Experiences with Formal Specification of Fault-Tolerant File Systems

Roxana Geambasu
University of Washington
roxana@cs.washington.edu

Andrew Birrell
Microsoft Research
birrell@microsoft.com

John MacCormick
Dickinson College
jmac@dickinson.edu

Abstract

Fault-tolerant, replicated file systems are a crucial component of today's data centers. Despite their huge complexity, these systems are typically specified only in brief prose, which makes them difficult to reason about or verify. This paper describes the authors' experience using formal methods to improve our understanding of and confidence in the behavior of replicated file systems. We wrote formal specifications for three real-world fault-tolerant file systems and used them to: (1) expose design similarities and differences; (2) clarify and mechanically verify consistency properties; and (3) evaluate design alternatives. Our experience showed that formal specifications for these systems were easy to produce, useful for a deep understanding of system functions, and valuable for system comparison.

1. Introduction

Fault-tolerant, replicated file systems have become a crucial component of today's dependable enterprise datacenters. For example, the Google File System (GFS) [9], Niobe [16], and Dynamo [7] underlie many of the web services offered by Google, Microsoft, and Amazon.com, respectively. Many other fault-tolerant file systems have been developed in academic settings, as well (*e.g.*, [15, 19]). All of these systems are extremely complex, including sophisticated asynchronous protocols, e.g., for replica consistency, recovery, and reconfiguration.

Despite their complexity, fault-tolerant file systems have typically been described only in a few pages of prose, which can be incomplete, inaccurate, or ambiguous. This makes reasoning about and proving system properties hard and error-prone. In contrast to prose, formal specifications in a language such as TLA+ [13] are unambiguous and provide solid grounds for model checking and formally proving system properties. The advantages of formal specifications have been previously reported for various types of systems, e.g.: caches [11], space shuttle software [6], and local and distributed file systems [18, 20].

We wished to explore how formal specifications and methods can help in understanding, comparing, and proving properties of another important class of systems: *fault-tolerant, replicated file systems*. To do this, we wrote formal specifications for three real-world, successful fault-tolerant file systems – GFS, Niobe, and Chain [19] – and used those to analyze, compare, and prove properties of the systems. This paper presents our experience writing and using those formal specifications. Overall, we found that formal specifications improve understanding of system functioning, enable better comparison, and are reasonably easy to produce.

We found specifications particularly useful for three purposes. First, specifications help crystallize differences and similarities of the systems' mechanisms. For instance, we find that GFS and Niobe have substantial overlap in mechanisms; our specification isolates common mechanisms and provides a clear view of what is similar and different. Second, specifications enable understanding and mechanical verification of the systems' consistency semantics. To reason about a system's consistency, we reduce the system to a much simplified analog (called a *SimpleStore*), and use refinement mappings [1] to verify that the system implements its SimpleStore. We then reason about and compare consistency properties of SimpleStores. Third, specifications enable comfortable experimentation with alternative system designs, which can be a valuable tool for a designer.

Our approach is pragmatic. While our specifications, in principle, enable full formal proofs [11], we rely on model checking of limited instances of the systems to confirm properties comfortably. By revealing various ways in which specifications are valuable in fault-tolerant file system analysis and comparison, we hope to convince system builders of the utility of specifying their own systems.

After providing some background (Section 2), we demonstrate the three usages of specifications (Sections 3, 4, and 5). We then review previous work (Section 6) and share some lessons from our practical experience (Section 7).

2. Background

2.1. Overview of the Studied Systems

In the three studied systems, each data object is stored at a group of replicas (groups can overlap), and the group is managed by a single master. The systems are reconfigurable, allowing failed or disconnected replicas to be removed from the group and new replicas to be added.

GFS. GFS provides a file-level write/append/read interface to clients. Files stored in GFS are partitioned into fixed-size chunks, each of which is replicated by a group. The master assigns a unique primary to each group. To perform a chunk write or append, the client sends the data to all replicas and then submits a write request to the primary, who acknowledges the write if all replicas have succeeded. To read from a chunk, the client goes to any of the chunk's replicas. Although in the published paper the master was not guaranteed to be reliable, we will assume it is here, to enable comparison to Paxos-based Niobe and Chain.

Niobe. Niobe offers an object-level read/write interface, where the object is the replication unit. Similarly to GFS, a unique primary exists for each group. To perform a write, the client submits the data to the primary, which writes it to disk and forwards it to the secondaries. The secondaries perform the write and acknowledge it to the primary. If any secondary fails to ACK the write in a timely manner, the primary proposes to the master that the failed replica(s) be removed from the group. After all replicas have ACKed the write (or have been removed), the primary responds to the client with success if the write succeeded at a configurable number of replicas, or with error otherwise. To read an object, the client goes to the primary.

Chain. Chain imposes a structure on the replica group: replicas are arranged in a chain. Writes are sent to the head of the chain and travel along the chain toward the tail, where they get acknowledged. Reads are sent to the tail, which returns its local value. While GFS and Niobe support network partitions, the original Chain paper implicitly assumes no network partitions. For example, it does not specify how to prevent a client from reading from a stale, but still alive tail. We assume in this work no network partitions for Chain.

2.2. TLA+ and Refinement Mappings

TLA+ is a formalism based on temporal logic, especially suited for specifying asynchronous distributed systems [13]. To specify a system, one describes its allowed behaviors using a state-machine approach. One specifies the variables that compose the system's state, a set of initial states, and the transitions leading from one state to another. A TLA+ specification can be enhanced with properties, which can be model-checked using the Temporal Logic Checker (TLC [14]). Because TLC exhaustively checks a system's state space, which is typically exponential, it can generally be used only on small instances of a system.

A refinement mapping [1] is a technique used to reduce one specification to another. Using refinement mappings, we reduce our specification of each system to a simple model of the system (called a *SimpleStore*). Figure 1 illustrates a refinement mapping: it maps a system's state space onto the SimpleStore's state space. A system *implements* its SimpleStore if all the system's client-visible behaviors can be mapped onto valid SimpleStore behaviors.

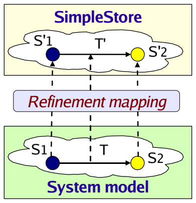


Figure 1. Refinement mapping from a system to its SimpleStore. Clouds represent state spaces. The refinement mapping maps the system's states onto states in SimpleStore. System states S_1 and S_2 map onto states S'_1 and S'_2, respectively (S'_1 and S'_2 may be the same state). The mapping is valid if for any S_1 and S_2, for any system transition T from S_1 to S_2, there exists a SimpleStore transition T' leading from S'_1 to S'_2 (possibly the identity transition).

We do not attempt to *prove* implementations. Instead, we specify the mappings as TLA+ properties, and *model-check* them for limited instances of the systems (three replicas). Of course, to prove implementation for any instance, one can perform full proofs. Such proofs, although in principle enabled by our specifications [11], are out of scope here. Still, our model checking covers the typical setting in industry systems like GFS, which do three-way replication.

Having verified that a system implements its SimpleStore, proving history-based consistency properties about the system (*e.g.*, linearizability) is known to be reducible to proving them for its SimpleStore [11], which is significantly easier than reasoning about the whole system.

3. Comparing System Mechanisms

We produced TLA+ specifications for all three of the systems. For GFS and Chain, specifications are based on published papers describing the systems and (email) conversations with the systems' designers; for Niobe, one of the designers participated in this work and is a co-author of this paper. Table 1 provides the sizes of our specifications and the time to write them. For each system, we specified at least how reads, writes, and replica removal are done. For Chain, we also specified the recovery mechanism. Due to the expressiveness of a formalism such as TLA+, specifications distill core replication mechanisms and protocols from the systems' complexity. As a result, our specifications are small (500 TLA+ lines, or about 10 pages), yet precise, high-level models of the systems. Overall, we found specifications to be extremely helpful for an in-depth understanding of systems, as well as reasonably easy to produce.

Specifications also prove valuable for a crisp comparison of the mechanisms in different systems. While a detailed examination of the specifications would show how the key differences and similarities stand out clearly in TLA+, we

	Chain	Niobe	GFS
TLA+ Lines	410	480	705
Time to write	3 weeks	2 weeks	2 weeks

Table 1. The three TLA+ specifications. For each system, we show the TLA+ specification size, and the time to produce the first working version by one person, without prior TLA+ knowledge. The Chain specification was the first to be written, and took longer due to lack of experience with TLA+. The GFS specification is significantly longer, as we specify writes and appends separately.

```
GFS                                    | Niobe
FinalizeWrite(prim, w) ≜               | FinalizeWrite(prim, w) ≜
 ∀ r ∈ Replicas :                      |  ∀ r ∈ Replicas :
    ∨ AckedWrite(prim, r, w)           |     ∨ AckedWrite(prim, r, w)
    ∨ Timeout(prim, r, w)              |     ∨ ( ∧ Timeout(prim, r, w)
                                       |         ∧ r ∉ group)
```

Figure 2. Design differences in TLA+. The figure shows TLA+ snippets from the Niobe and GFS modules. The purpose of this figure is *not* TLA+ instruction, but rather to help the reader visualize how design differences (shown in a box) stand out clearly in TLA+.

choose to provide only an example here and make specifications available online [8].

From reading the original papers, GFS and Niobe seem very different systems, designed and optimized for quite different client semantics and workloads. However, as we were creating their specifications, it became clear to us that the systems had in fact a lot of mechanisms in common. Fundamentally, they both rely on a single master and a primary-secondary replication scheme. Consequently, we abstracted this structure into a common TLA+ module, which we extended in the Niobe and GFS specifications. This factorization turned out to be a powerful effect: the common module has 291 TLA+ lines, the modules specific to GFS-writes and Niobe are 189 lines and 287 lines, respectively, and the initial, unsplit Niobe specification was about the same size as the factorized one. In other words, the two systems' specifications have over half their TLA+ lines in common.

After factorization, the core differences between the two systems stood out clearly in TLA+. For example, our specifications make clear the distinction between write finalization in GFS and Niobe. Figure 2 illustrates this distinction in a side-by-side comparison of a part of the function specifying when writes are finalized. In GFS (left side), the primary finalizes a write after the write request to each of the replicas has either been acknowledged or has timed out. In Niobe (right side), the primary finalizes a write only after each replica has either acknowledged the write, or it has timed out and *has been successfully removed from the group*. This last condition represents the distinction and is signaled by a box in the figure.

By abstracting out the key aspects that differentiate real systems, specifications also help us understand the trade-offs that each system bargains for. As one example, from the above design distinction, we learn that while GFS can achieve better write latency, Niobe never leaves the replica set in an inconsistent state, even after a failed write. As another example, by allowing the client to read from any replica, GFS achieves better read performance for workloads where many simultaneous clients read the same data.

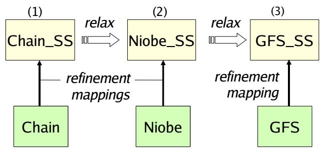


Figure 3. SimpleStore and refinement mapping for each system. We first construct and verify Chain's SimpleStore (Chain_SS), then relax Chain_SS to construct Niobe_SS, and further relax Niobe_SS to arrive at GFS_SS.

4. Understanding and Comparing Consistency

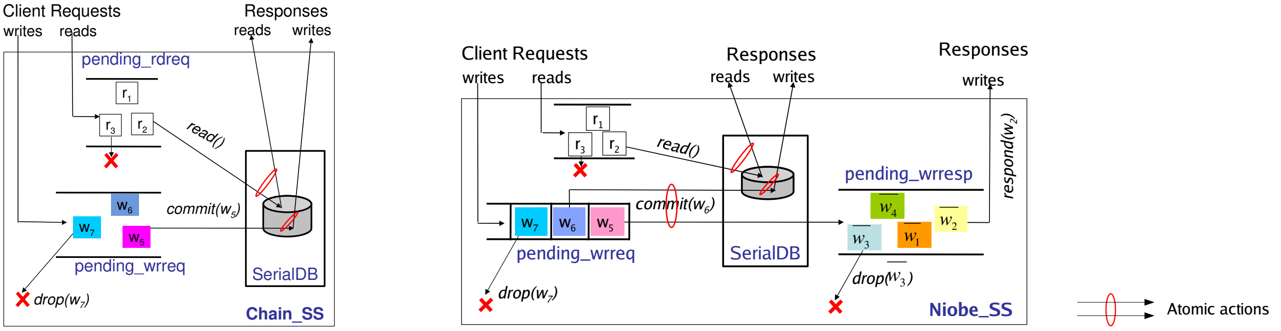
Currently, designers of fault-tolerant file systems typically rely only on reasoning to understand their systems' consistency. Reasoning about a full system can be complicated, faulty, lengthy, and inefficient (especially if the design is not yet finalized). In this section, we provide our experience with applying formal methods to understand and compare consistency properties of fault-tolerant file systems.

Our technique combines TLA+ specifications, refinement mappings, and model checking. In a nutshell, we reduce the systems to simplified, client-centric models (SimpleStores) and analyze and compare the consistency of those models instead. A system's SimpleStore *captures all client-visible behavior*, but abstracts out many lower-level details, hence making proofs of consistency properties easy. We specify SimpleStores formally in TLA+, produce refinement mappings from each system to the appropriate SimpleStore, and use model checking to validate the reduction. Then, by proving consistency properties about a system's SimpleStore, we infer that the system has those properties.

To enable comparison, we start by building a SimpleStore for the most strongly consistent system and then relax it to match the behavior of weaker systems. Figure 3 shows the order in which we reduce systems to their SimpleStores.

4.1. The Chain SimpleStore

Figure 4(a) shows the structure of the Chain SimpleStore (Chain_SS). It has two components: a reliable serial database (SerialDB) and two unreliable incoming channels (`pending_rdreq`, `pending_wrreq`). Clients push their read and write requests into `pending_rdreq` and `pending_wrreq`, respectively. SerialDB takes requests *one by one* from the channels, handles them, and responds to the client immediately. All SerialDB actions are atomic and persistent. Channels are unreliable: they can reorder or drop requests. If a



(a) Chain SimpleStore

(b) Niobe SimpleStore

Figure 4. Structure of Chain (a) and Niobe (b) SimpleStores. Sections 4.1 and 4.2 provide detailed descriptions.

Chain_SS variable	Mapping from Chain state to Chain_SS variable
pending_rdreq	Read requests at the tail
pending_wrreq	Union of all requests in the input channel of each live replica
SerialDB disk	Value of last write committed by tail

Table 2. Refinement mapping from Chain to Chain_SS (intuition). We show how to compute each variable in Chain_SS from the state in Chain.

channel drops a request, the request is never handled by SerialDB and hence is never responded to. To handle a read request (the *read()* action), SerialDB responds to the client with the current value of its disk. To handle a write request (the *commit()* action), SerialDB saves the write's value to its disk and sends a response to the client.

We produced a refinement mapping from Chain to Chain_SS (Table 2). We model-checked the refinement mapping using TLC, for a limited instance of the Chain system: three replicas, one object, and two data values. The check took two days and finished successfully, providing high confidence that indeed Chain implements Chain_SS.

Using Chain_SS, we can infer client-centric consistency properties of Chain, namely *linearizability*. Thanks to its simplicity, Chain_SS can be proved linearizable in about half a page [8]. Hence, Chain must also be linearizable.

Using formal methods, we were thus able to prove that Chain is linearizable for the common three-replica case. This is a powerful effect: using comfortable (and error-free) model checking of a simple model, we fortified the traditional error-prone reasoning about a full asynchronous protocol. Our method thus increases our trust in the system's behavior in the face of failures.

4.2. The Niobe SimpleStore

Intuitively, Niobe seemed to map well onto Chain_SS, so we attempted to model check a mapping between Niobe and Chain_SS. However, TLC revealed an example of Niobe behavior which is not mappable onto any Chain_SS behavior. The behavior, which requires 10 message exchanges, captures the case when an old primary's write succeeds and is responded to the client after a write of a newer primary. This behavior leads to a still linearizable history, however, it cannot be captured by Chain_SS. What we need is support for out-of-commit-order response delivery to clients.

Hence, we extended Chain_SS in Niobe_SS to add support for this behavior (Figure 4(b)). Two modifications are needed. First, in Niobe_SS, the input channel `pending_wrreq` is *ordered* and writes are committed in channel order. Second, to tolerate out-of-commit-order response delivery, SerialDB places responses to writes into a pending response channel (`pending_wrresp`). This channel can later drop a response or deliver it. Now, when SerialDB *commits* a write w, it moves all preceding writes in `pending_wrreq` to the `pending_wrresp` channel, commits w's value to disk, and ACKs w to the client. Later on, some of the moved writes might succeed (*respond()*), others might fail (*drop()*).

As with Chain, we model-checked a refinement mapping from Niobe to Niobe_SS for the same 3-replica instance of the system. The check finished successfully in 3 days, providing high confidence that Niobe implements Niobe_SS. Niobe_SS remains *linearizable*. The proof is slightly more involved than for Chain_SS, but certainly manageable [8] and significantly easier than for a full system.

4.3. The GFS SimpleStore

Even if we assume master reliability, GFS cannot be mapped onto either Chain or Niobe SimpleStores. We identified three counter-examples:

Ex. 1 **Non-atomic writes**. A GFS write can be split into multiple writes that go to different sets of replicas, and are thus serialized by different primaries.

Ex. 2 **Stale reads**. In GFS, a client can read from a stale replica, *i.e.*, one that is no longer part of the group and has missed some updates.

Ex. 3 **Read uncommitted**. Reads in GFS can go to any replica, so a client can read the value of an in-progress write. This can lead to non-sequentially consistent behaviors, like the one shown in Figure 5.

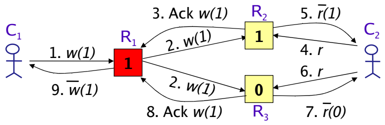


Figure 5. Counter-example for sequential consistency for GFS. R1 is the primary. The partially ordered sequence of messages (order numbers are shown) leads to a non-sequentially consistent history: $< w(1)$, r, $\bar{r}(1)$, r, $\bar{r}(0)$, $\bar{w}(1) >$ (barred operations represent responses).

The above examples are also counter-examples to sequential consistency and linearizability. However, the result that GFS is not linearizable is not surprising, nor does it enable comparison to Chain and Niobe's consistency models. What is more interesting is that by eliminating the first two counter-examples, we were able to map GFS onto a simple extension of Niobe_SS, with a well-understood consistency model. Hence, we make two assumptions:

A1 *Writes and reads never cross chunk boundaries*, and

A2 *Reads never go to stale replicas*.

Using the same technique as before, we reduced a GFS specification incorporating these assumptions to GFS_SS, which extends Niobe_SS as follows. A *read()* in GFS_SS returns either (1) the value of SerialDB, (2) a value from `pending_wrreq` or `pending_wrresp`, or (3) a dropped write.

GFS_SS offers standard regular register semantics [12], which are weaker than linearizability, but stronger than safe semantics. The proof is again very simple [8]. Thus, using formal methods, we were able to identify two assumptions that upgrade GFS' consistency guarantees to well-understood regular register semantics. This finding casts light on GFS' consistency model, which we found hard to grasp from the original paper.

5. Inspecting Alternative Designs

In the previous sections, we showed that formal methods can aid in understanding and comparing mechanisms and consistency properties of fault-tolerant file systems. Our experience indicates that formal methods can be a valuable tool during the design phase of a system, as well. They can be used by a designer to evaluate alternative designs comfortably. To inspect the effects of an alternative design on consistency, a system builder only modifies the specification and re-checks the refinement mapping, to verify whether the system still implements its SimpleStore.

Using our framework (consisting of TLA+ specifications, SimpleStores, and refinement mappings), we experimented with a simple design alternative for Niobe. As we have seen, one distinction between Niobe and GFS designs is that the former directs all reads and writes to the same primary, while the latter allows all replicas to answer a read request. GFS' read-any decision has an important impact on performance, since it increases GFS read throughput by distributing bandwidth across distinct replicas.

As a specific question of alternative design, what would happen to Niobe's consistency semantics if it were to employ the same read-any policy as GFS? Without extra mechanism, Niobe would no longer be linearizable, since it admits behaviors like the one shown for GFS in Figure 5. However, we were able to model-check a mapping from Niobe with read-any to GFS_SS, which shows that it must offer regular register semantics.

An interesting follow-up question is whether read-any Niobe can give up or simplify some of its mechanisms (*e.g.*, reconciliation at primary take-over) without losing the regular-register status. This question is a good example of a new question space whose exploration is enabled by our framework and an interesting point of future work.

6. Related Work

Formal modeling and methods have long been used to reason about software [3, 4] and hardware [11, 17]. We leverage these techniques and apply them to several fault-tolerant file systems. While formal methods have been widely used in hardware designs [11], builders of fault-tolerant file systems have still not adopted modeling and verification as a general practice. By sharing our experience, we hope to convince those builders of the utility and practicality of formally specifying their systems.

Our work is by no means the first with this goal. Many previous works report on the benefits of applying formal methods to various classes of systems, *e.g.*: caches [11], space shuttle software [6], on-line transaction processing systems [10], local and distributed file systems [18, 20, 21], and many others (a wealth of examples are presented in a survey [5]). Our work shows *how* and *why* to apply several formal methods to another important application domain: *enterprise fault-tolerant, replicated file systems*. From this body of previous works, the closest to ours are those presenting formal modeling case studies for local or distributed file systems (*e.g.*, Coda, AFS) [18, 20, 21]. Fault-tolerant file systems differ from these systems in that they include new types of complex mechanisms, *e.g.*, automatic reconfiguration and recovery. We believe that our study geared toward fault-tolerant file systems is likely to have impact in this specific domain in ways that previous studies may not.

Some works [2, 18] introduce new formal frameworks especially designed for modeling file systems. Because these specialized frameworks do not support model checking, proofs require manual effort. In contrast, we apply generic formalisms, which enable automatic verifications.

The technique of reducing complex systems to simple models to reason about consistency has been used before [11, 19]. In particular, the storage service model introduced in [19] is a valid abstraction, however the model's use of histories made it inappropriate for model-checking.

7. Conclusions and Lessons Learned

We have presented our experience with applying formal methods to analyze and compare three real-world fault-tolerant file systems. We now share four of the lessons we learned from our experiment.

First, moderately detailed TLA+ specifications of real systems are not as hard to produce as we had thought beforehand. For example, one student wrote a first workable specification for GFS in about two weeks. Clearly, the more in-depth the specification is, the more time it takes to write. But overall, we believe that writing a high-level specification by a system designer is a fairly easy task, yet a remarkably useful one for understanding the system.

Second, we found that the exercise of writing TLA+ specifications exposed similarities in seemingly dissimilar systems. This was the case for GFS and Niobe, where we factored out all common mechanisms into one abstraction. We believe that our common TLA+ specification can ease the building of specifications for other primary-secondary-master systems (*e.g.*, Boxwood [15]).

Third, formal specifications enable insightful semantic comparison, even between strongly and weakly consistent systems. By building client-centric models of the systems and comparing them, we were able to understand better how the systems behave and to reach several conclusions, *e.g.*:

1. Niobe and Chain perform similarly from a client perspective, implementing similar client-centric models.
2. GFS can be upgraded to regular register semantics via a clear set of assumptions.
3. GFS' design decision to read from any replica for performance heavily influences its consistency model. In particular, if Niobe were to adopt this design decision for performance, its consistency model would degrade from linearizability to regular-register.

Finally, we found that intuition can often be unreliable, and thus backing it up with formal verification is useful. For example, after verifying that Chain implemented Chain SimpleStore, we truly believed that the same model was right for Niobe, as well, without realizing that it missed one type of Niobe transition. It then took several iterations of the model to arrive at the right model (Niobe_SS).

Thus, our practical experience has shown that formal specifications and methods are useful tools for designing, analyzing and comparing fault-tolerant file systems. Through the use of such tools, systems designers can increase the trust in the behavior of these important infrastructure components in the presence of failures.

8. Acknowledgements

We thank Hank Levy and Chandu Thekkath for their valuable comments on the paper, and Idit Keidar, Dahlia Malkhi, and Yuan Yu for their ideas during our work. We also thank the anonymous reviewers for their valuable comments. This work was mostly done during an internship at Microsoft Research. Roxana Geambasu is supported in part by National Science Foundation Grant NSF-614975.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 1991.

[2] K. Bhargavan, M. Shapiro, and F. le Fessant. Modelling replication protocols with actions and constraints, 2003.

[3] M. Bickford and D. Guaspari. Formalizing the chain replication protocol. http://www.cs.cornell.edu/Info/Projects/NuPrl/FDLcontentAUXdocs/ChainRepl, 2006.

[4] D. Chkliaev, P. van der Stok, and J. Hooman. Formal modeling and analysis of atomic commitment protocols. In *Proc. of the Conference on Parallel and Distributed Systems*, 2000.

[5] E. Clarke and J. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4), 1996.

[6] J. Crow and B. D. Vito. Formalizing space shuttle software requirements: four case studies. *ACM Transactions on Software Engineering and Methodology*, 7(3), 1998.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, and W. V. P. Vosshall. Dynamo: Amazon's highly available key-value store. In *Proc. of ACM SOSP*, 2007.

[8] R. Geambasu, A. Birrell, and J. MacCormick. TLA+ Specifications and Proofs for Niobe, GFS, and Chain. http://cs.washington.edu/homes/roxana/fm/, 2007.

[9] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *Proc. of ACM SOSP*, 2003.

[10] I. Houston and S. King. CICS project report: Experiences and results from using Z. In *Proc. of Formal Development Methods*, 1991.

[11] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. Tuttle, and Y. Yu. Checking cache-coherence with TLA+. *Formal Methods in System Design*, 2003.

[12] L. Lamport. On interprocess communication. *Distributed Computing*, 1986.

[13] L. Lamport. *Specifying Systems*. Addison Wesley, 2003.

[14] L. Lamport, Y. Yu, and L. Zhang. TLA+ tools. research.microsoft.com/research/sv/TLA_Tools, 2007.

[15] J. MacCormick, N. Murphy, M. Najork, C. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. of OSDI*, 2004.

[16] J. MacCormick, C. Thekkath, M. Jager, K. Roomp, L. Zhou, and R. Peterson. Niobe: A practical replication protocol. *ACM Trans. Storage*, 2008.

[17] K. Shimizu and D. Dill. Using formal specifications for functional validation of hardware designs. *IEEE Des. Test*, 2002.

[18] M. Sivathanu, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and S. Jha. A logic of file systems. In *Proc. of the USENIX Conference on File and Storage Technologies*, 2005.

[19] R. van Renesse and F. Schneider. Chain replication for high throughput and availability. In *Proc. of OSDI*, 2004.

[20] J. Wing and M. Vaziri. A case study in model checking software systems. *Science of Computer Programming*, 1997.

[21] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proc. of OSDI*, 2004.