Experiences with Protocol Description

Pamela Zave
AT&T Laboratories—Research, Florham Park, New Jersey, USA
Email: pamela@research.att.com

***Abstract*—This paper presents three conclusions about the description of protocols, based on extensive experience: (1) Informal methods are inadequate for widely used protocols. (2) Lightweight formal methods are easy and useful. (3) Informal natural language cannot be trusted, but natural-language paraphrases of formal language can be trusted for certain purposes. None of these conclusions will be new or surprising to participants in this workshop. The purpose of this paper is to provide new, specific, and relevant evidence for these conclusions, in the hopes that researchers can use this evidence to justify their methods and influence the thinking of others.**

I. Introduction

In this paper, "description" refers to all the ways people communicate about protocols. "Specifications," which are expected to be formal and complete, are only one kind of description. "Documentation" is another kind of description, with the connotation of being informal, but yet complete. The discussion of a protocol in a research paper is a third kind of description that is seldom complete.

This paper presents three conclusions about the use of formal languages for describing protocols, based on extensive experience. None of these conclusions will be new or surprising to participants in this workshop. The purpose of this paper is to provide new, specific, and relevant evidence for these conclusions, in the hopes that researchers can use this evidence to justify their methods and influence the thinking of others.

The first conclusion (Section II) is that purely informal methods are inadequate for widely used protocols. The second conclusion (Section III) is that a formal description can be useful even if it falls far short of the "gold standard" of a complete formal specification that has been verified correct by means of an automatically checkable proof. These conclusions require no further introduction.

The third conclusion (Section IV) concerns the distinction between two kinds of natural language: the normal everyday kind, and language that is a paraphrase of a formal description. While the former can never be trusted as a protocol description, the latter, for many purposes, can.

Much of Section IV is a cautionary tale about the use of everyday natural language to describe a protocol in a research paper. This tale should have the effect of making researchers extremely careful if they attempt such a thing.

The positive side of Section IV, for researchers who do use formal descriptions for rigorous protocol engineering, is its endorsement of natural-language paraphrases. This is important because many researchers face the following dilemma. They wish to publish their results in venues where they will be noticed by people who design and use protocols. (This is more valuable than publishing in venues dedicated to formal methods, where the audience can appreciate the results but not use them.) The papers get poor reviews in practical venues, however, because the unfamiliar formalism makes them seem out-of-scope, not self-contained, and just too hard to read.

This seemingly insurmountable obstacle could be overcome if: (1) paper writers would include natural-language paraphrases of everything that they state formally, and (2) paper readers could be convinced that, for purposes of reading and understanding an applied-research paper, a natural-language paraphrase of a formal description is almost as good as the formal description itself. With this mutual understanding, an applied-research paper based on a formal protocol description could be self-contained, even for readers who skip the formalism entirely.

The fundamental purpose of Section IV is to make paraphrases credible and thereby promote this mutual understanding between writers and readers, which may be the only way to get the right people interested in technology that can help them.

II. Informal methods are inadequate for widely used protocols

A. The Session Initiation Protocol (SIP)

SIP is the dominant protocol for IP-based voice and multimedia applications, and has been standardized by the Internet Engineering Task Force (IETF). In keeping with the IETF philosophy of standardization based on "rough consensus and working code," it is described primarily in informal English.

The baseline description of SIP is 268 pages long [1].[1] Even when it was written, this document was not self-contained. Now that the protocol has been used extensively and extended frequently, its description (as of 2009) consists of 142 documents totaling tens of thousands of pages [3].

There is ample evidence that this description is not suitable for a protocol that is so widely used. The rest of this section contains some of this specific evidence.

The main reason for standardizing protocols is so that hardware and software produced by different equipment vendors will interoperate. If the standard is adequate, then all equipment compliant with the standard will be guaranteed to interoperate with all other such equipment. Yet for many years SIP interoperation was achieved only by twice-yearly "bake-offs." A bake-off was an event at which engineers from various vendors gathered in one room, with all their equipment, to test and re-program until their equipment interoperated on the test cases.

[1]This document replaced the original SIP standard [2].

People who use a protocol should be able to refer to its description to answer questions about the protocol. I am a member of a research group that develops tools and technology for SIP applications. Even for baseline SIP, we spend many hours trying to get the answers to simple questions such as, "Can a protocol endpoint in state *S* send a message of type *m*?" We search the document for clues, and argue their meanings like Biblical scholars. We rarely achieve certainty. Examination of SIP discussion forums indicates that we are not the only ones in this situation.

Regardless of how it is described, a protocol should be consistent. SIP experts worry that SIP's many extensions have introduced inconsistencies, and are well aware that inconsistencies could survive the existing documentation and standardization process. In fact, their fears are well founded. Even two of the earliest extensions, reliable provisional responses [4] and the *update* transaction [5], can cause violations of fundamental assumptions of the protocol [6].

A widely used protocol should not be unnecessarily complex. Each capability should be generalized as much as is reasonable and convenient, in preference to adding new capabilities that accomplish similar and overlapping goals.

The most important piece of SIP is the *invite* transaction, a three-way handshake allowing two endpoints to set up and negotiate the parameters of a set of media channels between them. The early extensions of reliable provisional responses [4] and the *update* transaction [5] serve the same function as the invite transaction, in different but overlapping circumstances. The cost of these extensions is considerable. They require five new message types, and create inconsistencies as reported above. A simplified model of the correct use of reliable provisional responses alone requires 11 states and 15 state transitions [6].

Yet with the addition of a single Boolean flag to the messages of the *invite* transaction, all of this additional complexity could be avoided, and all media-control functions could be performed with *invite* transactions alone [7]. As with detection of inconsistencies, informal description obscures the protocol and interferes with recognition of generalizations and other insights.

B. The Chord ring-maintenance protocol

In contrast to the wide industrial deployment of SIP, the ring-maintenance protocol for the Chord distributed hash table may be chiefly of interest to researchers. However, the interest is great—as of March 2011, according to Citeseer, the conference paper introducing Chord [8] is the fourth-most-referenced paper in computer science. Also, this paper just won the 2011 SIGCOMM Test-of-Time Award.

The protocol is described in the Chord papers in the form of very brief pseudocode [8], [9], [10]. Although this pseudocode appears to be more formal than natural language, it is incomplete in comparison to most formal descriptions in lacking types, preconditions, exceptions, and realistic definitions of distributed operations. Most importantly, it has no formal semantics and is not subject to any form of automated analysis.

This description is not sufficient, even for research purposes. This section contains two specific examples of its insufficiency.

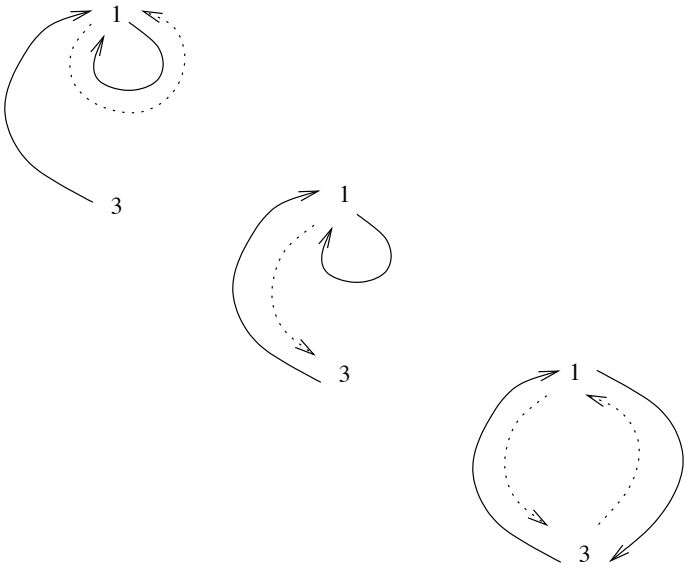

Fig. 1. How a Chord network with one node grows to two.

A study of the properties and correctness of the protocol reveals many problems with it, ranging from fundamental to easily fixable [11]. Reviewers of this work have said, ". . . many of the problems they identify are actually fixed by going through the process of implementation," and ". . . I am not sure that implementers of DHTs are not already aware of most of the issues." These statements may be true, but if so, none of the changes are documented in the public record.

More seriously, Chord is known for its fast and analyzable performance [10], [12]. In general, Chord performs well because its ring-maintenance operations are asynchronous and loosely coupled (see below), but this is exactly the characteristic that causes it to be incorrect and therefore unreliable [11]. It seems extremely likely that if implementers have altered the protocol to make it more correct, then they have also degraded its performance. It is misleading to claim a beneficial change without also acknowledging its harmful side-effects. Without better description of the protocol that is being implemented, a discussion of the trade-offs cannot take place.

Figure 1 illustrates some of the most basic concepts of the protocol. The solid arrows represent *successor* pointers, while the dotted arrows represent *predecessor* pointers. In an early state, a network has one node (here with identifier 1) whose successor and predecessor point to itself. In the left snapshot, a node 3 has joined the early network.

In the transition from the left to middle snapshot, node 3 *stabilizes* and *notifies*. The result in this case is simply to move node 1's predecessor pointer to 3.

In the transition from the middle to right snapshot, node 1 *stabilizes* and *notifies*. In the stabilization operation, it acquires its successor's predecessor, which is 3. Because 3 is a better successor to node 1 than 1, node 1 adopts 3 as its successor. In the notification operation, node 1 notifies its new successor 3 that node 1 is now its predecessor.

Note that a sequence like this is the *only* way that a Chord ring of size one can ever become a Chord ring of size two. Note also that no notion of atomicity applies. Each stabiliza-

tion is scheduled independently by its initiating node, at fixed intervals rather than upon the occurrence of some triggering event. Each stabilization is followed by a notification, with some unspecified delay.

Several papers have studied Chord implementations with model checking, among them [13]. This paper reports that the techniques applied were successful in discovering the following invariant: "A node's predecessor is itself if and only if its successor is itself." Given that the middle snapshot in Figure 1 shows a necessary stage of unknown duration, the protocol studied in [13] cannot be the documented Chord protocol. This leaves us wondering what changes were made, why they were made (unless they were due to accidental misunderstanding), and how the two protocols relate to each other.

III. Lightweight formal methods are easy and useful

Lightweight modeling is constructing abstract formal models of key concepts. These formal descriptions tend to be small and incomplete. *Lightweight analysis* is investigating the properties of formal models with tools based on exhaustive enumeration, so that analysis is "push-button," and yields results with little human effort. This section offers two examples of lightweight modeling languages and their analyzers.

A. Promela and Spin

We have built many models of SIP in the language Promela, with varying purposes and scopes [6], [14], [15], [16], and analyzed them with the model checker Spin [17].

The structure of our models is deliberately kept simple. Each protocol endpoint is a finite-state machine with a few major states. Minor state information is maintained in variables with Boolean or enumerated types. In each major state, there is a list of applicable guarded commands. Each command has a predicate (guard) true if a message might be received in that state or might be sent in that state. If a message might be received, the corresponding command defines the endpoint's immediate response to it (which might include sending other messages or updating local state). If a message might be sent, the corresponding command sends the message and updates local state.

There are assertions within the guarded commands, but only for one specific reason. They document the expected values of the minor state variables when a message is received in a major state.

Model checking verifies that these models do not deadlock, that they are complete in the sense of describing every message that can be received in every state, and that all parts of the model are necessary, *i.e.,* reachable. The techniques for performing this verification are simple and push-button [6]. There is no attempt to verify more abstract properties of the protocol, mostly because no such properties have been proposed.

The models are very incomplete, because almost all of the information in the many large message fields allowed by SIP is removed by abstraction. Many messages are represented only by their types. In messages that carry descriptions of media sessions [18], the complex session descriptions are mapped to the enumerated values *offer* and *answer*, indicating their role in offer/answer negotiation of the details of media sessions. We retain this information so that we can check consistency with respect to offer/answer negotiation.

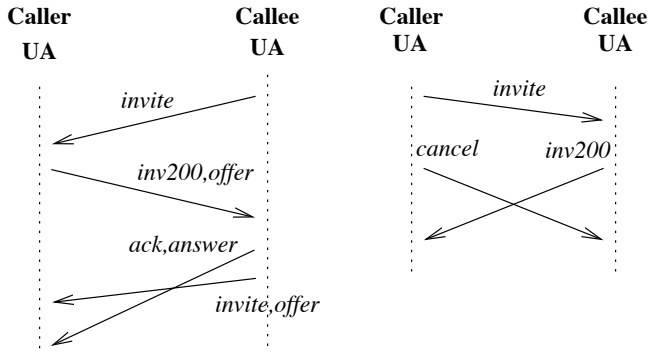

Fig. 2. Race conditions arise when messages going in the same or opposite directions cross in transit.

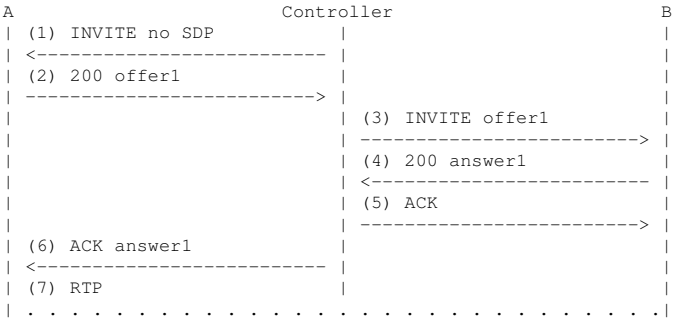

Fig. 3. Because the IETF standard for message-sequence charts shows instantaneous message delivery, race conditions appear to be impossible.

Because of their simplicity, the models are easy to read and maintain. Yet they are complete enough to provide our group with useful everyday documentation.[2] We have also used them to generate automatically a large number of test cases [16].

Analysis of the models also allows us to discover unknown properties of SIP. One instance is the inconsistencies mentioned in Section II-A. Another instance concerns race conditions.

In protocols, a race condition occurs when two messages cross in transit, either going the same direction (Figure 2 left) or different directions (Figure 2 right). Either kind of race can happen in SIP, especially when SIP messages are sent via UDP at the transport level.[3]

Many SIP documents use message-sequence charts to show particular common scenarios. These charts are rendered in ASCII by means of IETF macros, and look like Figure 3. Note that these charts represent message transmission as instantaneous, so that race conditions are impossible. Not surprisingly, SIP race conditions are not well documented, and their handling is incompletely standardized.

[2]Because of the uncertainties mentioned in Section II-A, we footnote our models with the sections in the standard where clues were found.

[3]SIP messages can use TCP or UDP.

A SIP document [19], many years in the making, records 7 possible race conditions within the scope of our earliest models [6]. Our models point to race conditions wherever they show that a message can be received when it is not expected or desired. By examining these places in our models, it is straightforward to find the same 7 race conditions and 42 others.[4]

B. Alloy and the Alloy Analyzer

The purpose of the Chord protocol is to maintain the ring structure of the network even though nodes can spontaneously join, fail, or leave silently. Because the protocol is best understood through the global network properties it holds invariant and the ideal properties it seeks to restore, it can be described well in Alloy [20]. The Alloy language is a powerful and flexible medium for expressing structural network properties.

The description of Chord operations in Alloy is slightly larger than the original pseudocode, primarily because the Alloy descriptions include extra information such as preconditions. Unlike the pseudocode, they can be checked with the Alloy Analyzer to get a complete characterization of their possible effects on the network state.

For example, Figure 4 shows how the operations as described in the pseudocode can allow the cycle to be broken [21]. The example applies to a network of any size, with the three snapshots showing only a portion of the network. In the left snapshot, the node with identifier 10 is joining. It has reached the same stage as node 3 in the middle snapshot of Figure 1.

In the middle snapshot, node 10 has failed or silently left the network. This is represented by the fact that 10 no longer has a successor pointer. In the transition from the middle snapshot to the right snapshot, node 6 has stabilized, causing it to change its successor pointer from 12 to 10. Also, node 12 has detected that 10 is no longer a member, and has flushed its predecessor pointer to 10.[5]

The right snapshot is a state from which a Chord network might not be able to recover, if it does not have the relevant redundant successor information (see Section IV) in place at the right time. Even if it can recover, some information will be temporarily inaccessible.

These problems can be avoided if node 6 checks, before discarding its previous successor 12, that the new successor 10 is still a member of the network. This is easily added to the stabilization operation as a precondition.

This counterexample to the correctness of Chord can be discovered simply by checking model instances in which stabilization occurs, to see if a cycle is always present in the final state of the operation. The problem is detectable by checking all instances of networks with 2 nodes, or any number of nodes greater than 2.

[4]Later documents show that ASCII art for race conditions has been developed. Otherwise [19] could not have been written.

[5]The flush is not necessary to break the cycle, as predecessor pointers do not define the cycle.

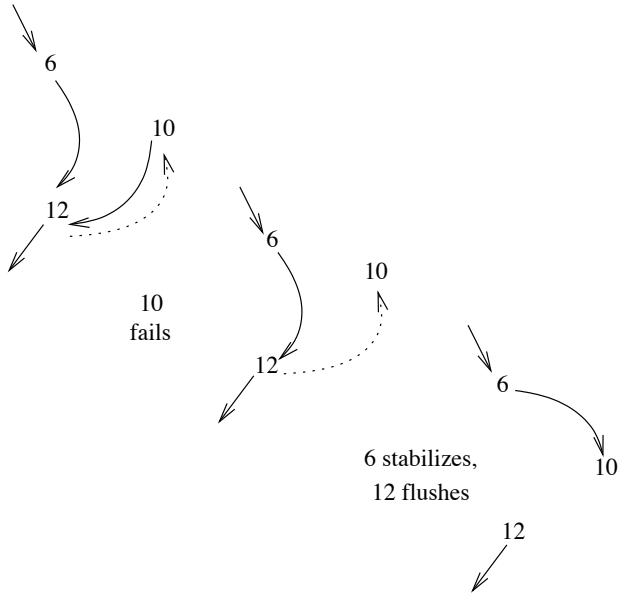

Fig. 4. How the cycle in a Chord network can be broken.

IV. Which natural language you can and cannot trust

Section I explained the need for natural-language description that people can trust for certain purposes, such as understanding the results in a research paper. The need arises because the readers who can benefit from the results are not necessarily the people who can read a particular formal language with ease.

There is a form of natural language that readers can trust, namely paraphrases of formal language. This section uses an example to show the difference between informal language and paraphrases of formal language.

Returning to Chord, Figure 5 shows a possible well-formed state of a Chord network. This figure shows successor pointers as solid arrows, as do Figures 1 and 4. Like node 10 in the middle of Figure 4, node 18 has no pointers because it is not a member of the network. It has failed or left the network—colloquially, it is *dead*.

To provide robustness in the face of node death, a member can also have a *successor2* pointing to a node other than its immediate successor. Second successors are shown as dashed arrows in the figure. Node 2 has just become a member, and does not have a second successor yet.

A desired invariant of Chord networks has been stated as follows [10]:

"If *v* is on the cycle, then *v.successor* is the first live cycle node following *v*."

This is ordinary or informal English, because it is not based on a comprehensive underlying formal model.

Consider first the term *cycle*, which is not rigorously defined, but is clearly meant to include only members, and to exclude appendages such as node 2. Most readers of [10] probably assume that the cycle is the subset of members that can reach themselves by following successor pointers, but with this definition the network in Figure 5 has no cycle at all.

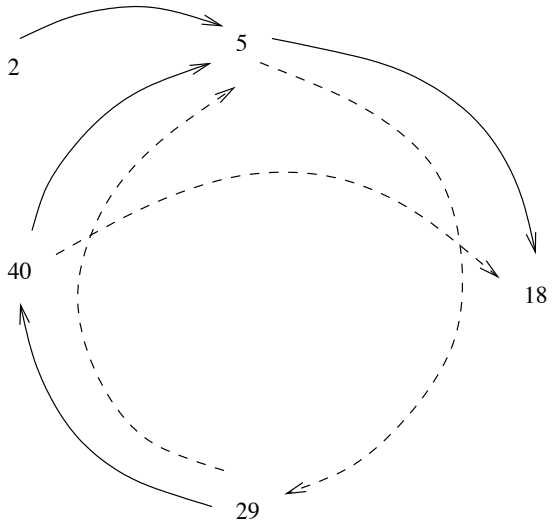

Fig. 5. A snapshot of a valid Chord network.

The Alloy model of Chord mentioned in Sections II-B and III-B defines a node's *best successor* as its first live successor, which may be its first or second successor.[6] The Alloy model defines a *cycle member* as a node that can reach itself by following best-successor pointers, expressed in an Alloy predicate as:

```
pred CycleMember [n: Node, t: Time] {
   n in n.(^(bestSucc.t))  }
```

The *cycle* is the set of all cycle members. This definition successfully captures our intuition, which is that the cycle in Figure 5 consists of nodes 5, 29, and 40.

With this plausible definition the network does not satisfy the stated invariant, however, because 5 is in the cycle, and its successor is neither live nor a cycle node.

Using the formal definitions of *cycle* and *best successor* as a foundation, we can improve the invariant by rewriting it as:

"If *v* is in the cycle, then *v*'s best successor is the first cycle node following *v*."

We now turn our attention to the original word "following." Following in what sense? If "following" means following pointers, then the statement is a tautology. If "following" means following in integer order on node identifiers, then the invariant is more meaningful but still wrong. The best successor of node 40 is 5, which does not follow it.

Chord identifiers come from a bounded set of natural numbers, and the relevant order on these identifiers wraps around from the highest number to zero. It is not useful to define "following" in such an order, however, because every identifier follows (and precedes) every other identifier. A more useful concept is that of "between," defined by the following Alloy predicate:

```
pred Between [n1,n2,n3: Node] {
   lt[n1,n3] => ( lt[n1,n2] && lt[n2,n3] )
           else ( lt[n1,n2] || lt[n2,n3] )
}
```

The built-in predicate *lt* is ordinary "less than" on natural numbers. The predicate *between* is true if and only if argument *n2* lies between arguments *n1* and *n3* in identifier order.

The many problems with the original Chord invariant are not surprising, because it is well known that ordinary informal natural language works poorly for rigorous description. Readers are right to distrust it when they find it in a paper.

To get an Alloy definition of the intended invariant property, we must reformulate it in terms of *between* (a ternary concept) rather than *following* (a binary concept). The result is:

```
pred OrderedCycle [t: Time] {
   let cycle =
      { n: Node | CycleMember[n,t] } |
   all disj n1, n2, n3: cycle |
      n2 = n1.bestSucc.t
   => ! Between[n1,n3,n2]
}
```

This predicate is based on the formal definitions of *best successor, cycle member* and *between*. If you don't understand the Alloy syntax, its English paraphrase is:

"If *n1* and *n2* are any two distinct cycle nodes, and *n2* is the best successor of *n1*, then no third cycle node falls between them in identifier order."

This paraphrase can be understood by any computer scientist. Yet all its terms are precisely defined, and its validity has been established by a great deal of automated analysis. It can be trusted to be unambiguous, consistent with intuition about ring networks, and suitable for the purpose of documenting and conveying some information about the properties of Chord. If it fails as a description for any reason, everyone has the formal model to fall back on.

V. Conclusion

Despite the maturity of formal description languages and formal methods for analyzing them, the description of real protocols is still overwhelmingly informal. The consequences of informal protocol description drag down industrial productivity and impede research progress, as the experience reported in this paper demonstrates.

One hears many prejudices from people who design and use network protocols. These prejudices help to explain and prolong the problem of inadequate protocol description. Among them, some people believe that formal descriptions:

- are difficult to write;
- must be complete (in the sense of covering every aspect of a protocol);
- must be completely verified as correct, yet cannot be verified manually because that is too difficult, and cannot be verified automatically because of computational complexity;
- are difficult to read;
- are useless unless every potential reader knows the formal language.

[6]Real successor lists are longer than 2, but two successors are sufficient for lightweight modeling. Other constraints in the model guarantee that each member node always has a best successor. The constraints are a deterministic version of Chord's probabilistic assumption that successor lists are long enough to include a live successor, with high probability.

This paper has presented evidence from experience with real protocols that none of these beliefs is necessarily valid. This evidence also shows how partial models, push-button analysis, and natural-language paraphrases make formal description of protocols surpassingly useful and practical.

REFERENCES

[1] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session Initiation Protocol," IETF Network Working Group Request for Comments 3261, 2002.

[2] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg, "SIP: Session Initiation Protocol," IETF Network Working Group Request for Comments 2543, 1999.

[3] J. Rosenberg, "A hitchhiker's guide to the Session Initiation Protocol (SIP)," IETF Network Working Group Request for Comments 5411, 2009.

[4] J. Rosenberg and H. Schulzrinne, "Reliability of provisional responses in Session Initiation Protocol (SIP)," IETF Network Working Group Request for Comments 3262, 2002.

[5] J. Rosenberg, "The Session Initiation Protocol (SIP) UPDATE method," IETF Network Working Group Request for Comments 3311, 2002.

[6] P. Zave, "Understanding SIP through model-checking," in *Proceedings of the Second International Conference on Principles, Systems and Applications of IP Telecommunications*. Springer-Verlag LNCS 5310, 2008, pp. 256–279.

[7] ——, "Audio feature interactions in voice-over-IP," in *Proceedings of the First International Conference on Principles, Systems and Applications of IP Telecommunications*. ACM SIGCOMM, 2007, pp. 67–78.

[8] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for Internet applications," in *Proceedings of SIGCOMM*. ACM, August 2001.

[9] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for Internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, February 2003.

[10] D. Liben-Nowell, H. Balakrishnan, and D. Karger, "Analysis of the evolution of peer-to-peer systems," in *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*. ACM, 2002, pp. 233–242.

[11] P. Zave, "Why the Chord ring-maintenance protocol is not correct (Extended Abstract)," AT&T Research, Tech. Rep., March 2011.

[12] S. Krishnamurthy, S. El-Ansary, E. Aurell, and S. Haridi, "A statistical theory of Chord under churn," in *Peer-to-Peer Systems IV*. Springer-Verlag LNCS 3640, 2005.

[13] C. Killian, J. A. Anderson, R. Jhala, and A. Vahdat, "Life, death, and the critical transition: Finding liveness bugs in systems code," in *Proceedings of the Fourth USENIX Symposium on Networked System Design and Implementation*, 2007, pp. 243–256.

[14] E. Cheung and P. Zave, "Generalized third-party call control in SIP networks," in *Proceedings of the Second International Conference on Principles, Systems and Applications of IP Telecommunications*. Springer-Verlag LNCS 5310, 2008, pp. 45–68.

[15] P. Zave, G. W. Bond, E. Cheung, and T. M. Smith, "Abstractions for programming SIP back-to-back user agents," in *Proceedings of the Third International Conference on Principles, Systems and Applications of IP Telecommunications*. ACM SIGCOMM, 2009.

[16] G. W. Bond, E. Cheung, T. M. Smith, and P. Zave, "Specification and evaluation of transparent behavior for SIP back-to-back user agents," in *Proceedings of the Fourth International Conference on Principles, Systems and Applications of IP Telecommunications*, 2010.

[17] G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[18] J. Rosenberg and H. Schulzrinne, "An offer/answer model with the Session Description Protocol," IETF Network Working Group Request for Comments 3264, 2002.

[19] M. Hasebe, J. Koshiko, Y. Suzuki, T. Yoshikawa, and P. Kyzivat, "Example call flows of race conditions in the Session Initiation Protocol (SIP)," IETF Network Working Group Request for Comments 5407, 2008.

[20] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[21] P. Zave, "Counterexamples to correctness of the Chord ring-maintenance protocol," AT&T Research, Tech. Rep., June 2010.