Using Belief to Reason About Cache Coherence

L. Mummert, J.M. Wing, and M. Satyanarayanan

May 1994
CMU-CS-94-151

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The notion of belief has been useful in reasoning about authentication protocols. In this paper, we show how the notion of belief can be applied to reasoning about cache coherence in a distributed file system. To the best of our knowledge, this is the first formal analysis of this problem. We used an extended subset of a logic of authentication [4, 5] to help us analyze three cache coherence protocols: a *validate-on-use protocol,* an *invalidation-based protocol*, and a new *large granularity protocol* for use in weakly connected environments. In this paper, we present two runs from the large granularity protocol. Using our variant of the logic of authentication, we were able to find flaws in the design of the large granularity protocol. We found the notion of belief not only intuitively appealing for reasoning about our protocols, but also practical given the optimistic nature of our system model.

Research is sponsored in part by the Air Force Materiel Command (AFMC) and the Advanced Research Projects Agency (ARPA) under contract number F19628-93-C-0193. Support also came from IBM Corporation, Digital Equipment Corporation, Intel Corporation, and Bellcore.

Keywords: distributed file systems, cache coherence, Coda file system, belief, logic of authentication

1 Introduction

In their seminal work on a logic of authentication [4, 5] Burrows, Abadi and Needham identify the central role played by *belief* in reasoning about the correctness of authentication protocols. They demonstrate the power of this reasoning by using it to identify errors and inefficiencies in a number of published protocols, one of which had been proposed as an international standard. The novel contribution of our work is the application of the notion of belief to a domain that has to our knowledge never been subjected to formal analysis: cache coherence in a distributed file system.

Caching of data at clients plays an important role in meeting the performance and scalability requirements of large distributed systems [11, 19]. Caching has also been exploited to mask temporary failures of communication [12, 16]. The value of caching is especially high when bandwidth and connectivity are at a premium. This situation arises in mobile computing, where *weak connectivity* is the norm. A weak connection is characterized by low bandwidth or intermittence.

The work described in this paper arose from our efforts to exploit weak connectivity in the Coda File System [21]. Since communication is expensive in weakly-connected environments, we sought to keep the volume and frequency of client-server communication to a minimum. This made our cache coherence protocol more complex than earlier protocols that had been developed for LAN environments. The need to ensure the correctness of this more complex protocol led us to explore techniques to reason systematically about it.

In our investigation, we realized that the notion of "belief" was at the heart of our cache coherence protocols. Informally stated, the correctness criterion for these protocols is as follows: *If a client believes that a cached file is valid then the server that is the authority on that file had better believe that the file is valid.* This insight led us to explore the logic of authentication. That logic allows one to focus on the beliefs of parties involved in an authentication protocol, and the changes in those beliefs as the parties communicate. Analogously, we need to reason about the behavior of clients and servers in a distributed file system, and the changes in their beliefs about cached data as they communicate.

As in the logic of authentication (henceforth referred to as the "BAN logic"), we reason about the beliefs of "principals"; in our system they are clients and servers. Our system model allows clients to update data during network partitions, such as in disconnected operation [16], when a client cannot contact any servers. Therefore, a connected client may reference a file that it and the server both believe to be valid, despite the possibility that the file is truly invalid (in a global sense) because of an update the server has not yet received. Determining global knowledge or "absolute truth" in this model is impossible [10]. Fortunately, we do not want or need to.

Reasoning about cache coherence is in some ways simpler, but in other ways more difficult, than reasoning about authentication. For example, we do not have to worry about malicious intent. We can also ignore replay attacks, since we can assume that duplicate suppression is performed by the underlying communication protocol. However, we cannot ignore two key characteristics of distributed systems: failures and transmission delay. Indeed, for these reasons we could not directly use the work in verifying cache coherence for multiprocessors [6, 17].

In the rest of this paper, we describe how we extended a subset of the BAN logic to help us analyze cache coherence protocols for distributed file systems. We applied our logic to three different protocols: a *validate-on-use protocol,* an *invalidation-based protocol*, and a new *large granularity protocol* for weak connectivity. Our strategy consists of first defining the state space and state transitions, then identifying all reachable states, and finally verifying that all possible runs of the protocol are correct (i.e., maintain cache coherence).

In the next three sections of this paper, we present our system model, our logic, and a statement of the correctness criterion for the cache coherence protocols. Then in Section 5, we describe the large granularity protocol, the most complicated of the three protocols that we have analyzed, and we present two of the fifteen possible classes of runs of the protocol. We then discuss the effects of failures and transmission delay on correctness. We close with a discussion of related work and a summary of our conclusions.

2 System Model

We designate hosts as clients or servers of the file system. Clients and servers communicate by sending messages to each other via remote procedure call [2]; each request made by one party requires a response from the other. To represent client C sending a message M to server S, we use this notation:

$$C \rightarrow S : M$$

Clients speak only to servers, not to other clients. We assume the underlying communication protocol addresses end-to-end concerns such as guaranteeing authenticity and eliminating duplicate messages.

Exactly one repository, which could be one server or a group of servers, is the authority for each file system object. In this paper, we use the generic term "server" for a repository. A file system object is any data contained by a server that may be cached at a client, including files, portions of files, file attributes, or version numbers. For now, it suffices to think of these objects as files; later in Section 5.1 they may also be version numbers.

The local state of a client, C, includes a set of cached data, $C.D$, and a set of beliefs, $C.B$, about objects in its cache. The local state of a server, S, includes a set of data objects, $S.D$, for which it is considered the authority, and for each client C, a set of beliefs, $S_C.B$, that includes which objects are present in C's cache and their validity.

The global state of the system is a tuple of all clients' and servers' local states, plus an *agreement set* A_{CS}, which determines for each data object d whose authority is S and is cached at C, whether the server and client copies are equal. It is this state variable that approximates global knowledge about the validity of all files. It represents pairwise knowledge, which is attained between connected pairs of clients and servers.

State transitions occur when a component of the global state changes. For the most part, this is when clients and servers exchange messages. We discuss the types of messages they exchange in our protocol in Section 5.

We reason about the presence or absence of file system objects cached at clients and their validity. An object is *valid* if it is the most recent copy in the system. Otherwise, it is *invalid*. Recency is determined by a timestamp associated with the file. The timestamp is replaced whenever the file is updated.

Since servers may not hear about updates immediately, validity is global knowledge and cannot always be determined. However, if C and S agree on an object, and S believes its copy is valid, then C should be able to conclude that its cached copy is valid. If S receives an update from a client other than C, then regardless of the global validity of the updated copy, S is justified in telling C that its copy of the object is now invalid.

A *run* of a cache coherence protocol begins with an initial message and ends with a final message. Each protocol has a predefined set of initial and final messages. Failures can terminate runs; however, failures are detected by message timeouts. If a message times out, the principal that sent the message considers it a final message. However, if a client and server both believe a run is in progress, then the run ends once both principals detect the failure. We give an example of this in Section 5.4.

Before a run, a client C considers all objects in its cache suspect; that is, it neither believes an object d is valid, nor believes d is invalid. During a run, C and S accumulate beliefs about d as a result of exchanging messages. At the end of the run, C and S discard their beliefs regarding the validity of d. After a run, C must again consider all cached objects suspect because it cannot check if they are valid, nor can S notify C that they are not.

3 Logic

Our logic is a subset of the BAN logic with a few extensions. Below, P and Q are principals, which are either clients or servers. S refers to a server and C refers to a client. We denote a message by X; a file system object by d. The constructs we use are:

P **believes** X	P behaves as if X is true.
P **sees** X **from** Q	P receives message X from Q.
P **controls** X	P is an authority on X.

The notions of belief and control are taken directly from the BAN logic. The statement P **believes** X is equivalent to $X \in P.B$, where $P.B$ is the belief set for principal P. The **sees** construct is derived from the BAN logic based on our assumptions about the underlying communication mechanism.

The following constructs, extensions to the BAN logic, are for reasoning about file system objects:

$d \in P$	P has a copy of d, i.e., $d \in P.D$. The copy held by P is denoted d_P[1].
valid(d_P)	The value of d's timestamp at P is greater than or equal to the timestamp associated with every other copy of d in the system.

Messages can contain the above two constructs and their negations. As in the BAN logic, a message X can consist of formulae, data, or both. For example, a message might contain a formula about some object d such as *valid*(d_C), or it might contain the object itself (simply d). We classify messages further based on their contents. For example, the various kinds of update requests form one class of messages. We denote an update request involving object d as *update*(d).

Belief sets can contain the above two constructs and their negations, as well as statements of the form P **believes** X.

The axioms in our logic are:

A1. $\forall d \in C.D$ $(C$ **believes** $d \in C)$
A2. $\forall d \notin C.D$ $(C$ **believes** $d \notin C)$
A3. P **believes** $X \Rightarrow \neg(P$ **believes** $\neg X)$

The first two axioms simply state that client C is allowed to believe what it knows about the contents of its cache.

The third axiom says belief sets must be internally consistent. In the BAN logic, beliefs are *stable*, meaning that once a principal holds a belief, it holds that belief for the duration of the protocol. Thus during their protocols, belief sets only grow. In contrast, in our system files may become invalid because of updates. Because of this, beliefs about the validity of files may change. Axiom 3 guarantees at most one of X and $\neg X$ appears in P's belief set. If a new belief is derived during a run that contradicts a currently held belief about d, the new belief *supersedes* the old one because it is based on more recent information. Thus if $C.B = \{$*valid*$(d_C)\}$, and a message arrives invalidating d_C, then $C.B$ would become $\{\neg$*valid*$(d_C)\}$.

The converse of A3, $\neg(P$ **believes** $X) \Rightarrow (P$ **believes** $\neg X)$, does not hold. In other words, absence of belief is distinct from belief of the opposite.

It may be the case that principal P has no beliefs regarding X, in other words it believes neither X nor $\neg X$. Since $X \in P.B$ is equivalent to P **believes** X, then $X \notin P.B$ is equivalent to $\neg(P$ **believes** $X)$. Thus, for example, if *valid*(d_P) does not appear in P's belief set, then we can say $\neg(P$ **believes** *valid*$(d_P))$. Again, this does not mean that P believes d_P is invalid, as explained above.

[1] Although P is a parameter, we find using it as a subscript more readable.

The inference rules in our logic are:

R1. The *visibility rule* says if a principal sees a message, it sees its components. This rule is taken from the BAN logic.

$$\frac{P \textbf{ sees } X, Y \textbf{ from } Q}{P \textbf{ sees } X \textbf{ from } Q, P \textbf{ sees } Y \textbf{ from } Q}$$

R2. The *message interpretation rule* says if a principal sees a message, it can believe that the sender believes what it said in the message. This is derived from the BAN message meaning and nonce-verification rules, and it follows from our assumptions about the underlying communication mechanism.

$$\frac{P \textbf{ sees } X \textbf{ from } Q}{P \textbf{ believes } Q \textbf{ believes } X}$$

R3. The *jurisdiction rule*, taken directly from the BAN logic, says if P believes Q is an authority on X, then P may believe whatever Q believes about X.

$$\frac{P \textbf{ believes } Q \textbf{ controls } X, P \textbf{ believes } Q \textbf{ believes } X}{P \textbf{ believes } X}$$

R4. The *update rule* says observers of an update invalidate old versions of the updated data. Below, $C' \neq C$, and S is the repository for d.

$$\frac{S \textbf{ believes } \textit{valid}(d_C), S \textbf{ sees } \textit{update}(d) \textbf{ from } C'}{S \textbf{ believes } \neg\textit{valid}(d_C)}$$

4 Goal of Cache Coherence

The goal of a cache coherence protocol is to ensure that no invalid object is ever portrayed as being valid. That is, for all clients C and objects d,

$$\text{if } C \textbf{ believes } \textit{valid}(d_C) \text{ then } \textit{valid}(d_C)$$

In practice, we cannot achieve this ideal, because our system model allows partitioned updates. Therefore the *correctness criterion* we use is: for all clients C, servers S, and objects d for which S is the repository,

$$\text{if } C \textbf{ believes } \textit{valid}(d_C) \text{ then } S \textbf{ believes } \textit{valid}(d_C)$$

Notice that the correctness criterion is defined on a per-object basis.

Unlike authentication, cache coherence is not a final system state to be achieved after running the protocol, but an invariant to be maintained while running it. To argue a cache coherence protocol correct, our obligation is to prove the invariance of the correctness criterion over each run of the protocol.

5 Protocol Analysis

We analyzed three different protocols: a *validate-on-use protocol,* an *invalidation-based protocol*, and a new *large granularity protocol* for weak connectivity.

In these protocols, a client may send a server a *fetch* request for new data, a *validation* of already cached data, or an *update*. Servers may respond to fetch and validation requests with new data, and an indication if already cached data is valid. In invalidation-based protocols, update requests cause servers to send *invalidation* messages to clients caching the updated data.

For example, in a validate-on-use protocol, the fetch and validation messages are initial messages, and the response to these messages is the final message. In such a protocol, runs are very short, and servers keep no state about client caches. In invalidation-based protocols, the client's response to an invalidation message is a final message. The response to a validation request is a final message if it is negative (i.e., the file is not valid).

For these protocols, the following assumptions apply to all runs:

S1. $\forall d \in S.D \;\; (S$ **believes** $\mathit{valid}(d_S))$
S2. $\forall d \in S.D \;\; d \in C \Rightarrow (C$ **believes** S **controls** $\mathit{valid}(d_C))$
S3. $\forall d \in S.D \;\; d \in C \Rightarrow (C$ **believes** S **controls** $\neg\mathit{valid}(d_C))$

The first assumption states that a server believes all the data it stores is valid. The last two assumptions say a server is the authority on the validity of data it stores.

In the rest of this section, we describe and analyze two runs of the large granularity protocol[2]. The description includes a definition of initial and final messages for the protocol.

5.1 Protocol Description

The cache coherence scheme used by the Coda file system is based on *callbacks*. When a client caches a file, the server promises that it will notify the client if the file changes. This is called a *callback promise*, or just a *callback*. If the file is updated, the server sends an invalidation message, called a *callback break*. If the file is invalidated, the client discards it. If a failure occurs, the client retains the file but considers it suspect. The client does not discard the file, because it is cheaper to validate the file when the failure is repaired than it is to re-fetch it. We assume for simplicity that a client does not discard a file unless the file is invalidated. Of course, in practice clients may discard files for other reasons, such as lack of space.

The large granularity cache coherence protocol extends the Coda scheme. To reduce client-server communication in failure-prone environments, callbacks may be maintained on *volumes* in addition to or instead of files. A *volume* is a collection of files forming a partial subtree in the file name space [22]. A file is contained in exactly one volume.

A callback on a volume constitutes proof that all cached files in the volume are valid. To establish a volume callback, the client caches the version number for the volume. The server increments the volume version number whenever a file in that volume is updated.

A run of this protocol concerns a file f, and optionally the version number v from volume V containing f. Before requesting v, the client must have at least one file in V in its cache, and all cached files in V must be valid. This requirement ensures the files at C correspond to the version number it receives.

A client may validate v just as it would a file. If it has both file and volume state at the beginning of a run, it may validate them in either order. If a client validates v successfully, it receives a callback for the volume. No further communication is necessary to read any file in the volume until the callback is broken or a failure occurs.

The initial messages for this protocol are any one of the following: a fetch for a file or a version number, or a validation for a file or a version number.

The final messages for this protocol are: the client's response to an invalidation of a file or volume, and a failed validation of a file or volume. Since a client can hold callbacks on both the file and its volume, the run ends when the *file* is discarded or rendered suspect. A failure or an invalidation for the file is sufficient to end the run. An invalidation for the volume ends the run only if there is no callback on the file.

[2] A complete analysis of all three protocols appears in [18].

Without loss of generality, we can analyze this cache coherence protocol by considering one client, C, one server, S, one file, f, and one volume, V, with version number v. Implicitly, the system includes at least one other client to represent remote updates. We can capture the system state as a tuple of four variables, $(C.D, C.B, S.B, A)$[3], where

- $C.D$ ranges over $\emptyset, \{f\}$, and $\{f, v\}$. This means if the volume version number is cached then so is a file from that volume.[4]
- A is the agreement set on the cached objects. It ranges over the following values:

$$\emptyset, \{f_C = f_S\}, \{f_C \neq f_S\}, \{f_C = f_S, v_C = v_S\}$$
$$\{f_C = f_S, v_C \neq v_S\}, \{f_C \neq f_S, v_C \neq v_S\}$$

Note that because the volume version number is updated whenever an object in the volume is updated, it is not possible for f to be invalid and v to be valid at the same time.

A run of the protocol maps some initial state $(C.D_i, C.B_i, S.B_i, A_i)$ to some final state $(C.D_f, C.B_f, S.B_f, A_f)$. The state space is restricted in the following ways. For all states, $(C.D, C.B, S.B, A)$, in a run:

1. For each object d (f or v) in $C.D$, $d_C = d_S$ or $d_C \neq d_S$ must be in A. This simply means if d is cached at C, d_C either matches the copy at S or it does not. If $C.D = \emptyset$ then $A = \emptyset$.
2. When an object is invalidated, the client must discard it. An invalidation for the file is an implicit invalidation for the volume. More precisely,
$$(f_C \neq f_S) \in A \Rightarrow C.D_f = \emptyset$$
$$(v_C \neq v_S) \in A \Rightarrow v \notin C.D_f$$
3. At the end of a run, either d is not cached, or it is cached and agrees with the server. This follows from 2 above, because once the client discovers d is invalid it discards it. Thus $A_f = \emptyset$ or $A_f = \{d_C = d_S\}$.

We classify runs by the initial and final cache contents of the client ($C.D_i$ and $C.D_f$) and by the initial agreement set (A_i)[5]. The state transition diagram for a client is shown in Appendix A. This diagram is for a single file; there is a separate, independent state machine for each file. Nodes are labeled with the contents of the client's cache. For each object in $C.D_i$, the contents of the agreement set determines which transition will be taken concerning that object. For example, a run consisting of a cache miss on file f and ending with a failure has initial system state $(\emptyset, \{f \notin C\}, \emptyset, \emptyset)$ and final system state $(\{f\}, \{f \in C\}, \emptyset, \{f_C = f_S\})$. This corresponds to a path from the upper leftmost state to the middle rightmost state in the state transition diagram. With this diagram, one can generate all possible runs of the protocol.

In the next two sections, we analyze two runs: (1) a cache miss with no failures, and (2) a successful validation followed by a communication failure. The first lets us introduce our notation and shows how we use the axioms and rules of our logic. The second serves as an example of a validation and a failure. For both, we assume initially that transmission of messages and failure detection are instantaneous; we discuss how timing affects correctness in Section 5.4.

We base our proof of invariance on either f_C or v_C, depending on which callback, if any, is established first. It is never the case that the client switches from depending on one type of callback to another during the run. If the proof of invariance concerns v_C, and f_C is contained by the volume whose version number is v_C, if C **believes** *valid*(v_C) then it can be confident that *valid*(f_C) as well.

[3] We do not need $S.D$ because our analysis includes only one server.

[4] We consider a simplified model consisting of only f and v, even though in practice it would take more than one file to make obtaining a volume callback worthwhile. We discuss this in Section 6.

[5] We do not need to consider A_f because it will either be empty or indicate agreement as stated in item 3 above.

C believes	Message	S believes	Notes
$f \notin C$			cache miss
	$C \rightarrow S: \;\; f \notin C$		request f
		$f \in C, \mathit{valid}(f_C)$	record callback promise
	$S \rightarrow C: \;\; f, \mathit{valid}(f_C), f \in C$		send f, callback status
$f \in C$ S **believes** $f \in C$ $\mathit{valid}(f_C)$			
	$\vdots$		
	$C' \rightarrow S: \;\; \mathit{update}(f)$		C' updates f
		$f \in C, \neg\mathit{valid}(f_C)$	C's copy stale
	$S \rightarrow C: \;\; \neg\mathit{valid}(f_C)$		callback break for f_C
$f \in C$ S **believes** $f \in C$ $\neg\mathit{valid}(f_C)$			supersedes $\mathit{valid}(f_C)$
$f \notin C$ $[S$ **believes** $f \in C]$ $[\neg\mathit{valid}(f_C)]$			C discards f C erases beliefs
	$C \rightarrow S:$		C responds to invalidation
		$[f \in C, \neg\mathit{valid}(f_C)]$	erase callback promise

Figure 1: Run Starting with a Cache Miss, Ending with an Invalidation

5.2 Cache miss, no failures

From the client's viewpoint, this run corresponds to the topmost path in the state transition diagram of Appendix A. The critical transitions for the client are a fetch of f, and an invalidation of f. The initial and final system states for this run are both $(\emptyset, \{f \notin C\}, \emptyset, \emptyset)$.

The run proceeds as follows. A request involving f is issued at C, however f is not present in C's cache. C sends the initial message to S requesting a copy of f. S records the fact that C is caching f ($f \in C$ on S). This is the callback promise. When C receives the response from S, it may use the data to service requests for f until S tells it otherwise. Since no failures occur in this case, eventually some other client updates f, rendering C's copy invalid. The server sends C an invalidation message (the callback break), causing C to discard its copy of f. C sends the final message to S indicating that it received the invalidation, and S discards its callback promise on f for C.

In Figure 1, we show the evolution of C's and S's beliefs as the protocol runs. The diagram is read left to right, then top to bottom. Time moves from top to bottom. Under the column named "C **believes** " we keep track of $C.B$, the set of client beliefs; similarly for the column named "S **believes** ." We show the entire belief set whenever an element of the belief set changes. For example, at the beginning of the run, f is not in C's cache. Using axiom 2, we derive C **believes** $f \notin C$, shown at the top of the "C **believes** " column. The notation "[x]" means $\neg(P$ **believes** $x)$. We do not use beliefs involving $f \in C$ in our proof, but we show them to motivate why certain messages are being sent.

As we walk through this example, we show that the invariant (rewriting the implication as a disjunction)

$$\neg(C \textbf{ believes } \mathit{valid}(f_C)) \vee (S \textbf{ believes } \mathit{valid}(f_C))$$

holds initially and is preserved across each step that changes the beliefs about f's validity. Initially, there are no beliefs in $C.B$ about the validity of f, because f is not even in C's cache. That means $\mathit{valid}(f_C) \notin C.B$, therefore $\neg(C$ **believes** $\mathit{valid}(f_C))$. Thus the invariant is established.

The cache miss causes C to send the initial message (to S), which does not change either belief set.

When S sends the second message, we derive S **believes** *valid*(f_S) using assumption S1 stated at the beginning of Section 5. In this message, S sends a copy of f to C. The copy is henceforth known as f_C. Since $f_C = f_S$ when S sends f, we can say S **believes** *valid*(f_C).

When C receives the second message, C **sees** f, *valid*(f_C), $f \in C$ **from** S. Using the visibility rule, we have C **sees** *valid*(f_C) **from** S[6]. Using the message interpretation rule, we derive C **believes** S **believes** *valid*(f_C). Using the jurisdiction rule instantiated with *valid*(f_C), we conclude C **believes** *valid*(f_C). But since S **believes** *valid*(f_C), the invariant still holds.

When the remote update to f occurs, S receives a message containing an update request involving f from some client $C' \neq C$. That is, S **sees** *update*(f) **from** C'. Using the update rule, we have S **believes** $\neg$*valid*(f_C). S sends C an invalidation message for f_C. If we assume the message arrives at C instantaneously, both parties change their beliefs at the same instant and the invariant still holds. Of course, the message does not arrive instantaneously. We discuss that in Section 5.4.

When C receives the invalidation message, we have C **sees** $\neg$*valid*(f_C) **from** S. Using message interpretation, we have C **believes** S **believes** $\neg$*valid*(f_C). Using assumption S3 and the jurisdiction rule instantiated for $\neg$*valid*(f_C), we conclude C **believes** $\neg$*valid*(f_C). This supersedes C **believes** *valid*(f_C). Since belief sets must be internally consistent (axiom A3), we know $\neg(C$ **believes** *valid*$(f_C))$ and the invariant holds.

C discards f and responds to the invalidation message, ending the run. Since C no longer has a copy of f, clearly *valid*$(f_C) \notin C.B$, and therefore $\neg(C$ **believes** *valid*$(f_C))$. Thus at the end of the run, the invariant holds.

5.3 Volume validation, followed by failure

From the client's viewpoint, this run corresponds to the path in Appendix A from state (f, v) to $(f, \mathbf{v})$ to (f, v). The critical transitions for the client are the validation of v and detection of a failure. The initial and final system states for this run are $(\{f, v\}, \{f \in C, v \in C\}, \emptyset, \{f_C = f_S, v_C = v_S\})$.

When this run begins, C already has volume and file state in its cache. C sends V's identifier and volume version number v to the server to determine if anything in V has been updated. In this case, the validation is successful (i.e., nothing has changed), so C may assume all cached state from V is valid. In addition, C receives a callback promise for V, meaning S will notify C if anything in V changes. At this point C may consider all files in V valid, though we show only f.

The run ends when a failure severs the connection between C and S. Here we simply show the failure and its effect, assuming it is detected instantly by both parties. In Section 5.4 we discuss how failures are detected, and the impact of failure detection on correctness.

This run is shown in Figure 2. As before, we walk through this run showing the invariant holds initially and after each step that changes either party's beliefs about the validity of f. Initially C cannot be certain of the validity of v_C, so *valid*$(v_C) \notin C.B$ and therefore $\neg(C$ **believes** *valid*$(v_C))$. Thus the invariant is established.

C sends the validation request for v to S. Using assumption S1, we obtain S **believes** *valid*(v_S). Since $v_C = v_S$ in this case, we have S **believes** *valid*(v_C), and the invariant still holds.

When C receives the response, we have C **sees** *valid*(v_C) **from** S. Using message interpretation we have C **believes** S **believes** *valid*(v_C). Using assumption S2 and the jurisdiction rule, we have C **believes** *valid*(v_C). Since S **believes** *valid*(v_C) the invariant holds.

When S detects the failure, it discards its beliefs about C. This includes beliefs about which objects C has cached, and the validity of those objects. When C detects the failure, it discards its beliefs about S, and the validity of objects in its cache. It retains beliefs about the presence or absence of objects in its cache; these beliefs are always derivable using the axioms, because they are based on strictly local information.

[6] In practice, it is not necessary for S to include *valid*(f_C) in the response to a fetch request. The client C simply assumes that data received from a server is valid.

C believes	**Message**	**S believes**
$f \in C, v \in C$		
	$C \rightarrow S: \quad v \in C$	
		$v \in C, \mathit{valid}(v_C)$
	$S \rightarrow C: \quad v \in C, \mathit{valid}(v_C)$	
$f \in C, v \in C$ $\mathit{valid}(v_C)$ S **believes** $v \in C$		
	$\vdots$	
	failure	
$f \in C, v \in C$ $[S$ **believes** $v \in C]$ $[\mathit{valid}(v_C)]$		$[v \in C, \mathit{valid}(v_C)]$

Figure 2: Run Starting with a Validation, Ending with a Failure

5.4 Timing

Failures and transmission delay affect the correctness of our cache coherence protocols. While a message from a server to client C is in transit, C may believe, however briefly, that its copy of f is valid when it is not. Thus, without assuming instantaneous transmission of messages, the case analysis in section 5.2 is incorrect because during the time it takes for the server's invalidation message to be received by the client, the client still believes the file is valid. While transmission time is often assumed to be negligible in LAN-based environments, this assumption is not valid in weakly connected environments. However, the transmission time is bounded by the timeout period used by the underlying communication protocol, denoted by β. If a message is not acknowledged within β after it is sent, the sender declares a failure. The timeout period is a system parameter, and is usually on the order of a minute.

It also takes time for clients and servers to detect failures. During the interval between the occurrence of a failure and its detection, it is possible for a client to use invalid data because the server was unable to notify the client of updates. This interval, denoted by τ, defines a window of vulnerability for the protocol. To bound the failure detection interval, clients and servers probe each other periodically, and declare failures if messages time out. These probe messages serve as final messages when failures occur.

Let β be the message timeout period as above, and let ρ be the probe interval. Assume clients and servers use the same probe interval, but do not necessarily probe each other at the same time. Then the failure detection interval $\tau = \rho + \beta$ at most.

Our notion of correctness is bounded by τ. We call a protocol τ*-correct* if the interval in which it does not meet the correctness criterion is at most τ. In Coda τ is composed of a probe interval of 10 minutes, and a message timeout of 15 seconds. A 0-correct protocol obeys the correctness criterion strictly. Even a validate-on-use protocol, such as in early versions of the Andrew File System [20] and Sprite [19], cannot achieve 0-correctness because of transmission delay.

The timetable in Figure 3 shows the worst-case behavior of a system whose failure detection interval is τ. We begin in the middle of a run, where C has f cached and a callback promise from S. Both C and S believe f_C is valid. Let t_p be the latest time at which C probes S successfully before the failure, and let t_f be the time at which the failure occurs. Eventually either C or S sends a message and discovers the failure. If the principals detect the failures through probes, C is still correct in believing that f_C is valid, even though the principals are likely to detect the failure at different times.

The worst case occurs if another client updates f while C is partitioned from S, but before C has detected the failure. S tries to break C's callback on f at time t_i but fails. This is a *lost callback*. As far as S is concerned, the run is over, and it discards its beliefs about C. However, the run is not over until C detects the failure. Until then, C

C believes	**Message**	**S believes**	**Notes**
$f \in C, \mathit{valid}(f_C)$		$f \in C, \mathit{valid}(f_C)$	
S **believes** $f \in C$			
	$\vdots$		
	$C \rightarrow S:$ *probe*		$t_p,$ probe successful
	$\vdots$		
	failure		$t_f,$ C and S partitioned
	$\vdots$		
	$C' \rightarrow S:$ *update*(f)		C' updates f
		$f \in C, \neg \mathit{valid}(f_C)$	
	$S \rightarrow C:$ $\neg \mathit{valid}(f_C)$		t_i
		$[f \in C, \neg \mathit{valid}(f_C)]$	$t_i + \beta,$ S declares failure
	$\vdots$		
	$C \rightarrow S:$ *probe*		$t_p + \rho$
			$t_p + \rho + \beta = t_p + \tau,$
			C declares failure
$f \in C$			
$[S$ **believes** $f \in C]$			C erases beliefs
$[\mathit{valid}(f_C)]$			

Figure 3: Worst Case Behavior During a Failure

believes f_C is valid when it is not. This interval is largest when the failure and the update occur immediately after t_p. At time $t_i + \beta$, when S gives up hope of ever reaching C with its invalidation message, C is still blissfully ignorant of the status of f. C does not discover a problem until it tries to contact S at $t_p + \rho$. It is not until $t_p + \tau$ that C declares failure and demotes f to suspect status. Thus τ is the longest period in which C can believe f is valid when it is not.

6 Evaluation

Not surprisingly, formal analysis gave us a better understanding of our protocol. Below we discuss more specifically how the analysis helped to correct bugs in our design. We conclude this section with a discussion of some of the simplifications we made to our model, and how our definitions could be extended.

6.1 Benefits of Formal Analysis

When we first designed the large granularity protocol, we began with an informal, narrative specification. This resulted in an underspecified protocol. Initially we thought there were ten classes of runs; after the formal analysis we realized there were fifteen. The runs we missed fell into two categories. The first involves a looping behavior that can occur if a client holds both file and volume callbacks. If the volume callback is broken, the run does not end because the file callback is still present. The file may still be used without contacting the server. Unfortunately, the volume callback may be re-established and broken ad infinitum until the file callback is lost or broken. The loop is visible in the state transition diagram of Appendix A, between states $(\mathbf{f}, 0)$ and $(\mathbf{f}, \mathbf{v})$. In practice, this loop is avoidable by setting a policy at the client which determines when to obtain a volume callback. While we realized that this looping behavior could occur, we did not realize how pervasive the behavior could be. It can occur in any state in which a file callback is held, which happens in most runs.

The second category involves ordering: if both f and v are present at the beginning of the run, they may be validated in either order. This is depicted in the state transition diagram of Appendix A, by having transitions for both f and v from the bottom leftmost initial state. It is legitimate for f to be validated first if conditions do not favor establishing a volume callback upon connection, for example, because of a high rate of remote updates in the volume.

Different orders may result in different runs (e.g., if f is valid but v is not).

While writing the proofs for the large granularity protocol, we were pleasantly surprised that we could base the proof on whatever piece of data for which the client established a callback first. That is, either we reasoned about f being valid or v being valid. There is one exception, which does not occur in our implementation. If a client obtains a file callback while holding a volume callback, the client could lose the volume callback and still continue the run. A notion of hierarchy (i.e., x is "contained in" y) would help to switch the proof from one data type to the other cleanly.

Formal analysis also helped in generating test cases, and early in testing we uncovered a bug in the implementation that would not have harmed the correctness of execution, but the efficiency. In the implementation, the client's volume state consists of two fields: the version number, and the callback status. When a volume callback was broken, the client cleared only the callback status field. This is correct, because the client checks the callback status field to determine if the client has a callback on the volume. However, because the volume version number was still present, the client attempted to validate it with the server on reconnection. The validation was a waste because it was doomed to fail.

6.2 Extensions

Our analysis is simplified in two respects. First, we consider only a single f and v, even though in practice it would take more than one file to make obtaining a volume callback worthwhile. Given our simplified model, we must exclude the states where $C.D = \{v\}$ because there must be at least one file present to obtain a volume callback. In practice, this value for $C.D$ is permissible provided there are other files in volume V cached at the client.

Second, although we allow a repository to consist of a group of servers, our analysis ignores some of the practical aspects of replication. For example, if the client uses a replicated volume, it must collate responses from multiple servers. One complication is that the client may receive responses from only a subset of the servers because of network partitions. A small change to the definition of a run takes care of this problem. For a replicated service, the run ends when the client detects a change in the number of servers with which it is communicating. This definition is natural because if the number shrinks, a callback may be lost from a server that disappeared. If the number grows, a newly available server may hold updated versions of cached data.

In addition to allowing us to reason about a replicated service very naturally, the notion of a run could be extended in other ways. For example, some systems incorporate expiration times in their cache coherence mechanism [8]. We can address this by extending the definition of a run such that when expiration occurs, the run ends.

7 Related Work

Formal verification of cache coherence protocols in the hardware domain has been done by MacMillan and Schwalbe for the Encore Gigamax multiprocessor [17] and by Clarke and his colleagues [6] for the IEEE Futurebus+ standard. In both cases flaws were found during the process of verification. Because they were working in the hardware domain, they were able to ignore two aspects of distributed systems that we cannot: failures and transmission delays. They also have the additional advantage of working in a finite state space and were able to do an exhaustive case analysis using symbolic model checking techniques [7]. We could view our simplified model (single server, single client, etc.) as a finite state machine, as depicted in Appendix A, and thus complement our proof-theoretic analysis with model checking.

Network communication protocols must cope with failures and latency as we do. Formal specifications of these protocols (e.g., [3, 24]), written in languages such as LOTOS [13] and Estelle [14], typically abstract away from state and highlight instead the behavior of the communicating parties as interleavings of their events. We not only have to accommodate possible failure and timeout events, but we also need to reason explicitly about client and server state, e.g., what files are cached. We found for our work that our correctness condition is most naturally specified as a state predicate to be shown invariant over a small, finite set of state transitions rather than as a property of an infinite set of interleavings.

There is an abundance of specification logics based on first-order predicate logic, set theory, and/or algebras, embodied in languages like Z [23], VDM [15], and Larch [9], but they are more general than we need. We could have easily defined our domain of discourse (clients, servers, RPC, faults, transmission delays, belief and agreement sets, etc.) in terms of these more generic primitives, but we intentionally chose a modal logic that would let us highlight the essence of our correctness condition—belief.

Just as Bentley advocates inventing and using "little languages" for special-purpose programming [1], we suggest that "little logics" are appropriate for special-purpose reasoning. Our extended subset of the BAN logic is just right for our purposes; it is specific enough so that we do not need to define primitives such as belief from first principles, but general enough so that one can apply it to different kinds of protocols.

8 Conclusions

Designers of large distributed systems must cope with the the fact that failures are the rule, not the exception. Ideally, these systems should function despite them. This means that clients should operate with a certain amount of autonomy; when failures occur, there should be mechanisms to allow clients to reduce their dependence on servers. Optimistic approaches for data access, which allow data access and updates during partitions, are ideal for this purpose – they maintain a high degree of data availability, but at the expense of global consistency. By its very nature, optimism is based on belief, not on knowledge. Hence, we find belief a practical notion for reasoning about correctness in distributed systems.

Using our extensions to a subset of the BAN logic we were able to capture client and server behavior intuitively and succinctly. Rather than define a new logic, we were fortunate to have a logic we could apply to our domain. The notion of belief for reasoning about cache coherence for file systems held intuitive appeal to us when considering alternative formal approaches. The underlying state machine model suffices for describing relevant local and global state components, and critical state transitions. The pure state machine approach would require encoding belief in terms of state variables, and reasoning about those variables. Using belief allows us to reason at a higher level of abstraction.

A little formalism goes a long way. We discovered serious flaws in earlier designs of our protocol, and inefficiencies in its implementation. By characterizing formally the classes of runs of our protocol, we could easily check that we covered all cases that we expect to encounter in practice.

Larger distributed systems will require more complex resource management software to meet more demanding performance and availability requirements. Judicious use of suitable formalism can help increase our confidencein the correctness of the systems we build.

9 Acknowledgements

We thank Martin Abadi and Mark Tuttle for their comments on our extended abstract, and especially for their help in our understanding some of the subtleties of the BAN logic.

References

[1] Jon Bentley. Programming Pearls: Little Languages. *Communications of the ACM*, 29(8), August 1986.

[2] Andrew D. Birrell and Bruce J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[3] G.v. Bochmann and C.A. Sunshine. Formal Methods in Communication Protocol Design. *IEEE Transactions on Communications*, 28(4):624 – 631, April 1980.

[4] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. Technical Report 39, DEC Systems Research Center, February 1989.

[5] Michael Burrows, Martin Abadi, and Roger Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[6] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness. Verification of the Futurebus+ Cache Coherence Protocol. Technical Report CMU-CS-92-206, Carnegie Mellon University School of Computer Science, October 1992.

[7] E.M. Clarke, J.R. Burch, O. Brumberg, D.E. Long, and K.L. McMillan. Automatic Verification of Sequential Circuit Design. *Phil. Trans. R. Soc. London*, 339:105–120, 1992.

[8] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *The Twelfth ACM Symposium on Operating Systems Principles*, pages 202–210. ACM, December 1989.

[9] J.V. Guttag, J.J. Horning (eds.) with S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[10] J.Y. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. In *Proceedings of the Third ACM Symposium on Principles of Distributed Computing*, pages 50 – 61, 1984.

[11] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[12] L.B. Huston and P. Honeyman. Disconnected Operation for AFS. In *Proceedings of the USENIX Symposium on Mobile & Location Independent Computing*, pages 1 – 10, August 1993.

[13] Information Systems Processing – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behavior. Technical Report ISO 8807, International Standards Organization, 1988.

[14] Information Systems Processing – Open Systems Interconnection – Estelle – A Formal Description Technique Based on an Extended State Transition Model. Technical Report ISO 9074, International Standards Organization, 1989.

[15] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.

[16] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[17] K. McMillan and J. Schwalbe. Formal Verification of the Encore Gigamax Cache Consistency Protocol. In *Proceedings of the 1991 International Symposium on Shared Memory Multiprocessors*, April 1991.

[18] Lily B. Mummert, Jeannette M. Wing, and M. Satyanarayanan. Using Belief to Reason About Cache Coherence. Carnegie Mellon University, School of Computer Science Technical Report, in preparation.

[19] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134 – 154, February 1988.

[20] M. Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, and Michael J. West. The ITC Distributed File System: Principles and Design. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 35–50, December 1-4 1985.

[21] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4), April 1990.

[22] R.N. Sidebotham. Volumes: The Andrew File System Data Structuring Primitive. In *European Unix User Group Conference Proceedings*, August 1986. Also available as Tech. Rep. CMU-ITC-053, Carnegie Mellon University, Information Technology Center.

[23] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1989.

[24] A.J. Tocher. LOTOS and the Formal Specification of Communication Standards: An Example. In *Formal Methods: Theory and Practice*. P.N. Scharbach, editor. CRC Press, Inc., 1989.

A Client State Transition Diagram

The client's state transition diagram for a file and its containing volume is shown in Figure 4. The node labels combine $C.D$ and $C.B$. We denote the file by f and the volume version number by v. A "0" means the object is not present in the cache. For example, the state $(f, 0)$ means $C.D = f$ and $(f \in C) \in C.B$. An object in boldface means the client has a callback for the object (*valid*$(d_C) \in C.B$). The states with no callbacks $((0,0), (f,0), (0,v),$ and $(f,v))$ are either initial or final states. For each object in $C.D$, the agreement set A determines the first state transition concerning that object. For example, if $C.D = f$ and $A = \{d_C \neq d_S\}$, when C references the object its next transition will be a failed validation for f.

Looping is possible between states $(\mathbf{f}, 0)$ and $(\mathbf{f}, \mathbf{v})$. In theory, a client could repeatedly obtain and lose the callback for v, while it holds the callback for f. This is false sharing, and should be avoided. In practice, a reasonable policy for obtaining volume callbacks should prevent this.

There are two transitions that we do not show (or use), but could be added as optimizations. Both are from the state $(f, \mathbf{v})$. In this state, all of the cached objects in V are valid. The first transition would allow a client to obtain a file callback on f in case the volume callback is broken ($(f, \mathbf{v})$ to $(\mathbf{f}, \mathbf{v})$). The second transition is an invalidation for f ($(f, \mathbf{v})$ to $(0, 0)$). If a client holds a volume callback only, the server does not have any information on which objects the client has cached. If object f is updated, the server breaks the callback for v. If it sent the identifier of f instead, the client would interpret the message as an invalidation for v as well as f. If it has a copy of f, it would discard it. This is a small optimization that will save a validation for f that is doomed to fail.

The diagram is simplified in that we do not show transitions associated with error conditions, such as message timeouts.

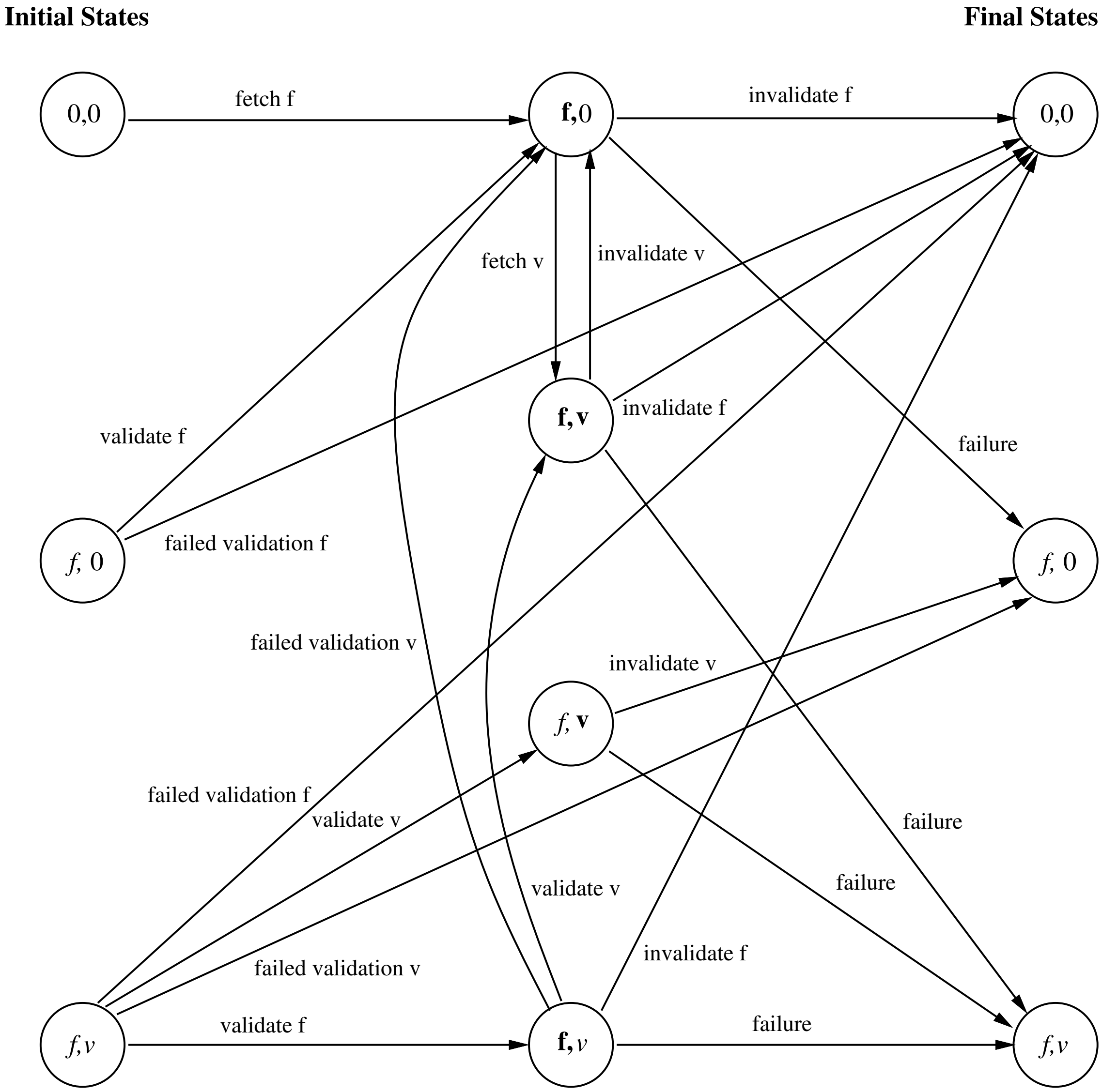

Figure 4: Client State Transitions

This is the state transition diagram for the client for a file-volume pair (f, v). A "0" means the object (f or v) is not present in the cache. An object in boldface means the client has a callback for the object.