be used for such purposes as synchronization within or between state machines, and execution of sanity checks.

Fault injection support The Loki proxy supports two fault injection features. The first feature is a *control modifier*. Control modifiers are stimuli that allow an experimenter to guide and drive the state of the SUS. That is useful because most distributed systems change their behavior based on responses to internal and external input such as client requests, messages, and system events. The proxy takes advantage of Loki's fault injection mechanism as a means to create stimuli, thus giving the proxy temporal control over their creation, based on global state triggering. Since control modifiers use the same injection mechanism as faults, general notions applying to faults also apply to control modifiers.

The second feature is the proxy's *network fault injector*. The network fault injector was built using system calls to IP Chains, a rule-based interface to IP firewall administration in the Linux kernel. Figure 1 illustrates the interaction between the network fault injector and IP Chains. The network fault injector supports the ability to deny network traffic into and out of the local host. The denial of service can be restricted based on hosts, ports, and/or network protocols. It is important that we be able to restrict the denial of network traffic to particular ports and network protocols, since we would otherwise be unable to avoid denying Loki runtime traffic, thus rendering this approach unusable. We can create a network partition using the network fault injector by denying service between two groups of hosts on the ports and network protocols that the SUS uses. The network fault injector also supports the ability to repair the network.

3 Coda overview, state machines, and faults

3.1 Coda overview

Review of concepts Coda [17] is a distributed file system designed to be highly available. It tolerates network partitions and server crashes through server replication and disconnected operation. *Server replication* allows data to be replicated across servers and makes the data accessible to a client even if not all of the servers are reachable. In Coda, data is replicated at the granularity of a *volume*. The data to be stored in the file system is divided into volumes, such that a subdirectory of the file system and the subdirectory's contents form a volume. The servers that host a replica of a particular volume form a *volume storage group* (VSG), and the subset of this group that is reachable by a client is the client's *available volume storage group* (AVSG). *Disconnected operation* allows a Coda client to cache data locally, to be used if the client's AVSG is empty. Replication of data is made transparent whenever possible.

In conjunction with those mechanisms, Coda uses an *optimistic data consistency* scheme that allows partitioned reads and writes to occur, by assuming that the same data object will not be accessed in separate partitions. Although the optimistic scheme improves the availability of data, Coda must support a mechanism to handle diverging replicas in the event that the optimistic assumption does not hold. That mechanism is called *resolution*, which is defined in [11, p. 13] as "the process of converging diverging replicas to the same value." The burden of detecting replica divergence is left to *Venus*, Coda's client cache management process. When a client accesses a replica and Venus detects a divergence, Venus requests that its AVSG resolve the divergence.

Coda breaks down resolution in two ways: automatic versus manual resolution, and directory versus file resolution. *Automatic resolution* is resolution that Coda can perform by itself without ambiguity; divergences that cannot be automatically resolved require a human to perform *manual resolution*. A distinction is also made between *directory resolution* and *file resolution*, because directories have well-defined semantics that allow many divergences to be automatically resolved, while files do not. In the rest of this paper, we will use the terms *directory resolution* and *file resolution* to refer to the resolution of a directory object and file object, respectively, and *resolution* or *volume resolution* for the process of resolving an entire volume, which is really just a series of directory and file resolutions.

Description of the directory resolution protocol This subsection presents a brief description of Coda's directory resolution protocol. A more detailed description of the protocol can be found in [11]. We consider the protocol for a volume that consists of exactly one directory, since that is what we used in our experiments. The protocol is started when a Venus process sends a server a ViceResolve request, making the server the *coordinator* of the directory resolution. The remaining servers in the client's AVSG become *subordinates* in the protocol. Together the coordinator and the subordinates form a *directory resolution group* (DRG). The directory resolution protocol consists of several phases that are performed one after another as requested by the coordinator. Subordinates are unaware that the protocol is taking place. Rather, they fulfill requests as made by the coordinator. During Phase 1 (LockVol) of directory resolution, the coordinator asks the DRG to lock the volume containing the directory to be resolved. It also compares the status of the directory at each server to determine if resolution is needed. In Phase 2 (FetchLogs), the coordinator collects from all subordinates the logs that it needs in order to resolve the directory, and merges the logs into a single buffer. Then, in Phase 3 (ParseLogs), the coordinator distributes the combined log buffer to the subordinates, which, along with the coordinator, perform compensating operations to bring their copies of the directory up to date. The subordinates return any inconsistencies that arise to the coordinator. If inconsistencies are reported, Phase 3.5 (HandleInc) is used by the coordinator to distribute the list of inconsistencies to subordinates to verify that the inconsistencies exist. If no inconsistencies exist, then in Phase 4 (InstallVV) the coordinator ships a new storeid for the directory to each subordinate, and the DRG commits the updated directory. If inconsistencies do exist, Phase 4 is skipped and Phase 5 (MarkInc) is used by the coordinator to order the DRG to mark the volume as inconsistent. Finally, Phase U (UnlockVol) is used by the coordinator to instruct the DRG to unlock the volume.

3.2 Loki state machines

Having reviewed Coda, our next step is to define Loki state machines for the Coda file servers in our experiments. Although no formal methods exist to guide state machine construction, state machines are given as input to Loki to facilitate fault triggering and measure computation and must be designed as such. We begin by describing our experiment scenario.

Experiment scenario In this work, we are interested in studying the effects of correlated network partitions that occur during a phase of Coda's directory resolution protocol. To study them, we designed an experiment with a Coda server and client on each of two hosts. Each server was limited to one lightweight process (LWP) to serialize its work[1]. After the servers and clients were initialized, a volume was replicated across the two servers and initialized with data. The volume, called the *target volume*, consisted of a single directory containing several files. After that setup, a network partition was injected. During the partition, each client performed a workload of reads and writes to its local replica such that the resolution of the partitioned updates could be performed automatically. After the workloads were completed, the partition was repaired. One of the clients then requested that the target volume be resolved. While the servers were performing resolution, a second correlated network partition was injected during some phase of the target volume's directory resolution, causing the resolution process to abort. After it aborted, the network partition was once again repaired; resolution was again requested, but this time was allowed to finish. We have performed correlated injections into Phases 3 and 4 of directory resolution, which correspond to fault injection *studies* named *P3* and *P4*. In addition, we have run a baseline case called *study B*, without the second correlated network partition.

Coda resolution state machine The Loki state machine in Figure 2 represents the state of a Coda server during volume resolution[2]. In the state machine, vertices represent states and arcs represent event-triggered transitions. If { and } appear in the name of an arc, it signifies that more than one event can cause the transition, e.g., `ev_{1,2}` represents both `ev_1` and `ev_2`. Also, some arcs are labeled with a list of events that can trigger the associated transition. The following describes the most common paths through the state machine; however, other paths are possible.

Each time a client accesses an object composed of diverging replicas, Venus sends a ViceResolve request to a server in its AVSG. In a new resolution process, that server becomes a coordinator, and the remaining servers in the AVSG become subordinates. A `viceresolve_received` event is generated by the coordinator when it receives the request, transitioning the coordinator to the `ResCoord` state, in which it resides between resolution actions. From here, the coordinator coordinates either directory or file resolution, depending on the object to be resolved. Either way, it first coordinates with the subordinates to lock the volume that owns the object to be resolved, generating an `rs_lockandfetch_start` event at each server in the DRG. That event transitions the coordinator to the `Csub1-LockVol` state and each subordinate to the `Sub1-LockVol` state. After locking the volume, a server generates an `rs_lockandfetch_end` event. That event transitions the coordinator back to the `ResCoord` state and a subordinate to the `ResWait` state.

If a directory is to be resolved, the coordinator then transitions to the `DirRes` state. While there, it coordinates the series of protocol phases described in Section 3.1. The phases are represented by the `{Csub, Sub}2-FetchLogs`, `{Csub,Sub}3-ParseLogs`, `{Csub,Sub}3.5-HandleInc`, `{Csub,Sub}4-InstallVV`, and `{Csub,Sub}5-MarkInc` states. States that begin with `Csub` are for the coordinator, while states that begin with `Sub` are for the subordinates. On the other hand, if a file is to be resolved, the coordinator transitions to the `FileRes` state, and then conducts file resolution.

When resolution is complete, the coordinator returns to the `ResCoord` state and instructs the DRG to unlock the previously locked volume. The unlocking takes place in the `{Csub,Sub}-UnlockVol` states. After the ViceResolve request is fulfilled, a `viceresolve_done` is generated by the coordinator, transitioning it to the `ResWait` state, where it is no longer a coordinator. If another resolution is requested, a new coordinator will transition to the `ResCoord` state, to conduct the requested resolution.

Supplemental state machine For our experiments, the Coda state machine was augmented with the supplemental state machine in Figure 3, which makes extensive use of auxiliary states to coordinate the steps of an experiment. The two state machines are joined through the arcs in Figure 3 that transition into and out of the box marked "Coda state machine." In addition to auxiliary states, the supplemental state machine also makes considerable use of auxiliary events and control modifiers. In the description of the state machine below, if an action is performed during a state, it will be a direct result of a control modifier that was triggered by that state, unless otherwise stated.

Coda servers begin in the `InitServer` state. After initializing, a server transitions to the `StartClient` state, from which Coda clients are launched. `SanityCheck0`

[1]Serialization was used to simplify server state machines.

[2]The state machine includes the state of both directory and file resolution. The state of file resolution helps in the interpretation of certain measures, such as end-to-end measures.

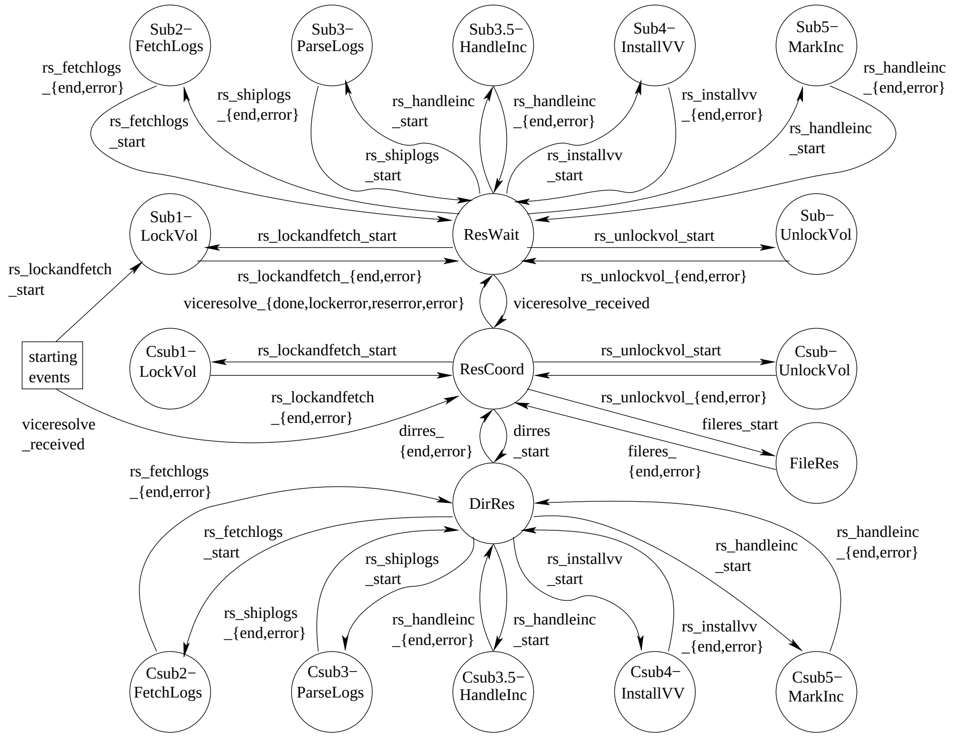

Figure 2. Coda resolution state machine

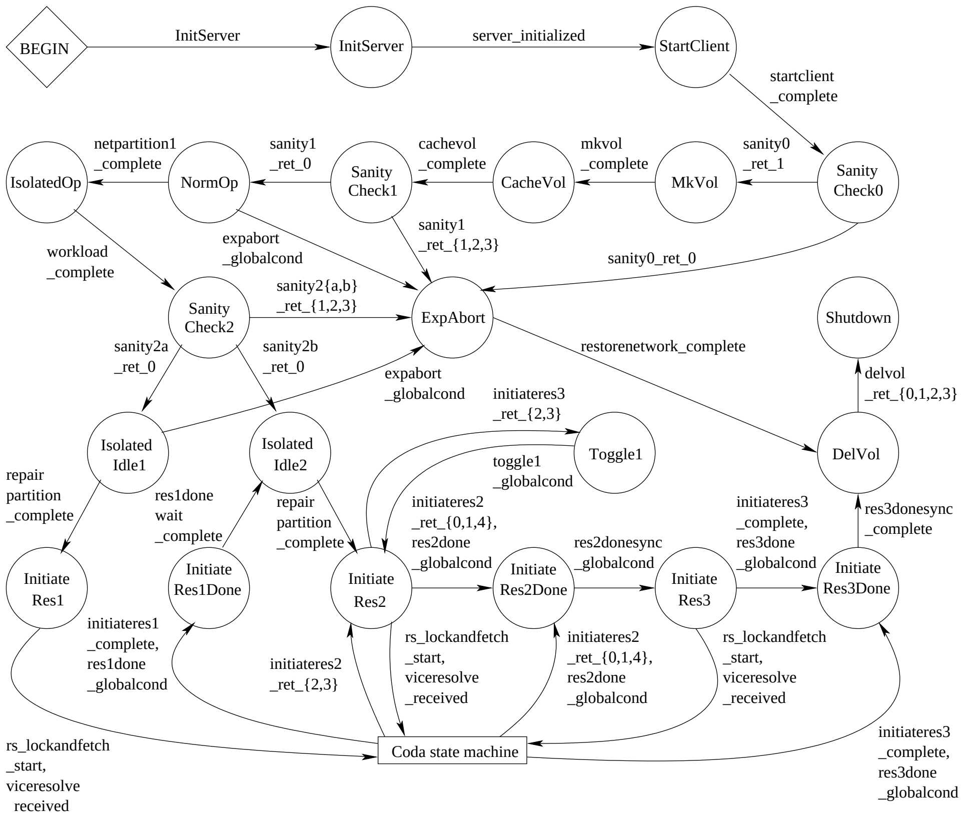

Figure 3. Supplemental state machine

is used to trigger a sanity check to ensure that all servers and clients properly initialized. The sanity check generates a returnval event to transition the server to the `ExpAbort` state and abort the experiment if the check is failed. Once each server makes it to the `MkVol` state, one of the servers creates a replicated volume and populates it with data for the experiment. Then, in the `CacheVol` state, each client's view of the volume is brought up to date. Once the `SanityCheck1` state is entered, another sanity check is used to ensure that the volume has been properly initialized and each client can access the volume. Assuming that the sanity check is passed, all of the servers meet in the `NormOp` state. The first network partition occurs there. We know the partition has completed when the `netpartition1_complete` event is generated, at which point all of the state machines will progress to the `IsolatedOp` state, in which a workload is applied by each Coda client. Upon completion of a workload at a host, the corresponding server transitions to the `SanityCheck2` state, in which we confirm that the workload performed correctly. If the setup is not completely correct, then the experiment is again aborted. Otherwise, the state machine transitions to `IsolatedIdle1` if the experiment requires a correlated network fault and `IsolatedIdle2` in the baseline case. From those states, the network partition is repaired, causing a transition to the `InitiateRes1` and `InitiateRes2` states, respectively.

When all of the servers are in an `InitiateRes`x state, a client uses an initiateres$x$ control modifier to request resolution of the experiment's replicated volume. The request generates a ViceResolve request at one of the servers, starting the resolution of the volume and its contents. As each server becomes involved in the resolution process, it transitions from the supplemental state machine to the Coda resolution state machine. The resolution process is completed or aborted with an `initiateres`x auxiliary event that causes each server to return to the respective `InitiateRes`x`Done` state.

Correlated network partitions are injected (for experiments requiring them) from within the resolution initiated by `InitiateRes1`. In all experiments, `InitiateRes2` is a state in which resolution is started and allowed to run to completion. `InitiateRes3` is used to update all clients on the contents of the resolved volume. `DelVol` uses a control modifier to clean up the volume at the end of the experiment, and during `Shutdown`, the experiment is completed, and servers and clients are shut down.

3.3 Faults and selected control modifiers

Here, we present the faults and some selected control modifiers used in our experiments. Table 1 specifies their injection triggers and the hosts at which they are injected. The faults are the two network partitions, each of which is injected using the network fault injector described in Section 2.2, in order to block TCP and UDP traffic on ports 2430 to 2433, i.e., Coda network traffic. netpartition1 is injected at both hosts when both file servers reside in the `NormOp` state. At host h1, it blocks network traffic from host h2, and at host h2 it blocks traffic from host h1. netpartition2 is only injected at host h1, where it blocks traffic from and to host h2. The trigger for netpartition2 is dependent on the study. In studies P3 and P4, x is replaced by `3` or `4` and $phasename$ is replaced by `ParseLogs` or `InstallVV`, respectively. This causes the correlated network partition to be injected into Phase 3 of the directory resolution protocol for study P3 and Phase 4 for study P4. netpartition2 is omitted from study B.

Table 1 also lists several control modifiers. The four sanity checks (sanity$\{$0,1,2a,2b$\}$) perform the checks described in Section 3.2. Each one produces a returnval event identifying the status of its sanity check and is triggered when both state machines are in the appropriate `SanityCheck` state. sanity2a is used for study B, while sanity2b is used for studies P3 and P4. sanity2a and sanity2b both perform the same actions but allow us to transition to different points in the state machine, depending on the study. repairpartition uses the network fault injector to repair network partitions, and is triggered when the two file servers are synchronized at either of the `IsolatedIdle` states. initiateres2 is one of the most important control modifiers. It requests resolution of the entire replicated volume after all of the faults have been injected. We computed many measurements from the resolution that initiateres2 requests. Its trigger and the state machines are designed so that initiateres2 can be injected multiple times if it returns an intermediate resolution result. When each initiateres2 finishes, it generates a returnval event indicating the status of the volume resolution, `initiateres2_ret_`x. When a final resolution outcome is reached, the `initiateres2_ret_`x event brings the server at host h1 from the Coda state machine into the supplemental state machine. Since the initiateres2 control modifier is injected only at host h1, the auxiliary event is generated only there. Therefore, the res2done control modifier is issued at host h2 to generate a globally conditioned event, `res2done_globalcond`, to transition the server at host h2 back into the supplemental state machine, where it can synchronize with the server at host h1.

4 Results

200 experiments were performed for each of our three studies. Experiments were executed on two Pentium II, 233 MHz machines with 128 MB of RAM running the Linux 2.4.17 kernel. The machines were located on a single 100 Mb Ethernet LAN. The following versions were used for

Table 1. Faults and selected control modifiers

Name	Type	Trigger	Host
sanity{0,1}	cm	`(h1-server:SanityCheck{0,1})&(h2-server:SanityCheck{0,1})`	h1,h2
netpartition1	fault	`(h1-server:NormOp)&(h2-server:NormOp)`	h1,h2
sanity{2a,2b}	cm	`(h1-server:SanityCheck2)&(h2-server:SanityCheck2)`	h1,h2
repairpartition	cm	`((h1-server:IsolatedIdle1)&(h2-server:IsolatedIdle1))` `|((h1-server:IsolatedIdle2)&(h2-server:IsolatedIdle2))`	h1,h2
netpartition2	fault	`((h1-server:Csub`x-$phasename$`)&(h2-server:Sub`x-$phasename$`))` `|((h2-server:Csub`x-$phasename$`)&(h1-server:Sub`x-$phasename$`))`	h1
initiateres2	cm	`((h1-server:InitiateRes2)&(h2-server:InitiateRes2))` `|((h1-server:InitiateRes2)&(h2-server:ResWait))`	h1
res2done	cm	`(h1-server:InitiateRes2Done)` `&((h2-server:ResWait)|(h2-server:InitiateRes2))`	h2

Coda packages: Coda 5.3.13, Coda kernel module 5.2.3, LWP 1.7, RPC2 1.11, and RVM 1.4. When the experiments finished executing, they were passed through Loki's analysis process to create a global timeline for each experiment and to determine the correctness of fault injections.

Before synthesizing results, we filtered improper experiments from consideration. Experiments are improper if they do not set up correctly, i.e., they fail a sanity check, or contain incorrectly injected faults (and/or control modifiers). We only needed to check the correctness of the netpartition2 fault, because when the state is reached in which any other fault or control modifier is to be injected, the state machines will wait in that state until it is injected. Therefore, if faults and control modifiers (other than `netpartition2`) are injected, they will be injected correctly.

4.1 Measure definitions

Three groups of measures were considered in our study. The first group of measures classifies the final outcome of the first uninterrupted resolution of the target volume, i.e., the resolution initiated by the initiateres2 control modifier. The returnval event that is generated by the final initiateres2 identifies the final resolution outcome. The first three measures in Table 2 correspond to the three possible final resolution outcomes. The mean of each measure gives us the fraction of experiments that produce the desired outcome.

The second group of measures considers the situation in which a temporary unavailability of resources prevents Coda from immediately producing a final outcome. When that occurs, an intermediate result noting the unavailability of resources is generated. The result is only an intermediate result because if the client continues to request that the target volume be resolved, one of the aforementioned final resolution results finally happens. Our experiments always produce a final outcome, because initiateres2 is repeated if an intermediate result occurs. The fourth through eighth measures of Table 2 pertain to temporarily unavailable resources.

Table 3. Final resolution outcome means

Study	# of Exps	res2_ correct	res2_ incorrect	res2_ manual
B	75	100.0%	0.0%	0.0%
P3	68	100.0%	0.0%	0.0%
P4	46	52.2%	0.0%	47.8%

The third group of measures pertains to end-to-end measures that estimate the time that clients spend waiting for resolution initiated by initiateres2, i.e., uninterrupted resolution. Each end-to-end measure is a sum of intervals. Each interval is measured from the time a Coda server receives the first ViceResolve request generated from an initiateres2 control modifier, to the time when the initiateres2 finishes and produces a returnval event. The final four measures in Table 2 make up the third group of measures.

4.2 Experimental results

Final resolution outcomes Table 3 shows our final resolution outcome findings. One of the most notable results is that none of the studies had experiments that resulted in incorrect resolutions. That is significant, because it tells us that data integrity was upheld. Next, we observe that the baseline case always resulted in automatic and correct resolution, which is what we expected. When we look at the results for study P3, we notice that all of the resolutions performed automatically and correctly. The directory resolution protocol was aborted when the correlated network partition was detected by the Coda servers. When the network was repaired and the client requested resolution again (during initiateres2), the directory resolution protocol was

Table 2. Measure descriptions

Measure	Description
res2_correct	1 if the target volume is automatically resolved correctly (i.e., Coda merges diverging replicas automatically in a way that is consistent with partitioned updates), 0 otherwise
res2_incorrect	1 if the target volume is automatically resolved incorrectly (i.e., Coda merges diverging replicas automatically but not in a manner that is consistent with partitioned updates), 0 otherwise
res2_manual	1 if the target volume requires manual resolution, 0 otherwise
res2_unavailable	1 if temporarily unavailable resources are experienced, 0 otherwise
res2_count	the number of times that initiateres2 must be invoked before producing a final resolution outcome
res2_count_g2	1 if res2_count is greater than 2, 0 otherwise
correct_if_g1_res2	only evaluated for experiments that observe more than 1 initiateres2 call; result is 1 if the target volume is automatically resolved correctly, 0 otherwise
g1_res2_if_correct	only evaluated for experiments whose target volume is resolved automatically and correctly; result is 1 if more than 1 initiateres2 call is observed, 0 otherwise
end2end_correct	end-to-end measure for experiments whose initiateres2 resolution is automatic and correct
end2end_manual	end-to-end measure for experiments whose initiateres2 resolution requires manual resolution
end2end-lockerr_{correct,manual}	end-to-end measure, for experiments with the indicated initiateres2 final resolution result, minus time spent waiting for the release of a volume lock from a previously failed resolution
end2end_{correct, manual}_x	portion of the end-to-end measure pertaining to the x^{th} initiateres2 call, for experiments with the indicated initiateres2 final resolution result

restarted. This behavior is as described in [11], and our experiments serve to verify it.

Lastly, we examine the final resolution outcomes from study P4. A correlated fault was injected during the phase of the directory resolution that is responsible for committing the directory resolution and converging on a storeid. We immediately see that P4 is the only study in which resolution resulted in something other than automatic and correct resolution. In fact, 47.8% of the time, the outcome for P4 requires manual resolution. That is a very significant result because of its potential impact on the volume's availability. Once Coda declares that manual resolution is required, a human must step in to converge the diverging replicas, and that could take a significant amount of time. Using Loki's measure support, we have confirmed that in each of the experiments from study P4 that require manual resolution, the interrupted Phase 4 ended in an `rs_installvv_error` event for at least one of the servers. Then, when directory resolution was restarted, the protocol omitted Phase 4 and performed Phase 5, in which the volume was marked as inconsistent, i.e., in need of manual resolution. On the other hand, in the experiments that resulted in automatic and correct resolution, none of the servers generated an `rs_installvv_error` event. In fact, the first resolution proceeded to the point that the resolved directory was committed despite the first network partition. However, not all of the files in the target volume were resolved. Thus, when initiateres2 was performed for the second time, the remaining unresolved files were resolved.

Table 4. Unavailable resource results

Measure	Study	# of Exps	Mean	Std Dev
res2_unavailable	B	75	0.0%	na
res2_unavailable	P3	68	0.0%	na
res2_unavailable	P4	46	52.2%	na
res2_count	B	75	1.0	0.0
res2_count	P3	68	1.0	0.0
res2_count	P4	46	1.5	0.5
res2_count_g2	P4	46	0.0%	na
correct_if_g1_res2	P4	24	100.0%	na
g1_res2_if_correct	P4	24	100.0%	na

Temporarily unavailable resources Next, we consider temporarily unavailable resources. Associated measures are listed in Table 4. First, we notice that exactly one initiateres2 is invoked for studies B and P3; only study P4 ever experienced unavailable resources. We also observe that the temporarily unavailable resource condition happens no more than once, resulting in two initiateres2 calls. Our final observation is that temporarily unavailable resources occurred in study P4 if and only if resolution started by initiateres2 resulted in automatic and correct resolution. We came to that conclusion because the means of both correct_if_g1_res2 and g1_res2_if_correct are 100.0%. Thus, experiments from study P4 either required manual resolution or experienced temporarily unavailable resources before resulting in correct and automatic resolution.

End-to-end results The final group of measures that we considered were end-to-end measures from the client's point of view. The measures we obtained are listed in Table 5 and graphically displayed in Figure 4. The figure presents end-to-end measures for four cases, each labeled with a study and final resolution outcome. For each case, the end-to-end measure is broken down into subcomponents in two ways. The first breakdown distinguishes which portions of the measure resulted from the first initiateres2 call versus a second initiateres2 call (if required). The percentage above the bar specifies the fraction of the bar due to a second initiateres2 call, while the percentage below the bar pertains to the first initiateres2 call. The second breakdown is labeled like the first, but distinguishes between time spent waiting for the previously locked volume to unlock versus time during which the system is successfully performing resolution work.

There are several points to consider. Most noticeably, studies containing correlated network partitions require nearly 12 to 14 times as much time to perform resolution as the baseline case. The reason is that attempts to lock the target volume during the second resolution fail because the subordinate, `h2-server`, maintains its lock on the target volume from the first resolution attempt, even though the protocol has been aborted due to the network partition. While `h2-server` holds the lock, the volume is unavailable. Furthermore, the client requesting the resolution blocks while resolution repeatedly fails to lock the already locked volume. Using Loki measures support, we have confirmed that this phenomenon occurred in all the experiments in studies P3 and P4.

According to [11, p. 71], the subordinate continually probes the coordinator in order to detect crashes (or network partitions). If the coordinator does not respond, the subordinate unlocks the volume; however, in our experiments, the subordinate did not unlock the volume until some time much later, presumably due to a timer expiration for the lock. When a subordinate is asked by a server to lock a volume, and the volume is already locked by a request from the same server, we hypothesize that the protocol could be improved if the subordinate released and relocked the volume. The subordinate could keep track of who asked it to lock the volume and for what purpose; if the same server later asked the subordinate to lock the volume, and the subordinate still had the volume locked from the first request, then the subordinate could unlock and relock the volume.

Another observation is that for experiments from study P4 that require manual resolution, the time spent performing resolution work is extremely small compared to the amount of resolution work in all the other cases. That is significant, because it tells us that if manual resolution is required, it is quickly detected and reported.

Our final observation from the end-to-end results concerns experiments from study P4 that result in automatic and correct resolution. As previously mentioned, those experiments always experience temporarily unavailable resources, thus requiring the client to request resolution twice. The interesting point is that the end-to-end measure is shorter for the P4 experiments that resulted in automatic and correct resolution than for the P3 experiments that resulted in automatic and correct resolution, even though study P4's experiments are the ones that experienced temporarily unavailable resources. While we are not exactly sure why this occurs, it does give us some insight into the nature of the temporarily unavailable resources. It would appear that the unavailability of resources is not due simply to a timer expiration, or we would expect the end-to-end measure to be larger for study P4. In addition, Figure 4 also shows us that in the correct case, some resolution work is being done during the first initiateres2 call. That may indicate that the correlated network partition causes some of the file system's resources to be temporarily tied up during the first initiateres2 call, requiring a second initiateres2.

5 Conclusion

This paper has presented an experimental evaluation of the Coda distributed file system using globally-state-triggered fault injection. Coda claims to provide high availability by tolerating network partitions and crash failures through the use of server replication and disconnected operation. We have focused our study on the effects of correlated network partitions on Coda's techniques for resolving partitioned updates. Our study demonstrates how a distributed protocol can be mapped to local state machines and combined with experiment state to form the local building blocks for the study's global state. Furthermore, we defined and computed a variety of measurements, including end-to-end metrics, based on the system's global state. Our results showed that Coda's resolution techniques never incorrectly resolved the target volume. However, in some cases, a correlated network partition caused the volume resolution to produce a result requiring manual resolution when the partitioned updates in the experiment should have been capable of being resolved automatically. Also, end-to-end resolution times from experiments involving correlated network partitions were 12 to 14 times larger than in the baseline case, due to time spent waiting to lock the previously locked target volume. Once the volume was lockable, we saw that Coda was able to quickly diagnose cases that it determined required manual resolution.

Table 5. End-to-end results

Study	*class*	# of Exps	end2end_*class* (s)		end2end-lockerr_*class* (s)	
			Mean	Std Dev	Mean	Std Dev
B	correct	75	21.0	0.3	21.0	0.3
P3	correct	68	287.8	14.2	21.9	3.3
P4	correct	24	258.8	3.2	23.6	2.1
P4	correct_1	24	240.5	3.0	-	-
P4	correct_2	24	18.3	0.4	-	-
P4	manual	22	248.2	7.9	0.64	0.02

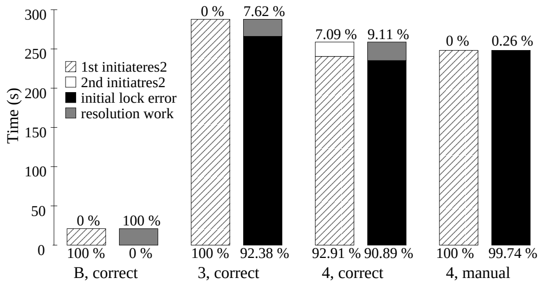

Figure 4. End-to-end comparison

Acknowledgments The authors would like to thank Jenny Applequist and Kaustubh Joshi for their contributions to this work. This material is based upon work supported by DARPA under grants F30602-96-C-0315, F30602-97-C-0276, and F30602-98-C-0187 and by the National Science Foundation under Grant No. CCR-00-86096. Any opinions, findings, and conclusions or recommendations expressed in this material are ours and do not necessarily reflect the views of DARPA or the National Science Foundation.

References

[1] J. Arlat, M. Aguera, Y. Crouzet, J. Fabre, E. Martins, and D. Powell. Experimental evaluation of the fault tolerance of an atomic multicast protocol. *IEEE Trans. on Reliability*, 39(4):455–467, Oct. 1990.

[2] D. Bhatt, R. Jha, T. Steeves, R. Bhatt, and D. Wills. SPI: An instrumentation development environment for parallel/distributed systems. In *Proc. of the 9th Int. Parallel Processing Symp.*, pages 494–501, 1995.

[3] R. Chandra, M. Cukier, R. M. Lefever, and W. H. Sanders. Dynamic node management and measure estimation in a state-driven fault injector. In *Proc. of the 19th IEEE Symp. on Reliable Dist. Systems*, pages 248–257, Oct. 2000.

[4] R. Chandra, R. M. Lefever, M. Cukier, and W. H. Sanders. Loki: A state-driven fault injector for distributed systems. In *Proc. of the Int. Conf. on Dependable Systems and Networks*, pages 237–242, June 2000.

[5] M. Cukier, R. Chandra, D. Henke, J. Pistole, and W. H. Sanders. Fault injection based on the partial global state of a distributed system. In *Proc. of the 18th IEEE Symp. on Reliable Dist. Systems*, pages 168–177, Oct. 1999.

[6] M. Cukier, D. Powell, and J. Arlat. Coverage estimation methods for stratified fault-injection. *IEEE Trans. on Comp.*, 48(7):707–723, July 1999.

[7] S. Dawson, F. Jahanian, T. Mitton, and T. L. Tung. Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In *Proc. of the 26th Int. Symp. on Fault-Tolerant Comp.*, pages 404–414, June 1996.

[8] K. Echtle and M. Leu. The EFA fault injector for fault-tolerant distributed system testing. In *Proc. of the IEEE Workshop on Fault-Tolerant Parallel and Dist. Systems*, pages 28–35, 1992.

[9] S. Han, K. G. Shin, and H. A. Rosenberg. DOCTOR: An integrated software fault injection environment for distributed real-time systems. In *Proc. of the Int. Comp. Perf. and Dependability Symp.*, pages 204–213, 1995.

[10] K. R. Joshi, M. Cukier, and W. H. Sanders. Experimental evaluation of the unavailability induced by a group membership protocol. In *Proc. of the 4th European Dependable Comp. Conf.*, pages 140–158, Oct. 2002.

[11] P. Kumar. *Mitigating the Effects of Optimistic Replication in a Distributed File System*. PhD thesis, Carnegie Mellon University, 1994.

[12] P. Kumar and M. Satyanarayanan. Log-based directory resolution in the Coda file system. In *Proc. of the 2nd Int. Conf. on Parallel and Distributed Information Systems*, pages 202–213, Jan. 1993.

[13] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proc. of the USENIX Winter 1995 Technical Conf.*, pages 95–106, 1995.

[14] F. Lange, R. Kroeger, and M. Gergeleit. JEWEL: Design and implementation of a distributed measurement system. *IEEE Trans. on Parallel and Dist. Systems*, 3(6):657–671, Nov. 1992.

[15] R. M. Lefever. An experimental evaluation of the coda distributed file system using the Loki state-driven fault injector. Master's thesis, University of Illinois at Urbana-Champaign, 2003.

[16] B. Noble and M. Satyanarayanan. An empirical study of a highly available file system. In *Proc. of the 1994 ACM SIGMETRICS Conf.*, pages 138–149, May 1994.

[17] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. on Comp.*, 39(4):447–459, Apr. 1990.

[18] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. K. Iyer. NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proc. of the 4th IEEE Int. Comp. Perf. and Dependability Symp.*, pages 91–100, Mar. 2000.

[19] D. T. Stott, M.-C. Hsueh, G. Ries, and R. K. Iyer. Dependability analysis of a commercial high-speed network. In *Proc. of the 27th Int. Symp. on Fault-Tolerant Comp.*, pages 248–257, 1997.