A Consistent History Link Connectivity Protocol *

Paul LeMahieu Jehoshua Bruck

California Institute of Technology
Mail Code: 136-93
Pasadena, CA 91125

Email: {lemahieu,bruck}@paradise.caltech.edu

Abstract

The RAIN (Reliable Array of Independent Nodes) project at Caltech is focusing on creating reliable distributed systems by leveraging commercially available personal computers and interconnect technologies. Fault-tolerance is introduced into the communication infrastructure by using multiple network interfaces per compute node.

When using multiple network connections per compute node, the question of how to monitor connectivity between nodes arises. We examine a connectivity protocol that guarantees that each side of a point-to-point connection sees the same history of activity over the communication channel. In other words, we maintain a consistent history *of the state of the channel. The history of channel-state is guaranteed to be identical at each endpoint within some bounded* slack.

Our main contributions are: (i) a simple, stable *protocol for monitoring connectivity that maintains a* consistent history *with* bounded slack, *and (ii) proofs that this protocol exhibits* correctness, bounded slack, *and* stability.

1. Introduction

Given the prevalence of powerful personal workstations connected over local area networks, it is only natural that people are exploring distributed computing over such systems. Whenever systems become distributed the issue of fault tolerance becomes an important consideration. In the context of the *RAIN* project (Redundant Arrays of Independent Nodes) [4] at Caltech (see Figure 1 for a photo), we've been looking into fault tolerance in several elements of the distributed system. One important aspect of this is the introduction of fault tolerance into the communication system by introducing redundant network elements and redundant network interfaces at each compute node. For example, a practical and inexpensive real-world system could be as simple as two Ethernet interfaces per machine and two Ethernet hubs. The work we have done is not specific to any networking technology, but we have been working primarily with Myrinet [3] networking elements as well as with Ethernet networks. The protocol described in this paper has been implemented as part of the Caltech RAIN system.

Figure 1. *RAIN* **system (ten nodes).**

An elementary piece of information about the system is whether there is *connectivity* between an interface on one machine and an interface on another. We describe here a modified *ping* protocol that guarantees that each side of the communication channel sees the same history. Each side is limited in how much it may lead or lag the other side of the channel, giving the protocol *bounded slack*. This notion of identical history can be useful in the development of applications using this connectivity information. For example, if an application takes error recovery action in the event of lost

*Supported in part by the NSF Young Investigator Award CCR-9457811, by the Sloan Research Fellowship, and by DARPA through an agreement with NASA/OSAT.

connectivity, it knows that both sides of the channel will see the exact same behavior on the channel over time and will thus take the same error recovery action. Such a guarantee may simplify the writing of applications using this connectivity information.

The protocol can be run at a high level, potentially as high as the application layer. At the same time, it can benefit greatly from low-level hardware hints about link conditions. A primary application for such a protocol is within a communication layer where its information can be used to mask or report connectivity problems between machines.

Although in some sense this is a consensus problem since the history of channel activity must be seen the same at both sides, it is not the general consensus problem people think of. We are only really interested in *eventual* consensus when the link is functioning. When the link has failed, we only care that the nodes see the failure. However, it is still useful to look at past work on consensus, such as Fischer, Lynch, and Paterson in [7], or in Lynch's book [10].

The connectivity problem has been addressed with different goals by Rodeheffer and Schroeder in the Autonet system [11, 12]. They were concerned with adaptive rates and skepticism in judging the quality of a link, whereas we are concerned with consistency in reporting the quality of a link. The connectivity problem has no doubt been considered in routing algorithms in the past, but we have seen no reference to keeping the history consistent at each side of a link.

Other than our own practical motivation for a consistent history of the channel state, Birman [1] gives general motivation for consistency in failure reporting for the purpose of improving reliability of distributed systems.

Our main contributions are: (i) a simple, *stable* protocol for monitoring connectivity that maintains a *consistent history* with *bounded slack*, and (ii) proofs that this protocol exhibits *correctness*, *bounded slack*, and *stability*.

The structure of the paper follows closely the contributions listed above. In Section 2 we define the problem. In Section 3 we explain the protocol. We omit the proofs for correctness, bounded slack, and stability in this shortened paper. A full version of the paper can be found in [9]. We finish in Section 5 with conclusions.

2. Problem Definition

We consider the following problem (Figure 2): given two nodes connected by some bidirectional communication channel, what is the state of the channel connecting them?

We desire a stable protocol that guarantees both sides see the same history of the channel up to some given *bounded slack*. Figure 3[b] shows what we desire in a consistent history between ends of a communication channel. We desire

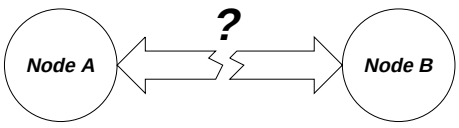

Figure 2. The problem: is the communication channel *Up* or *Down*?

that neither side be permitted to lag or lead the other by an arbitrary number of observed channel state transitions.

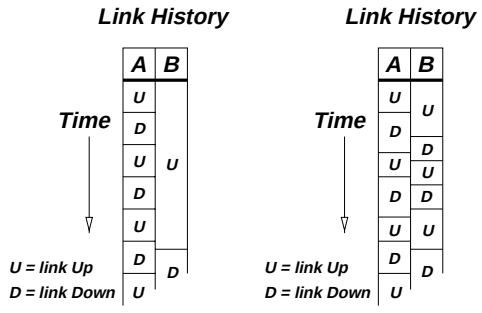

Figure 3. (a) Node A sees many more transitions than node B (b) Node A and B see tightly coupled views of the channel.

2.1. Defined Terminology

Below, we define some of the terminology used in this paper.

Nodes and Communication Channels. A distributed computing system is composed of a set of interconnected *nodes*. We are unconcerned with the underlying interconnect, but are interested in the existence of logical *communication channels* between a node and the other nodes in the system, on a point-to-point basis. The protocol runs over a pair of nodes connected by a communication channel.

Connectivity and Channel State. We consider two compute nodes *connected* only if bidirectional communication exists between them. Bidirectional communication is necessary for the implementation of reliable communication over unreliable channels. If a node finds itself connected to another node via a given channel, it considers that channel in the *Up* state. If a node finds itself not connected to another node via a given channel, it considers that channel in the *Down* state.

History. The sum of decisions made up about the state of a channel makes up that channel's *history*. Each of the

endpoints of the channel adds to its view of the channel-state history each time it decides the channel is *Up* or *Down*. A channel's history will be a series of channel states: *Up*, *Down*, *Up*, *Down*,.... Since the channel state is binary, a simple count of the number of state transitions suffices to fully describe the history.

Slack. As a node makes decision about the state of a channel, it may lead or lag the node on the opposite end of the channel. *Slack* is the amount a node may lead or lag its peer node. If t_a and t_b are the number of channel state transitions seen by node A and node B, respectively, and N is the slack parameter, than $|t_a - t_b| \leq N$ at all times.

Real Channel Event. A *real channel event* would be any spontaneously occurring information about the channel. The simplest would be "the channel appears to be up" or "the channel appears to be down." We'll look at the *timeout* (t_{out}) event that signifies that bi-directional communication has been lost. We'll also permit the *timein* (t_{in}) event, the complement to the timeout, that signifies that bi-directionally communication has been re-established. These are *real* events in the sense that they reflect information about channel activity beyond our control, not an event due to the protocol itself. The t_{in} event is not used in this shortened version of the paper since we only present the slack-2 case. The t_{in} event only plays a role in the protocol for slack greater than two.

Stability. A protocol determining the state of a communication channel should be *stable*. More precisely, for each channel event some bounded number of transitions (preferably one) should be seen by each endpoint.

Reliable Message Passing. The protocol we will describe requires *reliable message passing*. Since this is a protocol intended to work over unreliable channels, we are referring to software implemented reliability, such as a sliding window protocol. We require message passing that gives (eventual) guaranteed, in-order delivery.

2.2. Precise Problem Definition

We now present all the requirements of the protocol:

- *Correctness*: the protocol will eventually correctly reflect the true state of the channel. If the channel ceases to perform bi-directional communication (at least one side sees timeouts), both sides should eventually mark the channel as *Down*. If the channel resumes bi-directional communication, both sides should eventually mark the channel as *Up*.

- *Bounded Slack*: the protocol will ensure a maximum slack of N exists between the two sides. Neither side will be allowed to lag or lead the other by more than N transitions.

- *Stability*: each real channel event (i.e., timeout) will cause at most some bounded number of observable state transitions, preferably one, at each endpoint.

The system model is one in which nodes do not fail, but links intermittently fail. The links must be such that a sliding window protocol can function. See the discussion on data link protocols by Lynch in [10].

3. The Link Connectivity Protocol

This protocol uses *reliable message passing* to ensure that nodes on opposing ends of some faulty channel see the same state history of link failure and recovery. The reliable message passing can be implemented using a sliding window protocol, as mentioned above. At first it may seem odd to discuss monitoring the status of a link using reliable messages. However, it makes the description and proof of the protocol easier, preventing us from essentially re-proving sliding window protocols in a different form. For actual implementation, there is no reason to actually build the protocol on an existing reliable communication layer. The protocol can be easily implemented on top of ping messages (sent unreliably) with only a sequence number and acknowledge number as data (in other words, we can easily map reliable messaging on top of the ping messages).

The protocol consists of two parts:

- First, we have the sending and receiving of tokens using reliable messaging. Tokens are conserved, neither lost nor duplicated. Tokens are sent whenever a side sees an observable channel state transition. The observable channel state is whether the link is seen as *Up* or *Down*. The token-passing part of the protocol essentially *is* the protocol. Its job is to ensure that a consistent history is maintained.

- Second, we have the sending and receiving of ping messages using unreliable messaging. The sole purpose of the pings is to detect when the link can be considered *Up* or *Down*. This part of the protocol would not necessarily have to be implemented with pings, but could be done using other hints from the underlying system. For example, hardware could give instant feedback about its view of link status. For all the proofs to be valid, we must have that a t_{out} is generated when bi-directional communication has (probably) been lost, and a t_{in} is generated when bi-directional communication has (probably) been re-established.

The token-passing part of the protocol maintains the consistent history between the sides, and the pings give infor-

mation on the current channel state. The token-passing protocol can be seen as a filter that takes raw information about the channel and produces channel information guaranteed to be (eventually) consistent at both ends of the channel. The state machine of Figure 4 describes how each side of the protocol functions in the total system for $N = 2$.

Below we show the base case: the state machine for a slack of $N = 2$. Although not shown here, the full version of the paper [9] describes the similar general slack-N state machine. In Sections 4.1, 4.2, and 4.3 we discuss correctness, bounded slack, and stability for the protocol.

Now we describe the protocol for the base case where we have slack of $N = 2$. This is a significant case since it is the smallest value of slack for which any such protocol can work. Its description is hopefully somewhat simpler than the general case. A state machine as described in Figure 4 runs at each end of the link, at each node.

Intuitively, the state machine of Figure 4 shows the reaction to t_{out} events and T (token-receipt) events by the node at one end of the communication channel. The number of tokens currently held is t, and $2 - t$ is then the number of unacknowledged transitions the node has made. Note that $2 - t$ is at most 2, corresponding to the slack bound of two. The states can be described as follows:

1. *Up(t=2)*: The node is in the stable state. No unacknowledged transitions have been made by this node.

2. *Down(t=2)*: The node is catching-up with a transition seen by the other node that it itself did not see via a timeout. No unacknowledged transitions have been made by this node.

3. *Down(t=1)*: The node has seen a time-out and marked the channel as down. One unacknowledged transition has been made by this node (*Up* $\rightarrow$ *Down*).

4. *Up(t=1)*: The node has received acknowledgement (via a received token) for the *Up* $\rightarrow$ *Down* transition. One unacknowledged transition has been made by this node (*Down* $\rightarrow$ *Up*).

5. *Down(t=0)*: The node has seen a time-out and marked the channel as down, and is now blocked from further transitions by the bounded-slack constraint. Two unacknowledged transitions have been made by this node (*Down* $\rightarrow$ *Up* $\rightarrow$ *Down*).

Each state in Figure 4 is characterized by whether the node sees the channel as Up or $Down$, and how many tokens t are held by the node. The state transitions are labeled by the *action triggering the transition*, and the *action taken upon transition*. A trigger event is either a timeout t_{out} or receipt of a token T. The action taken is always whether a token is sent (1) or not (0). Note that a token T is sent whenever a transition for a *Up* state to a *Down* state, or from a *Down* state to a *Up* state is made.

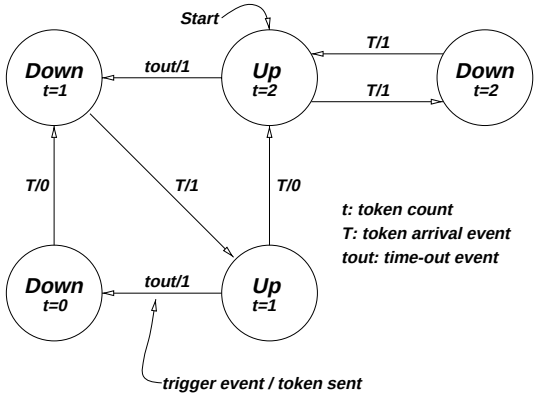

Figure 4. State machine for the connectivity protocol, slack $N = 2$.

4. Protocol Properties

The proofs of the following three theorems can be found in the full version of this paper [9].

4.1. Bounded Slack

This theorem is not actually specific to the protocol given above. A limited subset of the protocol (the token passing conditions) are sufficient to establish that slack is bounded.

Theorem 1 We take any protocol between two communicating nodes (A and B) with the following characteristics:

1. Each side starts with N tokens.
2. Tokens are never generated or destroyed
3. Tokens are sent exactly when a node decides the channel has made a change of state (*Up* $\rightarrow$ *Down* or *Down* $\rightarrow$ *Up*). In other words, tokens are sent for *observable state transitions* of the node.

Any protocol that meets these criteria will have a *bounded slack* property. If we call d_A and d_B the number of observable state transitions for node A and B, respectively, then

$$|d_A - d_B| \leq N$$

4.2. Stability

By *stability*, we mean that the protocol will exhibit finite response to a physical (timeout) event. We wouldn't want

a protocol that could repeatedly mark a channel as *Up* and *Down* in response to a single timeout. We require that the number of channel-state transitions is bounded by the number of physical timeouts in the system. More specifically, every timeout causes at most two $U \rightarrow D$ transitions: one at the side that sees the timeout explicitly, and possibly one at the peer node.

Theorem 2 For a system comprised of two nodes each running the state machine of Figure 4 and connected by a bi-directional communication channel, every *Up* $\rightarrow$ *Down* transition is directly caused by a timeout.

4.3. Correctness

By "correctness" we mean the simplest aspect of correct behavior: if the channel is down, both sides mark it as *Down*, and if the channel is up, both sides mark it as *Up*. The correctness requirement eliminates trivial protocols that always mark the channel as either *Up* or *Down*, and protocols that would allow themselves to get in a state where they must keep reporting the channel as *Up* simply to satisfy the bounded slack property, despite persistent timeouts.

Theorem 3 For a system comprised of two nodes each running the state machine of Figure 4 and connected by a bi-directional communication channel:

1. Each node will eventually mark the channel as *Up*, given that neither side sees timeouts (i.e., bi-directional communication exists).

2. Each node will eventually mark the channel as *Down*, given that both sides see timeouts (i.e., bi-directional communication does not exist).

5. Conclusions

Our main contributions are the creation of a a simple, stable protocol for monitoring connectivity that maintains a consistent history with bounded slack, and proofs that the protocol satisfies all these criteria. The protocol we've explained provides a mechanism for keeping the reporting of the channel state between two nodes consistent within a given slack. Consistency in the reporting of errors such as link connectivity problems can simplify the writing of applications acting on such error conditions, improving the overall reliability of a distributed system. A minimal slack of $N = 2$ is necessary for any protocol trying to guarantee consistency and still reflect the true state of the channel. A greater value for the slack permits the user of such a protocol tailor the degree to which the connectivity reporting truly reflects the current state of the channel at the expense of how tightly coupled the two nodes' histories must be.

There exists more work to be done in monitoring network errors in a distributed system. One straightforward extension is the reporting of connectivity of a clique of nodes. Such a protocol would tell whether a group of nodes is fully connected or not, and make the same guarantees of consistent history as the link protocol. Another useful protocol would be one that reports whether a given node is isolated from the group. Then, if any single node ever sees itself as isolated, all other group members also see it as isolated. A common trait to all these problems is that the decision being made is a *binary* one, for only then can one part of a system make a decision about a change in state *without communicating with other components.* This is a necessary condition so that decisions can be made independently, and eventual consistent history can still be guaranteed.

References

[1] K. P. Birman and B. B. Glade. Reliability through consistency. *IEEE Software*, 12(3):29–41, May 1995.

[2] K. P. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: A gigabit per second local area network. *IEEE-Micro*, 15(1):29–36, February 1995.

[4] V. Bohossian, C. Fan, P. LeMahieu, M. Riedel, L. Xu, and J. Bruck. *Computing in the RAIN: A Reliable Array of Independent Nodes*. Electronic technical report, http://www.paradise.caltech.edu/papers/etr029.ps.

[5] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

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

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

[8] G. J. Holzman. *Design and Validation of Computer Protocols*. Prentice Hall, New Jersey, 1991.

[9] P. S. LeMahieu and J. Bruck. *A Consistent History Link Connectivity Protocol*. Electronic technical report, http://www.paradise.caltech.edu/papers/etr023.ps.

[10] N. Lynch. *Distributed Algorithms*. Morgan Kaufman, New Jersey, 1996.

[11] T. L. Rodeheffer and M. D. Schroeder. Automatic reconfiguration in autonet. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, volume 25, pages 183–197. ACM, October 1991.

[12] T. L. Rodeheffer and M. D. Schroeder. A case study: Automatic reconfiguration in autonet. In S. Mullender, editor, *Distributed Systems*, chapter 11, pages 283–313. ACM Press, New York, second edition, 1993.