Connection Management Without Retaining Information[*]

Hagit Attiya[†] Shlomi Dolev[‡] Jennifer L. Welch[§]

November 22, 1995

.

Abstract

Managing a connection between two hosts in a network is an important service to provide in order to make the network useful for many applications. The two main subproblems are the management of serial incarnations of a connection and the transfer of messages within an incarnation. This paper investigates whether it is necessary for connection management protocols to retain state information across node crashes and between incarnations. The following results were obtained:

- When information is not retained across node crashes, incarnation management is not possible at all.
- When information is not retained between incarnations, incarnation management is possible if the network is FIFO and not possible if the network is non-FIFO.
- When information is not retained across node crashes, message transfer can be accomplished in networks that lose packets if the network is FIFO *and* the protocol is allowed a variable length grace period after a crash during which it need not deliver messages. However, message transfer cannot be accomplished if the network is non-FIFO or the grace period allowed is fixed.
- When information is not retained across node crashes, message transfer can be accomplished in networks that do not lose packets if the network is FIFO *or* the protocol need not be FIFO. Message transfer is not possible when the network is non-FIFO and the protocol must be FIFO.
- If the network has bounded capacity, then message transfer is possible without using stable storage. This indicates, somewhat surprisingly, that there is a data link initialization protocol that can withstand node crashes without stable storage.

[*]This research was supported by grant No. 92-0233 from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel, by TAMU Engineering Excellence funds, by NSF Presidential Young Investigator Award CCR-9396098, by Technion V.P.R. funds from B. and G. Greenberg Research Fund (Ottawa), and by the fund for the promotion of research at the Technion. An abridged version of this paper appeared as [5].

[†]Department of Computer Science, The Technion, Haifa 32000, Israel, hagit@cs.technion.ac.il.

[‡]Department of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva 84105, Israel, dolev@cs.bgu.ac.il.

[§]Department of Computer Science, Texas A&M University, College Station, TX 77843, welch@cs.tamu.edu.

1 Introduction

A major problem in the area of communication protocols is managing connections between two hosts across a wide-area network. Each connection between two specific hosts may have many incarnations over time. The task of managing connections between two hosts consists of *incarnation management*, managing serial incarnations of connections, and *message transfer*, transferring messages within an incarnation. Each host is associated with a node in the network; one node is the *sender* and the other is the *receiver*. The sender and receiver interact with the hosts and carry out a protocol to *establish* an incarnation of the connection, *transfer* messages from the sender's host to the receiver's host, and eventually *release* the connection. It is possible that a new incarnation of the connection will be requested later. Connection management forms the *transport layer* in the OSI network hierarchy and is built on top of the *network layer*. Transport protocols are at the heart of any wide-area communication network and form the basis for common protocols such as electronic mail, remote procedure call, talk, ftp, and rlogin.

A connection management protocol keeps information about the state of a connection in a *connection record* [34]. Most incarnation management protocols depend on having connection records available after a crash and before opening a new incarnation. But there are costs associated with maintaining connection records. A common technique to retain connection records across a crash is to keep them in stable storage, whose contents are unaffected by a crash. Even in systems which provide stable storage, updating it typically requires long access time and using it burdens the protocol. In addition, over a long period of time each host may have connections with many other hosts[1], while at any specific moment only a few of these connections have active incarnations. It can become prohibitively expensive to keep connection records indefinitely on each potential or past connection. (Cf. the protocol in [22] that uses synchronized clocks in order to avoid keeping information about inactive connections.) Consequently, we want to know whether it is necessary for the sender and receiver to retain connection records across crashes and between incarnations.

Since connection management is such a basic task, it is important to develop a rigorous basis for precise understanding of the issues concerning the appropriate network assumptions, the desired protocol specifications, and the interactions between the two. This theory should capture the important features of the problem, and still be simple enough that results can be proved formally. We make a step in this direction, focusing on the issue of retaining information for establishing and releasing connections, as well as for message transfer.

For our results, we assume a connectionless network layer, that is, a network layer that only supports primitives to send and receive packets, but does no error checking or flow control (e.g., IP in the TCP/IP protocol suite [14]). We assume that the network does not duplicate packets, and also that any corrupted packets are detected and discarded. In this environment, packets can be lost due to congestion, fragmentation, or a crash of an intermediate node. The network layer may or may not deliver packets in FIFO order. We consider only two extreme

[1]For example, the Internet, which runs TCP/IP, connects more than $200,000$ hosts [14].

types of ordering behavior by the network, perfectly FIFO and arbitrarily non-FIFO, and our results are stated as if these are the only possibilities. Dealing with intermediate ordering assumptions is left for further research.

Our formal treatment of the connection management problem models the ability to release a connection based on network misbehavior. Actual networks can "throw in the towel" and decide a connection should be terminated because of poor performance (cf. [29, pp. 377–378], and the usage of ICMP packets in TCP/IP [14, 32]). To capture this phenomenon, our formal model includes a special NProblem event by which the underlying network indicates that some severe packet transportation problem has occurred, e.g., all packets in transit were lost, or no packet is received during "too long" a time. A protocol that uses these indications can close the connection (and stop delivering messages) when some criterion on packet transportation is below its threshold. The use of NProblem makes it easier to design a protocol, since the nodes can "give up" and close a connection when it occurs. Not surprisingly, it makes proving impossibility results more difficult[2]. In order to make our results as strong as possible, we prove our impossibility results for protocols that rely on NProblem, while our protocols do not rely on NProblem to release connections.

We first consider incarnation management (IM) in the presence of crashes, without stable storage. Our results for this case are summarized in Fig. 1. In this case incarnations cannot be managed correctly, although the severity of the misbehavior exhibited depends on the specific network assumptions. In more detail, we show that incarnations cannot be managed correctly, even if the network is FIFO and does not lose packets. The error demonstrated is that actions of the two ends of the connection are not properly interleaved. In particular, one host may transfer messages assuming there is an active incarnation while the other does not participate in this incarnation. Since a "one-sided" connection could be considered non-hazardous, (and thus in Fig. 1 we use the term "weak NO" for it), we then show impossibility for FIFO networks that can lose packets. In this case, the error demonstrated is to fuse together into one incarnation at the receiver the delivery of messages that were transmitted at the sender on behalf of separate incarnations. (In Fig. 1 we use the term "strong NO" for such behavior since it demonstrates a severe violation of the requirements). This result assumes that the protocol is finite-state, a reasonable assumption satisfied by any real protocol.

We then consider incarnation management when information is not retained between incarnations and nodes do not crash. The results for this case are summarized in Fig. 2. They indicate that correct incarnation management is not possible for poorly behaved networks. Specifically, we show that correct incarnation management is not possible for networks which are not FIFO and can lose packets, but it is possible for networks either that are FIFO, even if packets can be lost, or that do not lose packets, even if delivery is non-FIFO.

We then turn to the subproblem of transferring messages within an incarnation. Ideally, every message transmitted by the sender's host is delivered to the receiver's host exactly once, and no messages are delivered that were not previously sent. In order to focus on the issues

[2]For example, one of the techniques in [17]—losing all the packets in transit and still requiring the protocol to perform a particular task—cannot be used (or requires more care).

network	loss	no loss
non-FIFO	strong NO (follows from below)	weak NO (follows from below)
FIFO	strong NO (Theorem 3.2)	weak NO (Theorem 3.1)

Figure 1: IM, Information not Retained Across Crashes

network	loss	no loss
non-FIFO	NO (Theorem 3.3)	YES (modify protocol below)
FIFO	YES (Theorem 3.4)	YES (follows from left)

Figure 2: IM, Information not Retained Between Incarnations

related to message transfer (MT), we assume a single infinitely long incarnation that remains open even if crashes occur.

We first consider the message transfer problem when the network can lose packets. One of the main factors in this case is whether the protocol is allowed a "grace period" after a crash, during which messages that are transmitted do not necessarily have to be delivered. Our results for this case are summarized in Fig. 3. They imply that message transfer is possible if the network is FIFO and the protocol is allowed a variable length grace period. However, message transfer is not possible if the network is non-FIFO or the grace period allowed the protocol is fixed. In more detail, we show that if the grace period must be fixed, then there is no protocol for this problem, even if the network is FIFO and the message delivery need not be. We further show that if the grace period is allowed to be variable, then there is a protocol for this problem that guarantees FIFO delivery of messages if the network is FIFO but can lose packets. If the network is not FIFO then there is no protocol for this problem.

Next we consider networks that do not lose packets. We show that in non-FIFO networks, achieving FIFO message delivery is impossible in the presence of node crashes, even if the liveness property to be satisfied is very weak. In the other three cases, there are simple protocols for message transfer; see Fig. 4.

The previous impossibility results for message transfer rely on the network having unbounded capacity to store up old packets. We also consider the case where the capacity (the number of packets that can be in transit at any given time) of the network is bounded. For this case, we have a message transfer protocol which can withstand crashes without stable storage. The protocol works even if the network can lose packets and is not FIFO but it is inefficient if the capacity is large. The protocol can be made more efficient for any capacity when the network is FIFO.

The statements of several of our impossibility results formalize beliefs held by practitioners. Known "folk" theorems in the practical community argue that unless some non-trivial timing assumptions are satisfied, nodes must keep connection records indefinitely and possess stable storage that can withstand crashes (cf. [29, Ch 6]). Roughly speaking, the argument is based on the "delayed duplicates" attack, in which old duplicate packets somehow collect in the network

protocol/network	non-FIFO	FIFO
fixed grace period	NO (follows from below and right)	NO (Theorem 4.1)
variable grace period	NO (Theorem 4.3)	YES (Theorem 4.2)

Figure 3: MT, Network Loses Packets

protocol/network	non-FIFO	FIFO
FIFO	NO (Corollary 4.5)	YES (simple protocol)
non-FIFO	YES (simple protocol)	YES (simple protocol)

Figure 4: MT, Network does not Lose Packets

and are then delivered to a node in such a way as to trick it into thinking a connection is open when it is not. It is true that (connectionless) network protocols do not detect and eliminate duplicates, but where can these duplicates come from and how can they be collected? For any reasonable type of hardware, duplicates are only caused by some protocol at some layer retransmitting a packet. The popular network protocol IP does not itself do retransmissions[3]; thus the only other possibility is that they come from the data link layer (which is concerned with reliable transmission between adjacent nodes over a physical link). But duplicates at the data link layer are easily avoided.[4] Furthermore, fiber optic networks of the near future may not even have a data link layer. Thus the assumption of arbitrarily delayed duplicates collecting in the network is too pessimistic in many reasonable situations.

Our results imply that even under a seemingly more benign assumption about the network (namely, no spontaneous duplicates), it is in many cases possible to collect duplicates so as to attack protocols for incarnation management and message transfer. This holds even for relaxed protocol requirements, including the existence of a grace period and the possibility of releasing a connection upon network misbehavior.

Our results that assume a non-FIFO network are clearly applicable to the environment of a connectionless network layer, where packets are routed independently. The results that assume a FIFO network layer are theoretically interesting since the lower bounds are made technically stronger. They also have practical relevance to a connection-oriented network layer, which manages connections between hosts, but at the network level, with specific control over routing decisions; in some cases, when a fixed route is used, packets arrive in FIFO order. They are also relevant to data link initialization procedures, which provide a reliable connection between nodes that are physically connected. The protocol for bounded capacity networks is particularly relevant to the data link layer, since the capacity of a physical link will generally be constant.

[3]Cf. [14], p. 98, where the specification of the protocol does not include retransmission, although implementations are not prohibited from retransmitting.

[4]The alternating bit protocol with stable storage to tolerate node crashes could be used [9]; the amount of stable storage needed at a node is just one bit for every adjacent link, which is feasible, unlike keeping a connection record for thousands of connections. Alternatively, our bounded capacity protocol, which does not rely on stable storage, can be used.

The other protocols serve more as counter-examples and show that specific assumptions used to prove an impossibility result cannot be relaxed.

Connection management has been studied intensively in the practical literature and many ingenious protocols have been suggested (e.g., [12, 13, 28, 29, 31, 34, 35]). All these protocols rely on some combination of timers, packet delay bounds, synchronized clocks and unique incarnation identifiers; it has been argued informally that some combination of these assumptions is necessary [35]. Our work complements these protocols by identifying precisely which assumptions on the system are necessary and sufficient, and exactly which requirements from the protocol it is impossible to achieve in certain cases. Other work that complements ours is the vast literature on verification of communication protocols, including, for example, [18, 20, 26]. These papers concentrate on verifying known protocols rather than investigating whether the assumptions they rely on are necessary.

There is some prior work on impossibility results for connection management. Our impossibility result for message transfer, when a fixed grace period is allowed, on a FIFO network that can lose packets improves the result in [17], which assumed the stronger progress property that starting *immediately* after the last crash, all messages transmitted must be delivered. Belsnes [11] studies how many packet exchanges are necessary in order to manage incarnations, under various requirements and assumptions. LeLann and LeGoff [21] show that a connection cannot be established by protocols of a particular form. Other theoretical studies of communication protocols have mostly concentrated on the data link layer [1, 3, 17, 19, 23, 25, 30, 33]. Most of this work concerns implementing protocols using *bounded size* packets, an issue we do not address. Our protocols for message transfer on bounded-capacity FIFO networks use an idea from self-stabilizing protocols for cleaning the system with a new label. (This idea was first employed in [15] and later used in [2, 8, 16].)

Our impossibility result for amnesic incarnation management protocols requires that the hosts discard all information after a connection is released. This condition does not allow hosts to maintain, e.g., a single global counter that is common to all connections a host might have. Following our work, and using a very similar framework, it was shown ([7, 6]) that in this case, a three-way handshake incarnation management protocol is possible, but there is no two-way handshake protocol unless some information is maintained for each possible connection.

The rest of this paper is organized as follows. In the next section, we describe our model. Section 3 concerns the problems caused by different incarnations of the same connection, while Section 4 includes the message transfer results. We conclude, in Section 5, with a discussion of our results and directions for future research.

2 Definitions

In this section we present our model of computation. Then we describe the architecture of the system, which consists of two hosts, two nodes (one per host), and a network connecting the two nodes. Our description of the problem is somewhat simplified since we ignore two-way traffic at the message level (one node only transmits messages and the other node only delivers messages). We then present a modular technique for proving impossibility in many situations, called pumping.

2.1 I/O Automata

In this subsection, we briefly describe the input-output automaton model [24], as simplified for our purposes. Each system component is modeled by an automaton. The automaton is a state machine whose state transitions are labeled with *actions*. If there is a transition from a state labeled with an action, then that action is *enabled* in that state. The actions of an automaton are classified as either *internal* or *external*. External actions model communication with the environment and are further classified as either *input* (providing stimuli from the environment) or *output* (generating information to the environment). Since the component has no control over when inputs occur, each input action is enabled in every state. Internal actions are private to the component, i.e., not visible to its environment.

An *execution* of an automaton is an alternating sequence of states and actions, beginning with an initial state, in which each action is enabled in the previous state and each state change correctly reflects the transition relation for the intervening action. An occurrence of an action in an execution is an *event*. An execution is *fair* if every output or internal action that is continuously enabled eventually occurs. Informally, an execution is fair if the automaton eventually gets to perform a pending output or internal action (and is not, say, swamped with inputs). We will require liveness properties only of fair executions. A (fair) *schedule* of an automaton is the sequence of events in a (fair) execution.

The system as a whole is also modeled by an automaton, the automaton resulting from the *composition* of the components. In order for the composition to be defined, each action must be shared by at most two automata, and then the action must be an input of one and an output of the other. The state set of the composition is the Cartesian product of the state sets of the component automata. There is a transition from state s' of the composition to state s labeled with action π if and only if (1) π is enabled in each component of s' that corresponds to an automaton with that action, and (2) each component in s correctly reflects the corresponding transition for π (or is unchanged if the corresponding automaton lacks π). Each action of the composition retains its previous classification as input, output, or internal, except that an action that is input to one component and output to another becomes internal.

We use the following notation: If α is a schedule of a composite automaton A and X is a component of A, then $\alpha|X$ denotes the restriction of α to actions of X.

2.2 System Architecture

The system consists of two nodes S (sender) and R (receiver), a host at each node, and a network connecting S and R. S, R and the network are explicitly modeled as I/O automata.

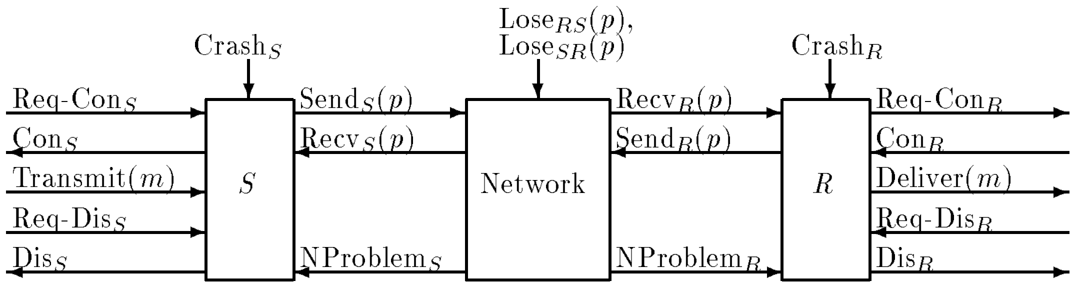

Figure 5: System architecture; hosts are not shown.

The hosts are not explicitly modeled. The system is the composition of S, R, and the network. (See Figure 5.)

2.2.1 Defining the Network

The actions of the network are (X is either S or R and Y is the other node):

- input Send$_X(p)$, X sends packet p,
- output Recv$_X(p)$, packet p is received by X,
- input Lose(p), packet p in transit is lost, and
- output NProblem$_X$, X is given an indication that there is a problem in the network.

A state of the network is two sets of packets, one set P_{SR} for packets from S to R and another set P_{RS} for packets from R to S. The effect of a Send$_S(p)$ action is to add p to the set P_{SR}. A Recv$_S(p)$ action is enabled whenever p is in the set P_{RS}; the effect of a Recv$_S(p)$ or Lose(p) action is to remove p from the set P_{RS} (if it is in the set). A packet is **in transit** (from R to S) if it is in P_{RS}. (Analogous definitions are obtained by reversing S and R.) We have just described a network that can lose packets—whether a packet is delivered or lost depends on which event happens first, Recv or Lose.

We will only require correctness of S and R for schedules of the system in which the network behaves "properly". We now define the various types of networks studied in this paper. (NA stands for network assumption.)

An automaton with the above actions is a **network** if every schedule satisfies NA1, NA2, and NA3 below:

NA1. There exists a one-to-one function $causep_{SR}$ from the set of Recv$_R$ events in the schedule to the set of Send$_S$ events in the schedule such that if $causep_{SR}(\text{Recv}_R(p)) = \text{Send}_S(p')$, then $p = p'$ (i.e., p and p' have the same contents) and Send$_S(p')$ precedes Recv$_R(p)$ (and analogously for packets from R to S). I.e., the network does not duplicate or corrupt packets, nor deliver spurious packets.

NA2. If the schedule is fair and if there is an infinite number of Send_S events, then there is an infinite number of Recv_R events (and analogously for packets from R to S). I.e., the network delivers infinitely many packets in each direction.[5]

NA3. For every prefix α of the schedule and $X \in \{S, R\}$:

1. if $\alpha = \alpha'$ $\mathrm{NProblem}_X$ for some α', then $P_X(\alpha'|N)$ is true.
2. if $P_X(\alpha|N)$ is true, then the next output event by the network involving X in the schedule is $\mathrm{NProblem}_X$.

The $\mathrm{NProblem}_S$ action of the network will be triggered depending on some predicate P_S on sequences of network actions that gives a necessary and sufficient condition for the event $\mathrm{NProblem}_S$ to occur. (For example, P_S might be that ten packets in a row are lost going from S to R.) Similarly, the predicate P_R controls the occurrence of $\mathrm{NProblem}_R$. Throughout this paper, we will only consider predicates P_X that are **loss**(2)**-only**, meaning that P_X is not true unless there have been at least two Lose events (in either direction) since the last $\mathrm{NProblem}_X$ event. Thus, the network ignores minor fluctuations which cause single packets to be lost, and triggers an NProblem only when things get worse.[6]

A network is **FIFO** if every schedule satisfies NA4 below; otherwise it is **non-FIFO**.

NA4. If $\mathit{causep}_{SR}(\mathrm{Recv}_R(p))$ precedes $\mathit{causep}_{SR}(\mathrm{Recv}_R(p'))$, then $\mathrm{Recv}_R(p)$ precedes $\mathrm{Recv}_R(p')$ (and analogously for packets from R to S). I.e., packets are delivered in the same order as they are sent.

A network is **non-losing** if every schedule satisfies NA5 below; otherwise it is **losing**.

NA5. If the schedule is fair, then causep_{SR} is onto (and analogously for packets from R to S). I.e., no packets are lost. (Note that NA5 implies NA2.)

2.2.2 Defining Incarnation Management

Ideally what happens is that the host at S requests a connection ($\mathrm{Req\text{-}Con}_S$), S communicates this to R and R checks with its host ($\mathrm{Req\text{-}Con}_R$). If R's host is agreeable (Con_R), R communicates this to S, who then tells its host (Con_S). Now the host at S transmits messages (Transmit) and S and R run a message transfer protocol so that they are delivered to R's host

[5] If this assumption does not hold, then there is a partition separating S and R, and clearly not much of interest can be achieved.

[6] Even if the network satisfies NA2, there is scope for misbehavior, such as losing a sequence of packets of any finite length. Such behavior might or might not trigger NProblem, depending on the choice of P_S and P_R.

(Deliver). The host at either S or R can unilaterally decide to end the connection (Req-Dis$_S$ or Req-Dis$_R$); once S and R have terminated the connection, the hosts are notified (Dis$_R$ and Dis$_S$). There is a third way that a connection can be released: if the network is experiencing problems, either S or R is notified (NProblem$_S$ and NProblem$_R$) and then they are free to decide whether to terminate any existing connection.

We now proceed more formally. The actions of S are

- input Req-Con$_S$, request from the host at S to establish a connection with R,
- output Con$_S$, indication to the host at S that the connection is established, at least as far as S is concerned,
- input Transmit(m), the host at S wants to transmit message m to the host at R,
- output Send$_S$(p), send packet p over the network to R,
- input Recv$_S$(p), receive packet p from R over the network,
- input Req-Dis$_S$, request from the host at S to disconnect the connection,
- output Dis$_S$, indication to the host at S that the connection is disconnected,
- input NProblem$_S$, indication that there is a problem in the network,
- input Crash$_S$ (optional), S crashes and recovers, and
- internal Step$_S$, S takes a local step.

The actions of R are

- output Req-Con$_R$, query to host at R whether it is willing to establish a connection with S (note that this is not symmetric, in that Req-Con$_R$ is an output from the host at R, while Req-Con$_S$ is an input to the host at S),
- input Con$_R$, indication from the host at R that it is willing to establish the connection (note that this is not symmetric, in that Con$_R$ is an input to the host at R, while Con$_S$ is an output from the host at S),
- output Deliver(m), deliver message m to the host at R,

as well as actions Send$_R$(p), Recv$_R$(p), Req-Dis$_R$, Dis$_R$, NProblem$_R$, Crash$_R$ (optional), and Step$_R$, which are completely analogous to those of S.

When we consider the case that nodes can crash, we use the Crash actions noted as optional above. To model the lack of stable storage, we require that there be a unique state rec_S of S that results from the Crash$_S$ action (and analogously for R). No matter what the state of S is immediately before a crash, it always returns to state rec_S immediately after the crash.

We will require a certain pattern of interactions between the hosts and the nodes. We now specify this pattern, called "well-formedness". Define the **host interface** of a schedule to be its restriction to the actions of the hosts, i.e., Req-Con, Con, Transmit, Deliver, Req-Dis, and Dis (but not Crash, NProblem, Send or Recv). A sequence of host events is defined to be **well-formed** if it can be extended to satisfy the following. Partition the sequence into sections, separated by Req-Con$_S$ events. Each section that begins with Req-Con$_S$, which is every section except possibly the first, is called an **incarnation**. If the first section is not an incarnation, then it must consist only of Dis events. Each incarnation must satisfy the following.

- The restriction of the incarnation to actions of the host at S has the form[7]

$$\text{Req-Con}_S\ [\text{Con}_S\ \text{Transmit}^*]\ [\text{Req-Dis}_S]\ \text{Dis}_S^+.$$

- The restriction of the incarnation to actions of the host at R has the form

$$\text{Dis}_R^*\ [\text{Req-Con}_R\ [\text{Con}_R\ \text{Deliver}^*]\ [\text{Req-Dis}_R]\ \text{Dis}_R^+].$$

- If Req-Con$_R$ occurs, it follows Req-Con$_S$, and if Con$_S$ occurs, it follows Con$_R$. So if we restrict to just Req-Con and Con actions, the sequence up to Con$_S$ is Req-Con$_S$ Req-Con$_R$ Con$_R$ Con$_S$. This is called the **real-time overlap** condition.

The disconnects in the first section would be due to Crashes that occur initially: below we will require a Crash to cause the incarnation to disconnect, but since crashed nodes cannot tell if there was an incarnation in progress or not, they can initiate a disconnect procedure even if there is no incarnation in progress. This is also the reason for the appearance of Dis$^+$ at the end of the conditions for S and R and the appearance of Dis$_R^*$ at the beginning of the condition for R. (Since an incarnation is defined to begin with Req-Con$_S$, there are no Dis$_S$ events at the beginning of the condition for S.)

If an incarnation includes Dis$_S$ and Dis$_R$, then it is **complete**. If an incarnation includes Con$_S$ but no Dis or Req-Dis, then it is **open**, and messages can be transferred. An incarnation in which one side wants to connect or disconnect but the other side has not yet done so is neither complete nor open.

We now define properties of schedules that reflect the assumptions on the hosts (HA*) and requirements on the protocol (IM*).

HA1. The hosts preserve well-formedness of the host interface. (I.e., if the schedule is well-formed so far, then any step by a host results in a schedule that is also well-formed).

IM1. The protocol preserves well-formedness of the host interface.

[7]Brackets mean optional, | gives alternatives, * means repeat 0 or more times, and + means repeat 1 or more times.

Next we put safety requirements on the message transfer function. IM2 states that every message delivered was previously transmitted within the same incarnation; IM3 is the FIFO property.

IM2. There is a one-to-one function *causem* from the set of Deliver events to the set of Transmit events in the schedule such that if $causem(\text{Deliver}(m)) = \text{Transmit}(m')$, then $m = m'$ (i.e., m and m' have the same contents) and Transmit(m') precedes Deliver(m) in the same incarnation. This is called the **message grouping condition**.

IM3. If $causem(\text{Deliver}(m))$ precedes $causem(\text{Deliver}(m'))$, then Deliver$(m)$ precedes Deliver(m').

Next we require that every Dis_S event can be attributed to a unique Req-Dis or Crash or NProblem event, and the same for every Dis_R. I.e., the protocol cannot simply decide to issue disconnects for no good reason. A natural notion would be that the event triggering a disconnect of an incarnation must occur in that incarnation. However, a crash might occur at R just before an incarnation starts at S and this crash should be allowed to close the incarnation. Thus our definition is looser. We only present impossibility results for the cases when crashes can occur; in our protocols, all of which are for no-crash cases, disconnects are only triggered by events in the same incarnation.

IM4. There is a one-to-one mapping from Dis_S events to previous Req-Dis, Crash, and NProblem events, and the same for Dis_R events. This is the **no unwarranted disconnects** condition.

(Our restriction to loss(2)-only predicates for the occurrence of NProblem implies that the loss of only one packet cannot cause a connection to be closed.)

We also require that any open incarnation *must* be closed in response to a crash. The motivation for this condition is that the most likely condition desired of the message transfer aspect of connection management is that the sequence of messages delivered in an incarnation should be a prefix of the sequence transmitted (and not have any messages skipped in the middle). In order to ensure this property in the face of crashes, an ongoing incarnation would have to disconnect in case crucial information for maintaining the prefix property had been lost.

IM5. If the schedule is fair, and if a Crash event occurs in an incarnation, then the incarnation is finite.

Since a node does not know whether an incarnation was in progress when it recovers from a crash, it must disconnect even if no incarnation is open.

Finally we have the liveness conditions, both for connecting and disconnecting (IM6) and for delivering messages (IM7). IM6 depends on R's host eventually responding to requests to open a connection, so we make a second host assumption, HA2.

HA2. If the schedule is fair, then every Req-Con$_R$ event is eventually followed by a Con$_R$ or Dis$_R$ event.

IM6. If the schedule is fair and consists of a finite number of incarnations, then the last incarnation is either complete or open.

Well-formedness alone guarantees that the schedule will consist of a finite or infinite sequence of complete incarnations, and if the number is finite, then there might be a final incomplete incarnation. IM6 says that neither the host nor the protocol can "stop playing" at inopportune times. It is possible for an infinite schedule to consist of a finite number of incarnations, where the last incarnation is infinitely long; it's important to allow for the possibility of an infinitely long incarnation in order to state liveness properties for the message transfer of the style of eventual delivery: e.g., in any incarnation with no Dis event, every message transmitted is delivered.

Any application-specific liveness properties on the type of message delivery to be supported should go here. For the lower bounds, we would like a weak condition that is still strong enough for our results to hold. We use IM7 below, but first we must make a definition. A schedule is **ping-pong** if the scheduling discipline for packet sends and receives in its associated execution is as follows. Every packet that is sent is received before the next packet is sent (thus no packet is lost). Furthermore, after each state in which both R and S have a Send action enabled, the next Send that occurs is at the node which did not perform the previous Send.

IM7. If the schedule is finite and is of the form α Req-Con$_S$ β, where β contains no Crash, no Req-Dis, no NProblem, at most one Lose, and at least one Transmit, then there exists an extension γ Deliver(m) of the schedule such that m is the last message transmitted in β, γ is ping-pong, and γ contains no Crash, Req-Dis, NProblem, or Transmit(m) events.

IM7 says that if we are in an open incarnation where everything is behaving beautifully, the latest pending Transmit must have a Deliver.

We now define the various types of incarnation management protocols. Let N be a certain type of network, i.e., every schedule of N satisfies NA1, NA2, NA3, and possibly NA4 and/or NA5.

The composition of S, R, and N forms an **incarnation management protocol** for that type of network if every schedule of the composition with no Crash events satisfies the following implication: If HA1 and HA2 are true, then IM1, IM2, IM4, IM6 and IM7 are true.

An incarnation management protocol is **FIFO** if IM3 is added to the list of IM properties in the above implication.

A (possibly FIFO) incarnation management protocol is **crash resilient** if Crash events are allowed to occur and IM5 is added to the list of IM properties in the above implication.

2.2.3 Defining Message Transfer

When focusing on the message transfer problem, the incarnation management model is simplified by considering only the actions Transmit, Deliver, Send, Recv, Step, Crash, and optionally Lose.

We now define the various types of message transfer protocols. Let N be a certain type of network, i.e., every schedule of N satisfies NA1, NA2, NA3, and possibly NA4 and/or NA5.

The composition of S, R, and N forms a **crash-resilient message transfer protocol** for that type of network if every schedule of the composition satisfies MT1:

MT1. There is a one-to-one function *causem* from the set of Deliver events in the schedule to the set of Transmit events in the schedule such that if *causem*(Deliver(m)) = Transmit(m'), then $m = m'$ (i.e., m and m' have the same contents) and Transmit(m') precedes Deliver(m). The one-to-one function requirement means that there are no duplications and no spurious messages; the $m = m'$ requirement means that there is no corruption of message contents. (This condition is the analog of IM2.)

A crash-resilient message transfer protocol is **FIFO** if every schedule satisfies MT2:

MT2. If *causem*(Deliver(m)) precedes *causem*(Deliver(m')), then Deliver(m) precedes Deliver(m'). This is the FIFO property (and is the analog of IM3).

Next we specify different liveness conditions achieved by a message transfer protocol, in the presence of node crashes.

A (possibly FIFO) crash resilient message transfer protocol is **exactly-once** if every schedule satisfies MT3:

MT3. If the schedule is fair and infinite but only has a finite number of Crash events, then every Transmit event following the final Crash event has a matching Deliver event.

Let t be a nonnegative integer. A (possibly FIFO) crash resilient message transfer protocol is **exactly-once with t-fixed grace period** if every schedule satisfies MT4:

MT4. If the schedule is fair, then every Transmit(m) event that is *not* one of the first t Transmit events after a Crash$_S$ or Crash$_R$ event has a matching Deliver(m) (w.r.t. *causem*). I.e., following a crash, up to t messages that are transmitted may fail to be delivered, but after that, the messages must be delivered. (Condition MT4 with $t = 0$ is the same as condition MT3.)

Let t be a function from system states to nonnegative integers. A (possibly FIFO) crash-resilient message transfer protocol is **exactly-once with $t()$-variable grace period** if every schedule satisfies MT5:

MT5. Suppose the schedule is fair. Choose an aribtrary Transmit(m) event. Let π be the latest preceding crash event and c be the system state immediately after π in the corresponding execution. (In case there is no such π, c is the initial system state.) If Transmit(m) is not one of the first $t(c)$ Transmit events after π, then it has a matching Deliver(m) (w.r.t. *causem*). This condition is similar to MT4, except that the number of messages after a crash that may fail to be delivered can depend on the system state. Intuitively, we allow the number of messages lost to depend on the behavior of the network in the execution leading to this state[8].

A (possibly FIFO) crash-resilient message transfer protocol is **at-most-once** if every schedule satisfies MT6 (a very weak condition):

MT6. If the schedule is fair and has only a finite number of Crash events but an infinite number of Transmit events, then there is an infinite number of Deliver events.

2.3 Pumping

Pumping is used in two results concerning incarnation management (Theorems 3.1 and 3.2) and two results concerning message transfer (Theorems 4.1 and 4.4). This technique is a slight generalization of the main technique used to prove the impossibility result of [17]. Pumping does not rely on non-FIFO or losing behavior of the network, although it does assume node crashes. We begin with a specific "reference" execution and modify it so that at the end of the modified execution a sequence of packets is in transit from S to R that will cause R to mimic its behavior in the original execution but without any further activity by S. We obtain the desired sequence of packets in transit by strictly alternating between S and R and each time replaying a little more of each one's history from the reference schedule and crashing the other one (thus "pumping up" the sequence of packets in transit).

Let $\mathcal{P}$ be any one of the crash-resilient protocols defined above, either for incarnation management or message transfer, for any one of the type of networks defined above. Let α be a schedule of $\mathcal{P}$. Thus α satisfies

- NA1, NA2, NA3, possibly NA4, and possibly NA5;
- HA1, HA2, IM1, IM2, IM4, IM5, IM6, IM7, and possibly IM3 if $\mathcal{P}$ is an incarnation management protocol;
- MT1, possibly MT2, and one of MT3, MT4, MT5, and MT6 if $\mathcal{P}$ is a message transfer protocol.

Assume further that α

[8]Variations similar in flavor to the requirement that a protocol is "bounded" were studied in [30, 33].

- is finite;
- is ping-pong;
- begins with Crash_S Crash_R;
- has no Lose events;
- when restricted to host actions begins with Dis_S and Dis_R in some order, if $\mathcal{P}$ is an incarnation management protocol.

Let $S_1, R_1, \ldots, S_k, R_k$ be the sequence of packets sent in α (which is also the sequence in which they are received, by the ping-pong property), divided into maximal length subsequences S_i and R_i such that all the packets in S_i are sent by S and all the packets in R_i are sent by R.

For any i, $1 \leq i \leq k$, let **sender**(α, i) be the restriction of α to actions of S from the beginning of α through the sending of all the packets in S_i. Similarly, for any i, $1 \leq i \leq k$, let **receiver**(α, i) be the restriction of α to actions of R from the beginning of α through the sending of all the packets in R_i. Let **pump**$(\alpha, 0)$ be the empty sequence. For any i, $1 \leq i \leq k$, let **pump**$(\alpha, i) =$ **pump**$(\alpha, i-1)$ sender(α, i) receiver(α, i).

For example, pump$(\alpha, 3)$ = sender$(\alpha, 1)$ receiver$(\alpha, 1)$ sender$(\alpha, 2)$ receiver$(\alpha, 2)$ sender$(\alpha, 3)$ receiver$(\alpha, 3)$. In words and ignoring host events: S crashes and sends all the packets in S_1. Then R crashes and as it receives the packets in S_1 (which are in transit), R sends all the packets in R_1. Then S crashes again and sends all the packets in S_1, and as it receives the packets in R_1 (which are in transit), S sends all the packets in S_2. Then R crashes again and as it receives the packets in S_1 and S_2 (which are in transit), R sends all the packets in R_1 and R_2. Then S crashes again and sends all the packets in S_1, and as it receives the packets in R_1 and R_2 (which are in transit), S sends S_2 and S_3. Then R crashes again and as it receives the packets in S_1, S_2 and S_3 (which are in transit), R sends all the packets in R_1, R_2 and R_3. The next lemma generalizes this example.

Lemma 2.1 *For any i, $1 \leq i \leq k$,*

(a) pump$(\alpha, i-1)$ sender(α, i) is a schedule satisfying the same network and host assumptions as α, and the packets $S_1, \ldots, S_i$ are in transit after it.

(b) pump(α, i) is a schedule satisfying the same network and host assumptions as α, and the packets $R_1, \ldots, R_i$ are in transit after it.

Proof: By induction on i. When $i = 1$, (a) and (b) are obvious. Suppose the inductive hypothesis is true for $i - 1 \geq 1$.

(a) To show that pump$(\alpha, i-1)$ sender(α, i) is a schedule, it is sufficient to show that all the packets received in sender(α, i), namely R_1 through R_{i-1}, are in transit after pump$(\alpha, i-1)$. This is true by the inductive hypothesis for (b) and the fact that NProblem never occurs (since there are no Lose events).

$S_1, \ldots, S_i$ are in transit at the end of pump$(\alpha, i-1)$ sender(α, i) because they are sent during sender(α, i), but none of them is received since there are no actions of R in sender(α, i).

Clearly none of the network assumptions satisfied by α is violated; HA1 is satisfied because of the Dis events at the beginning of α; HA2 is vacuously true because the schedule is not fair (packets are in transit at the end of α).

(b) Recall that pump(α, i) = pump$(\alpha, i-1)$ sender(α, i) receiver(α, i). To show that pump(α, i) is a schedule, it is sufficient to show that all the packets received in receiver(α, i), namely S_1 through S_i, are in transit after pump$(\alpha, i-1)$ sender(α, i). This is true by part (a).

$R_1, \ldots, R_i$ are in transit at the end of pump(α, i) because they are sent during receiver(α, i), but none of them is received since there are no actions of S in receiver(α, i).

Clearly none of the network assumptions satisfied by α are violated; HA1 is satisfied because of the Dis events at the beginning of α; HA2 is vacuously true because the schedule is not fair (packets are in transit). ■

Corollary 2.2 *pump(α, k) sender(α, k) satisfies the following.*

1. *It is a schedule.*
2. *S and R are in the same states after it as they are at the end of α.*
3. *No packets are lost in it.*
4. *$S_1, \ldots, S_k$ are in transit after it.*
5. *It satisfies the same network and host assumptions as α.*

Proof: 1. By Lemma 2.1, the packets $R_1, \ldots, R_k$ are in transit at the end of pump(α, k), and thus pump(α, k) sender(α, k) is a schedule.

2. Since pump(α, k) ends in receiver(α_k), which is $\alpha | R$, and begins with Crash$_R$, R is in the same state at the end of pump(α, k) as it is at the end of α. Since R takes no steps in sender(α, k), R is still in the same state at the end of pump(α, k) sender(α, k). By definition of sender(α, k), S is in the same state at the end of pump(α, k) sender(α, k) as it is at the end of α.

3. By construction, no packets are lost.

4. Since R takes no steps in pump(α, k) and no packets are lost, $S_1, \ldots S_k$ are in transit after pump(α, k) sender(α, k).

5. Clearly none of the network assumptions satisfied by α are violated; HA1 is satisfied because of the Dis events at the beginning of α; HA2 is vacuously true because the schedule is not fair (packets are in transit). ■

3 Incarnation Management

In this section we investigate the memory requirements for incarnation management. The requirements are to open and close connections correctly and to distinguish between messages belonging to different incarnations. We first study incarnation management when crashes are possible but nodes do not have stable storage. We then investigate the issue of incarnation management when crashes do not occur, but no state information is preserved between different incarnations of the same connection.

3.1 Crashes are Possible

We present two impossibility results for incarnation management when nodes can crash but have no stable storage. They both indicate that connection management is not possible; they differ in the particular bad behavior exhibited and the particular restrictions placed on the structure of the protocol.

The first result, Theorem 3.1, shows that even if the network is completely reliable, never losing packets and delivering them in FIFO order, there is no protocol. The contradiction is reached by showing that any proposed protocol can be forced into violating the real-time overlap condition. (Note that NProblem events never occur since no packets are lost.)

Theorem 3.1 *There is no crash-resilient incarnation management protocol for a FIFO non-losing network.*

Proof: Suppose in contradiction there is one. We claim there exists a finite ping-pong schedule α beginning with Crash_S Crash_R whose restriction to host actions begins with Dis events for S and R (in some order) and ends with

$$\text{Req-Con}_S \ \text{Req-Con}_R \ \text{Con}_R \ \text{Con}_S \ \text{Transmit}(m) \ \text{Deliver}(m) \ \text{Req-Dis}_S \ \text{Dis}_X \ \text{Dis}_Y$$

for some m where $X = S$ and $Y = R$ or vice versa. We now justify this claim. Since Crashes and Req-Con_S are inputs, they can happen at any time; IM5 implies that the Dis events follow the initial Crashes; Req-Con_S is a possible input; IM6 implies that Req-Con_R occurs; Con_R is a possible input; IM6 implies that Con_S occurs; $\text{Transmit}(m)$ is a possible input; IM7 implies that $\text{Deliver}(m)$ occurs; Req-Dis_S is a possible input; and IM6 implies that the Dis events occur. The schedule can be forced to be ping-pong by using the appropriate policy for delivering packets.

By Corollary 2.2 (the pumping technique), $\text{pump}(\alpha, k)$ $\text{sender}(\alpha, k)$ is a schedule that satisfies HA1, HA2 and NA1 through NA5. Thus, by the assumed correctness of the protocol, it should also satisfy IM1, preserving well-formedness. However, in the suffix $\text{sender}(\alpha, k)$, S opens a connection without R doing so, violating the real-time overlap condition of well-formedness. ■

While the above result shows a violation of the specification for incarnation management, one might argue that opening a "one-sided" connection is not so bad. In particular, since R does not establish a connection R is not receiving any wrong data. We remark that it is also possible to apply the pumping in reverse and trick R instead of S. In this case we will obtain a schedule whose suffix, restricted to host events, is Req-Con$_S$ Con$_S$ Transmit(m) Req-Dis$_S$ Dis$_S$ Req-Con$_R$ Con$_R$ Deliver(m) Dis$_R$. Note that this also violates the real-time overlap condition of well-formedness (IM1), but it does not violate the message grouping condition (IM2).

This reasoning motivates our second impossibility result, which shows that any finite-state protocol can be tricked into fusing together two separate connections at S into one at R. That is, messages from an earlier incarnation are delivered during a later incarnation, thus violating the message grouping property. This is a severe violation since different incarnations can be established on behalf of completely different application programs. These results rely on the network losing packets.

An incarnation management protocol is **finite-state** if both S and R are finite-state machines. We assume there are at least two distinct messages in the message alphabet.

Theorem 3.2 *There is no finite-state crash-resilient incarnation management protocol for a FIFO losing network.*

Proof: Assume there is such a protocol. Consider all ping-pong schedules with no Lose events of the form: S and R crash and disconnect and then establish an incarnation, Transmit and Deliver m_1, Transmit and Deliver m_2, ..., Transmit and Deliver m_k, and finally release the incarnation in response to Req-Dis$_S$. Restrict further the schedules so that each input action (namely, Req-Con$_S$, Con$_R$, Transmit, and Req-Dis$_S$) happens as soon as possible subject to the constraint that no packets are in transit (and well-formedness is preserved, of course). These schedules exist by IM5, IM6, IM7, and the fact that inputs are always enabled. The sequence $m_1, \ldots m_k$ of messages uniquely determines the schedule, which we denote by $sch(m_1, \ldots, m_k)$. Since the protocol is finite-state but there are infinitely many message sequences, there exist two sequences $m_1, \ldots, m_k$ and $m'_1, \ldots, m'_l$, such that $m_1, \ldots, m_{k-1} \neq m'_1, \ldots, m'_{l-1}$ and $m_k \neq m'_l$; furthermore, S and R are in the same states immediately before Transmit(m_k) in $sch(m_1, \ldots, m_k)$ as they are immediately before Transmit(m'_l) in $sch(m'_1, \ldots, m'_l)$. Call these states q_S and q_R.

Let α be a ping-pong schedule with no Lose events in which S and R crash, establish an incarnation, Transmit and Deliver m_1 through m_k, release the incarnation in response to Dis$_S$, crash, establish another incarnation, Transmit and Deliver m'_1 through m'_l, and release the incarnation. All Transmits occur when there are no packets in transit. Such a schedule exists by IM6, IM7, and the fact that inputs are always enabled.

Let S_1 be the sequence of packets sent by S in α between the first Crash$_S$ and the Transmit(m_k), S_2 between the Transmit(m_k) and the second Crash$_S$, S_3 between the second Crash$_S$ and the Transmit(m'_l), and S_4 between the Transmit(m'_l) and the end.

Intuitively, the proof proceeds by using the pumping technique to build another schedule such that all the packets sent by S in α are in transit at the end of the new schedule. We then lose the subsequence S_2, S_3 of packets from the middle. As a result, when the remaining sequence is delivered, R establishes a connection and delivers $m_1, \ldots m_{k-1}, m'_l$ in the same incarnation, violating IM2, the message grouping condition. The details follow.

Note that α', the result of concatenating the prefix of α up to just before Transmit(m_k) with the suffix of α starting with Transmit(m'_l) is a schedule, since the system state in α is the same at both points (namely, S is in q_S, R is in q_R, and no packets are in transit).

By Corollary 2.2 (the pumping technique), pump(α, k) sender(α, k) is a schedule.

Corollary 2.2 implies that at the end of pump(α, k) sender(α, k) the sequence S_1, S_2, S_3, S_4 of packets is in transit from S to R. Extend pump(α, k) sender(α, k) by appending a Lose event for every packet in S_2 and S_3. Then let NProblem$_S$ and/or NProblem$_R$ occur (if the predicates P_S and/or P_R are true). Then append Crash$_R$.

Finally, deliver the packets in S_1, S_4 to R, appropriately interleaved with output actions of R (i.e., Req-Con$_R$, Send$_R$) and a Con$_R$ response from R's host, so as to mimic R's behavior in α'. R has incorrectly combined the delivery of $m_1, \ldots m_{k-1}$ with the delivery of m'_l, data sent in distinct incarnations, violating IM2. ■

A similar impossibility result can be proved for protocols that handles every message as a "black-box". In such a protocol, the state of the sender, the receiver, and the network immediately following the transmission of a message can depend on any aspect of the execution so far, except for the contents of the messages transmitted so far. Thus, infinitely many states are allowed. Nevertheless, a technique similar to the one used in the proof of Theorem 3.2 can be used to fuse two incarnations into a single incarnation at R, during which a sequence of messages that was not transmitted in either of the original incarnations is delivered.

3.2 No Crashes But No Storage Between Incarnations

We now consider a well-behaved system in which nodes do not crash; thus there are no Crash actions. However, we do not allow the nodes to keep information in between incarnations of a connection. Formalizing this condition takes some work in order to prevent a protocol from "cheating" by storing incarnation information in packets that are in the network. We assume that S's state contains a queue of outgoing packets that are waiting to be introduced into the network, and the same for R. The remaining state components comprise the *accessible* state. The state transitions of S and R depend only the accessible state. We then assume that the following conditions are satisfied by the code for S and R:

1. Immediately after any Dis$_S$ event, S's accessible state has its initial value, a_S. S's accessible state does not change subsequently until Req-Con$_S$ occurs (although incoming packets can be handled and cause packets to be added to the outgoing packet queue).

2. Immediately after any Dis_R event, R's accessible state has its initial value, a_R. R's accessible state does not change subsequently until the receipt of a packet that was sent by S between the occurrences of Req-Con_S and Con_S for some incarnation (although incoming packets can be handled and cause packets to be added to the outgoing packet queue) [9].

That is, after a disconnect, S returns to its initial state (except possibly for its queue of outgoing packets) and remains in this state until a new incarnation is requested, and similarly for R. While S and R are between incarnations, they can generate packets (in response to packets received) but no information about the packets handled can be recorded. We call such protocols **amnesic**. We show that there is an amnesic protocol if and only if the network is FIFO.

The next theorem is proved by "stealing" packets. We take a specific "reference" execution and we replay successively longer prefixes of the reference execution. Each replay ends with the loss of a different packet. Because the protocol must tolerate the loss of a single packet, it must gracefully finish any pending task. Yet it is possible that the lost packet was not lost but only delayed in the non-FIFO network. Thus we can steal a single packet from each replay and keep it in the network. Then we deliver the packets we have collected to the receiver, tricking it into acting as if the reference execution is executed.

Theorem 3.3 *There is no amnesic incarnation management protocol for a non-FIFO losing network.*

Proof: Assume in contradiction there is such a protocol. We claim there exists a ping-pong schedule γ with no Lose events whose restriction to host actions is

$$\text{Req-Con}_S\ \text{Req-Con}_R\ \text{Con}_R\ \text{Con}_S\ \text{Transmit}(m)\ \text{Deliver}(m)$$

for some message m. We now justify this claim. Since Req-Con_S is an input, it can happen any time; IM6 implies that Req-Con_R occurs; Con_R is an input; IM6 implies that Con_S occurs; $\text{Transmit}(m)$ is an input; IM7 implies that $\text{Deliver}(m)$ occurs. The schedule can be forced to be ping-pong by using the appropriate policy for delivering packets.

Let $s_1, \ldots, s_k$ be the sequence of packets sent by S in γ. We inductively build a schedule $\gamma_1\delta_1\epsilon_1 \ldots \gamma_k\delta_k\epsilon_k$ as follows. Let γ_i be the prefix of γ through the sending of s_i.

Let ρ_0 be the empty schedule. Clearly after ρ_0, S and R are in their initial states and no packets are in transit.

Assume $\rho_{i-1} = \gamma_1\delta_1\epsilon_1 \ldots \gamma_{i-1}\delta_{i-1}\epsilon_{i-1}$ is a schedule after which S and R are in their initial states and $s_1, \ldots, s_{i-1}$ are in transit.

[9] The definition is asymmetric since in our definitions only the host at S can request to establish an incarnation.

Then $\rho_{i-1}\gamma_i$ is a schedule since every packet received in γ_i is sent in γ_i, S and R start γ_i in the same states that they end ρ_{i-1} in, the network is non-FIFO, and there are no Lose events (implying NProblem does not occur). Clearly, $s_1, \ldots, s_{i-1}$ are still in transit and s_i is also in transit.

We now define δ_i. If Transmit(m) is not in γ_i, then let δ_i be the empty string. Suppose Transmit(m) is in γ_i. Clearly γ_i Lose(s_i) is a schedule. By IM7, there exists an extension of γ_i Lose(s_i) with no Lose or Transmit(m) events in which Deliver(m) occurs. (Since P_S and P_R are loss(2)-only predicates, the loss of s_i alone will not trigger any NProblem.) Let δ_i be the shortest such extension. Since the network is not FIFO and the sender and receiver are in the same states after $\rho_{i-1}\gamma_i$ as they are after γ_i Lose(s_i), it follows that $\rho_{i-1}\gamma_i\delta_i$ is a schedule. Furthermore, s_i is not received in δ_i and only packets sent after ρ_{i-1} are received in δ_i. Thus $s_1, \ldots, s_i$ are still in transit.

We now define ϵ_i. By IM6, there exists an extension of $\rho_{i-1}\gamma_i\delta_i$ with no Lose event whose restriction to host actions is Req-Dis$_S$ Dis$_X$ Dis$_Y$ (where $X = S$ and $Y = R$ or vice versa). Let ϵ_i be the shortest such extension, followed by enough Send events to empty the outgoing packet queues at S and R.

Clearly $\rho_{i-1}\gamma_i\delta_i\epsilon_i$ is a schedule. After Dis$_S$ occurs in ϵ_i, S's accessible state remains a_S since there is no later Req-Con$_S$. After Dis$_R$ occurs in ϵ_i, R's accessible state remains a_R since γ_i is ping-pong and thus there are no undelivered packets sent by S between Req-Con$_S$ and Con$_S$. Thus at the end of ϵ_i, S and R are in their initial states. The packets $s_1, \ldots, s_i$ are still in transit.

Note that ρ_k is a schedule with no Lose events after which $s_1, \ldots, s_k$ are in transit. Thus ρ_k $(\gamma|R)$ is a schedule and it has an extension in which only R takes steps that ends with Deliver(m). However, in each $\gamma_i\delta_i\epsilon_i$ making up ρ_k there is a matching delivery in δ_i if the message m is transmitted in γ_i. Thus the last Deliver(m) has no matching Transmit, which is a violation of IM2, the message grouping condition. ■

We now show that there is an amnesic FIFO incarnation management protocol when the network is FIFO but can lose messages. Moreover this protocol uses bounded memory and only releases incarnations if explicitly requested to do so by either host; i.e., it doesn't depend on any NProblem events. The message transfer liveness requirement satisfied is that in an infinite incarnation of a fair schedule, the sequence of messages delivered is equal to the sequence transmitted.

Roughly speaking the incarnation management protocol synchronizes S and R on the incarnation by using the header 2 for packets that indicate the opening of a connection. Then, similarly to [10], the alternating bits 0 and 1 are used within an incarnation, both to transfer data items and to synchronize disconnections. More details follow.

When Req-Con$_S$ occurs, S repeatedly sends *open* with header 2 until receiving an acknowledgment. When R receives (*open*,2) for the "first" time (i.e., the most recent header received was not 2), it performs Req-Con$_R$; when Con$_R$ occurs, then R repeatedly sends (*ack*,2). When

S receives (ack,2) for the "first" time, it performs Con$_S$ and sets its header to 1. When Transmit(m) occurs, S adds m to its buffer (unless the buffer holds *dis*, indicating that disconnection is underway due to a Req-Dis$_R$).

To transfer the next message m in the buffer (which could be *dis*), S negates the current header and repeatedly sends m with the new header until receiving an acknowledgment; data messages are sent with alternating headers 0 and 1. When R receives a packet (m, h) for the "first" time, it performs Deliver(m) and repeatedly sends an acknowledgment with header h. When S receives an acknowledgment for the current header, it goes to the next message. If Req-Dis$_R$ occurs, R changes the contents of the acknowledgment packets it is sending to hold *dis*. If S gets a *dis* from R or Req-Dis$_S$ occurs, it sets its buffer to *dis*; note that a *dis* packet can also serve as an acknowledgment for the current header. When R receives *dis* for the "first" time, it performs Dis$_R$ and repeatedly sends an (ordinary) acknowledgment. When S receives an acknowledgment for the current *dis* message, it performs Dis$_S$.

Theorem 3.4 *There is an amnesic FIFO incarnation management protocol for FIFO losing networks that does not rely on NProblem.*

The protocol requires R to continue sending acknowledgments for the packets it receives even after Dis$_R$ and before the next incarnation begins. Note that such a behavior does not violate the amnesic property of the system because R does not need to have any information on the closed connection in order to acknowledge incoming packets[10]. It can be shown that such behavior is necessary in any protocol for this problem [4].

Note that the above protocol uses a small number of headers. Since the above protocol assumes that the network is FIFO, it also tolerates duplicate packets.

We now discuss the no-loss column in Fig. 2. The above protocol will obviously still work if the network does not lose any packets. It can also be modified to work even if the network is non-FIFO, as long as packets are not lost. The transformation consists of making sure that only one packet is ever in transit at a time, which can be done since packets are not lost and nodes do not crash [27]. In such cases, the network is essentially FIFO.

4 Message Transfer

We now consider the problem of recovering from crashes while guaranteeing some sort of reliable message transfer within a single infinitely long incarnation. We are interested in whether or not stable storage is required in the presence of crashes[11]. Thus throughout this section we assume nodes can crash.

[10]Tanenbaum [29, page 399] does not consider the possibility of this behavior, although no reason is given.

[11]Since the message transfer problem does not consider multiple incarnations, studying message transfer *without* node crashes would not provide any possibilities for failing to retain information according to our definition.

We first investigate message transfer when the capacity is unbounded, an assumption that approximates the situation in a wide-area network. Section 4.1 studies networks that can lose packets and Section 4.2 those that do not. We then consider, in Section 4.3, the situation when the capacity of the network is bounded.

4.1 Losing Networks

We now consider networks that can lose packets. We allow a message-transfer protocol to have a "grace period" after a crash during which messages need not be delivered. If the length of the grace period (in terms of the number of messages that don't have to be delivered) is fixed, then there is no protocol, even if the network is FIFO. However, if the length of the grace period can depend on the current state of the system (i.e., is "variable"), then there is a protocol if and only if the network is FIFO.

Theorem 4.1 *For any $t \geq 0$, there is no non-FIFO crash-resilient exactly-once message transfer protocol with t-fixed grace period for a FIFO losing network.*

Proof: Suppose in contradiction there is such a protocol. Let α' be a fair schedule that starts with Crash_S and Crash_R, and then includes $\text{Transmit}(m_1),\ldots,\text{Transmit}(m_{t+1})$ with no further crashes and no Lose events. By MT4 (the definition of the grace period), α' must include $\text{Deliver}(m_{t+1})$. Let α be the truncation of α' after the $\text{Deliver}(m_{t+1})$ event.

By Corollary 2.2 (the pumping technique), $\text{pump}(\alpha, k)$ $\text{sender}(\alpha, k)$ is a schedule. Corollary 2.2 also implies that at the end of $\text{pump}(\alpha, k)$ $\text{sender}(\alpha, k)$, S and R are in the same states as they are at the end of α. The suffix $\text{sender}(\alpha, k)$ contains a Crash_S event, no further crashes, and $t + 1$ Transmit events.

Let $\text{pump}(\alpha, k)$ $\text{sender}(\alpha, k)\alpha_1$ be a fair schedule such that α_1 has no Crash and no Transmit events and begins with Lose events for all packets in transit at the end of $\text{pump}(\alpha, k)$ $\text{sender}(\alpha, k)$. By MT4, there must be a $\text{Deliver}(m_{t+1})$ event in α_1 to match the last Transmit event in $\text{pump}(\alpha, k)$ $\text{sender}(\alpha, k)$. Since all the packets received in α_1 were sent in α_1 and S and R are in the same states at the end of α as at the end of $\text{pump}(\alpha, k)$ $\text{sender}(\alpha, k)$, $\alpha\alpha_1$ is also a schedule. But in this schedule, a duplicate message is delivered, namely m_{t+1}, violating MT1. ■

Note that the proof of Theorem 4.1 no longer holds when the notion of a grace period is modified as above. Since the number of messages not delivered during the grace period is not bounded a priori, the value of t that guarantees delivery after the initial crashes of S and R may not be big enough to guarantee delivery after the crashes in the pumping technique.

In fact, as we now show, for losing networks there is an exactly-once protocol with $t()$-variable grace period if the underlying network is FIFO. $t(c)$ is one greater than the number of packets in transit in c, the system state after a crash. The protocol is an adaptation of a self-stabilizing algorithm from [16].

The protocol works as follows. Initially (and after a crash), S has an empty buffer and sends *start* with header 0. When Transmit(m) occurs, S adds m to its buffer. To transfer the next message m in the buffer, S increments its header by 1 and repeatedly sends m with that header until receiving an acknowledgment for that header. The first time R receives a packet initially (or after a crash), R remembers the header and sends an acknowledgment for that header, but does not deliver the message. Subsequently, whenever R receives a packet with a certain header for the "first" time (i.e., the packet's header is different from the remembered header), R delivers the message in the packet as well as acknowledging it and remembers this header as the last one received.

Theorem 4.2 *There is a FIFO crash-resilient exactly-once message transfer protocol with $t()$-variable grace period for a losing FIFO network, where $t(c)-1$ is equal to the number of packets in transit in c.*

We now show that the assumption that the network provides FIFO delivery of packets is essential in order to obtain a protocol, even if the length of the grace period can depend on the state of the system after the crash and even if the protocol need not maintain FIFO delivery of messages. Note that this result relies on the assumption that the number of packets that can accumulate in the network is not bounded.

Theorem 4.3 *There is no non-FIFO crash-resilient exactly-once message transfer protocol with $t()$-variable grace period for a non-FIFO losing network, for any function t.*

Proof: To prove this theorem, we assume there is such a protocol and consider a schedule α where R and S crash and then there are many Transmit events (but no further crashes). After sufficiently many Transmits, there must be a Deliver(m) event that matches the last Transmit(m) event. We now use the stealing technique (cf. the proof of Theorem 3.3) to steal and hide packets from α, while still forcing Deliver(m) to occur. We then crash R and replay the packets we collected to cause R to erroneously deliver m again. The details follow.

Suppose in contradiction there is such a protocol. It has a finite ping-pong schedule Crash_S Crash_R γ that starts with Crash_S and Crash_R, resulting in system state c, then has $t(c)+1$ unique Transmit events (but no further crashes), and ends with a Deliver event to match the last Transmit event.

Let $s_1, \ldots, s_k$ be the sequence of packets sent by S in γ. We inductively build a schedule Crash_S Crash_R $\gamma_1\delta_1\epsilon_1 \ldots \gamma_k\delta_k\epsilon_k$ as follows. Let γ_i be the prefix of γ through the sending of s_i. (Note that γ_i has no Crash events in it.)

Assume $\rho_{i-1} = \text{Crash}_S\ \text{Crash}_R\ \gamma_1\delta_1\epsilon_1 \ldots \gamma_{i-1}\delta_{i-1}\epsilon_{i-1}$ is a schedule after which S and R are in their post-crash states rec_S and rec_R, and $s_1, \ldots s_{i-1}$ are in transit.

First note that $\rho_{i-1}\gamma_i$ is a schedule after which $s_1, \ldots, s_i$ are in transit, since all packets received in γ were sent in γ.

We now define δ_i. If Transmit(m) is not in γ_i, then let δ_i be the empty string. Suppose Transmit(m) is in γ_i. Clearly Crash_S Crash_R γ_i Lose(s_i) is a schedule. By MT5, there exists an extension of it with no Lose or Transmit(m) events in which Deliver(m) occurs.

Let δ_i be the shortest such extension. Since the network is not FIFO and the sender and receiver are in the same states after $\rho_{i-1}\gamma_i$ as they are after Crash_S Crash_R γ_i Lose(s_i), it follows that $\rho_{i-1}\gamma_i\delta_i$ is a schedule. Thus $s_1, \ldots, s_i$ are in transit after it.

Let ϵ_i be Crash_S Crash_R[12]. Since Crashes are inputs, $\rho_{i-1}\gamma_i\delta_i\epsilon_i$ is a schedule, and clearly after it S and R are in their post-crash states. Note that $s_1, \ldots, s_i$ are still in transit.

Since the network is non-FIFO,

$$\rho_k \ \text{Recv}_R(s_1) \ldots \text{Recv}_R(s_k) \ \rho_k \ (\gamma|R)$$

is a schedule, and it has an extension in which only R takes steps that ends with Deliver(m). However in each $\gamma_i\delta_i\epsilon_i$ making up ρ_k, there is a matching delivery in δ_i if the message m is transmitted in γ_i. Thus the last Deliver(m) has no matching Transmit, violating MT1. ■

4.2 Non-Losing, but Non-FIFO, Networks

We now assume that the network never loses packets but might deliver them arbitrarily out of order. Under this assumption, FIFO exactly-once message transfer is not possible. Note that Theorem 4.3 does not imply this result, since that theorem relied on having a losing network; yet this result does not imply Theorem 4.3, since this result only shows the impossibility of a *FIFO* protocol. In fact, as we discuss below, there does exist a non-FIFO protocol when the network does not lose packets. The impossibility can actually be shown for a very weak liveness condition.

Theorem 4.4 *There is no FIFO crash-resilient at-most-once message transfer protocol for a non-FIFO non-losing network.*

Proof: Suppose in contradiction that there is such a protocol. Let α be a finite ping-pong schedule that begins with Crash_S Crash_R and then contains a series of Transmit events, Transmit(m_1), ..., Transmit(m_j), ending with a Deliver(m_i) for some $i \leq j$. Let $s_1, \ldots, s_h$ be the sequence of packets sent from S to R in α. If R, starting in state rec_R (the state of R immediately following any Crash_R event), receives these packets, it will mimic its behavior in α and perform a Deliver(m).

[12] As in the proof of Theorem 3.3, ϵ_i is used to make R and S "forget" the execution so far. In Theorem 3.3, we caused R and S to close the connection, while here, we crash them instead.

By Corollary 2.2) (the pumping technique), pump(α, k) sender(α, k) is a schedule at the end of which S and R are in the same states as they are at the end of α and $s_1, \ldots, s_h$ are in transit.

We now show how to insert between pump(α, k) sender(α, k) and the later delivery of these packets the Transmit and Deliver for a distinct m'_k, causing a violation of the FIFO property.

Let β be a finite schedule that begins with Crash$_S$ Crash$_R$ and then contains a series of Transmit events, Transmit$(m'_1), \ldots,$ Transmit(m'_l), where $m_i \neq m'_h$ for all h, ending with a Deliver(m'_k) for some $k \leq l$.

Since β starts with S and R crashing and the network is non-FIFO, it follows that pump(α, k) sender$(\alpha, k)\beta$ is a schedule. Also, $s_1, \ldots, s_h$ are in transit after it. Hence,

$$\text{pump}(\alpha, k)\text{sender}(\alpha, k)\beta(\alpha|R)$$

is a schedule, and it has an extension in which only R takes steps that ends with Deliver(m_i). In this schedule, m_i is transmitted before m'_k, but delivered after m'_k, which contradicts MT2 (the FIFO property of the protocol). ■

The above theorem implies that, without stable storage, providing FIFO behavior for messages, when packets are not delivered in FIFO manner, is impossible in the presence of host crashes, even if the network does not lose packets and the protocol only has to deliver *some* messages.

Corollary 4.5 *There is no FIFO crash-resilient exactly-once message transfer protocol with* $t()$*-variable grace period for non-FIFO non-losing networks.*

The above corollary holds even if the grace period is allowed to be unbounded.

When the ordering properties of the protocol match those of the network, there is a simple protocol: Whenever Transmit(m) occurs, S sends a single packet with message m to R. Whenever R receives a packet from S, R performs the matching Deliver event. It is easy to see that this protocol guarantees (1) FIFO exactly-once message transfer on a FIFO non-losing network and (2) non-FIFO exactly-once message transfer on a non-FIFO non-losing network (as well as (3) FIFO at-most-once message transfer on a FIFO losing network and (4) non-FIFO at-most-once message transfer on a non-FIFO losing network).

4.3 Bounded Capacity

In this subsection we assume that at any time, at most *cap* packets are in transit from S to R (or vice versa), as will usually be the case when S and R are communicating directly over a physical link. (This constraint is modeled formally by restricting attention to schedules in which the Send and Lose events occur in such a way as to obey the capacity bound.)

Our interest in this assumption was motivated by the observation that several impossibility results for message transfer rely on the ability to collect an unbounded number of packets in the network, including some in this paper (e.g., 4.3) and one in [17]. In contrast, we now describe a crash-resilient exactly-once FIFO message transfer protocol for a FIFO losing network *with bounded capacity.* The capacity is used by the protocol as an upper bound on the number of packets from old transmissions to make sure that every transmitted message is delivered. A similar idea was used in [1]; however, their protocol dynamically maintains the upper bound on the number of packets in transit. Unfortunately, this upper bound grows exponentially during the execution of the protocol of [1].

Our protocol has the pleasing property that the number of packets sent depends on the capacity only after a crash has occurred at S. Note that $2cap$ is an upper bound on the total number of packets that can be in transit in both directions.

As Transmit events occur at S, S stores the messages to be transferred in a buffer and deals with them in FIFO order. When recovering from a crash (which of course will empty the buffer of pending messages), S repeatedly sends (initializing) packets $(start, 1)$ until it receives $(ack, 1)$; then, S repeatedly sends $(start, 2)$ until it receives $(ack, 2)$ and so on, until S repeatedly sends $(start, 2cap+2)$ and receives $(ack, 2cap+2)$. Only at this point, S starts to transfer messages (out of its pending buffer) as follows: To transfer a message m, S repeatedly sends packets $(m, 1)$ until it receives $(ack, 1)$; then, S repeatedly sends $(m, 2)$ until it receives $(ack, 2)$. Once $(ack, 2)$ is received, S is ready to send the next message m' in the same manner; that is, S sends $(m', 1)$ until it receives $(ack, 1)$; then, S sends $(m', 2)$ until it receives $(ack, 2)$.

R acknowledges any incoming packet with header h by sending the packet (ack, h). When R recovers from a crash, it first waits (while acknowledging incoming packets) for a non-*start* packet with header 1. Only at this point, R starts to deliver messages as follows: R delivers a message m to its host only upon receiving $(m, 1)$ and immediately afterwards $(m, 2)$.

Theorem 4.6 *There exists a FIFO crash-resilient exactly-once message transfer protocol for a losing FIFO network, if a bound on the capacity of the network is known.*

Proof: Consider the protocol just described.

We first prove the following invariant for any schedule of the protocol: Whenever S begins to transfer message m, there is no (old) packet with header 1 in transit. This is true initially. As long as no crashes occur, this continues to be true, since the network is FIFO.

A crash of R alone does not affect the invariant since R acknowledges every packet it receives with the header in that packet.

Now we show that a crash of S does not affect the invariant. Let c be the system state immediately after the last crash. Since the capacity is bounded, at most $2 \cdot cap$ distinct packet headers are in transit between S and R (in both directions) in c. If R has received a packet but not yet acknowledged it in state c, then at most one additional header could be sent by R. Thus, there are at most $2 \cdot cap+1$ distinct packet headers that are in the system but are not known to S in c. Let $\mathcal{D}$ be this set of $2\cdot$ *cap*$+1$ distinct headers and let $l(c)$ be one of $2\cdot$ *cap*$+2$ header values that is not in $\mathcal{D}$.

Before S starts sending non-*start* messages after the last Crash, S sends packets with all possible header values and waits for acknowledgments. In particular, S also repeatedly sends $(start, l(c))$ and waits for $(ack, l(c))$. S receives $(ack, l(c))$ only if R receives $(start, l(c))$ and sends $(ack, l(c))$. By the FIFO property, when S receives $(ack, l(c))$, all packets in transit have the header $l(c)$; moreover, from this point on, whenever S receives (ack, h) while sending packets with header h, all the packets in transit have the header h. This proves the correctness of the invariant.

Using the above we prove that the requirement for message transfer hold. Consider a schedule α of the protocol described above. We now define a function *causem* satisfying MT1 and MT2.

R performs Deliver(m) when it receives packet $(m, 2)$ immediately after receiving packet $(m, 1)$. S sends the packets $(m, 1)$ and $(m, 2)$ in response to a Transmit(m) event. Let *causem* map this Deliver to this Transmit.

Consider two events Deliver(m) and Deliver(m'), where Deliver(m) occurs before Deliver(m'). Then R receives at least one $(m, 1)$ packet, followed by at least one $(m, 2)$ packet, followed by at least one $(m', 1)$ packet, followed by at least one $(m', 2)$ packet. Since the network is FIFO, S sends $(m, 1)$ packet(s) and then $(m, 2)$ packet(s) in response to some Transmit(m) event, and subsequently S sends $(m', 1)$ packet(s) and then $(m, 2)$ packet(s) in response to some subsequent Transmit(m') event. Thus *causem* is one-to-one (MT1) and FIFO (MT2).

We now show MT3. Assume that α is fair, infinite, and has a finite number of Crash events. Because of NA2, S never gets stuck sending the same packet forever, i.e., eventually it gets an acknowledgment for the current packet. We will show that every Transmit(m) event occurring after the final Crash has a matching Deliver(m).

Suppose the last Crash is by R. Let Transmit(m) be an event occurring after the last Crash. Then, eventually after the last crash, S starts sending $(m, 1)$ and then $(m, 2)$. Eventually R receives $(m, 2)$ and performs Deliver(m).

Suppose the last Crash is by S. Let Transmit(m) be an event after the last crash. S repeatedly sends $(m, 1)$ in response, and then repeatedly sends $(m, 2)$ once it receives $(ack, 1)$. By the invariant proved above, this ack is a current ack sent by R, and not a "leftover." Thus R actually received one of the current $(m, 1)$ packets and when R receives the first $(m, 2)$ message afterwards, R will perform Deliver(m). ■

There is even a bounded capacity protocol for non-FIFO networks; however it is more inefficient in that at least *cap*$+1$ packets must be sent to transfer *each* message. (Cf. [4].)

5 Discussion

We have studied the necessity of retaining information between incarnations and across node crashes for two aspects of the connection management problem: incarnation management and message transfer. We proved that when state information is not saved between incarnations, the problem is solvable if and only if the network is FIFO. We also showed that incarnation management is not possible in the presence of crashes without stable storage. Furthermore, we showed that message transfer is possible in the presence of crashes without stable storage when packets can be lost if and only if the network is FIFO *and* the protocol is allowed a variable grace period after a crash during which it need not deliver messages. When packets are not lost, message transfer is possible if and only if either the network is FIFO or the protocol need not be. On the positive side, we have devised a data link initialization procedure that can withstand node crashes without stable storage, provided that the capacity of the physical link is bounded.

Our work forms another step in formalizing issues that arise at the transport layer in communication protocols. To the best of our knowledge, ours is the first theoretical study of this problem to incorporate the following practical features: indication of severe network misbehavior, grace period after a crash and bounded capacity of the physical links.

Many interesting issues remain to be studied, including flow control and buffering, analysis of the time-based techniques used in practical connection management protocols, and problems that arise in trying to do a clean disconnect [14, 29]. We hope that our model, definitions and techniques will be of help in continuing in these directions. In particular, we believe the NProblem action can be used to encapsulate timer-based mechanisms used to detect severe errors, by using an appropriate choice of predicates.

Another interesting aspect is to explore quantitative considerations such as the number of packets that have to be sent in the cases where incarnation management or message transfer is possible. For example, it would be interesting to know whether $\Omega(cap)$ packets are required in order to clear the connection after a crash, when the network is FIFO and the capacity, cap, is bounded.

Acknowledgments: We thank Reuven Cohen, Alan Fekete, Michael Kate, Steve Liu and Rinat Rappoport for helpful discussions and comments on earlier versions of this paper. The anonymous referees' many suggestions helped improve the paper as well.

References

[1] Y. Afek, H. Attiya, A. Fekete, M. Fischer, N. Lynch, Y. Mansour, D.-W. Wang and L. Zuck, *Reliable Communication over Unreliable Channels,* Technical Report YALEU/DCS/TR-853, Department of Computer Science, Yale University, October 1992. To appear in *Journal of the ACM.*

[2] Y. Afek and G. M. Brown, "Self-Stabilization over Unreliable Communication Media," *Distributed Computing*, Vol. 7, pp. 27-34, 1993.

[3] A. Aho, A. Wyner, M. Yannakakis and J. Ullman, "Bounds on the Size and Transmission Rate of Communication Protocols," *Computers and Mathematics with Applications,* Vol. 8, No. 3, pp. 205–214, 1982.

[4] H. Attiya, S. Dolev and J. L. Welch, "Memory Requirements for Connection Management," LPCR Report #9316, Technion, Israel Institute of Technology, June 1993.

[5] H. Attiya, S. Dolev and J. L. Welch, "Connection Management Without Retaining Information," *28th Hawaii International Conference on System Science,* Vol. II, pp. 622-631, 1995.

[6] H. Attiya, J. Kleinberg and N. Lynch, "Trade-Offs Between Message Delivery and Quiesce Times in Connection Management Protocols," to appear in *3rd Israel Symposium on Theory of Computing and Systems,* January 1995.

[7] H. Attiya and R. Rappoport, "The Level of Handshake Required for Establishing a Connection," *the 8th International Workshop on Distributed Algorithms,* pp. 179–193. Lecture Notes in Computer Science #857, Springer-Verlag.

[8] B. Awerbuch, B. Patt-Shamir and G. Varghese, "Self-Stabilization by Local Checking and Correction," *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, pp. 268-277, October 1991.

[9] A. Baratz and A. Segall, "Reliable Link Initialization Procedures," *IEEE Transactions on Communication*, Vol. COM-36, No. 2, pp. 144–152, February 1988.

[10] K. Bartlett, R. Scantlebury, and P. Wilkinson. "A Note on Reliable Full Transmission over Half-Duplex Links," *Communications of the ACM,* 12(5):260-261.

[11] D. Belsnes, "Single-Message Communication," *IEEE Transactions on Communication,* Vol. T-COM-24, No. 2, pp. 190–194, February 1976.

[12] E. W. Biersack and D. Feldmeier, "A Timer-Based Connection Management Protocol with Synchronized Clocks and its Verification," to appear in *Computer Networks and ISDN systems.*

[13] D. Cheriton, "Sirpent(TM): A High Performance Internetworking Approach," *Proc. ACM SIGCOMM,* pp. 158–169, 1989.

[14] D. Comer, *Internetworking with TCP/IP, Volume I: Principles, Protocols and Architecture,* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[15] E. W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Communications of the ACM,* Vol. 17, No. 11, pp. 643-644, 1974.

[16] S. Dolev, A. Israeli and S. Moran, "Resource Bounds for Self Stabilizing Message Driven Protocols," *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing,* pp. 281-293, August 1991.

[17] A. Fekete, N. Lynch, Y. Mansour and J. Spinelli, "The Impossibility of Implementing Reliable Communication in the Face of Crashes," *Journal of the ACM*, Vol. 40, No. 5 (November 1993), pp. 1087–1107.

[18] J. F. Kurose and Y. Yemini, "The Specification and Verification of a Connection Establishment Protocol using Temporal Logic," in *Protocol Specification, Testing and Verification II* (C. A. Sunshine, Ed), North-Holland, New-York, 1982.

[19] R. Ladner, A. LaMarca and E. Tempero, "Counting Protocols for Reliable End-to-End Transmission," Technical Report 92-10-06, Department of Computer Science, University of Washington, 1992.

[20] S. S. Lam and A. U. Shankar, "Protocol Verification via Projections," *IEEE Trans. on Software Engineering,* Vol. 10, pp. 325–342, July 1984.

[21] G. LeLann and H. LeGoff, "Verification and Evaluation of Communication Protocols," *Computer Networks,* Vol. 2, pp. 50–69, 1978.

[22] B. Liskov, L. Shrira and J. Wroclawski, "Efficient At-Most-Once Messages Based on Synchronized Clocks," *ACM Trans. on Computers*, Vol. 9, No. 2, pp. 125–142.

[23] N. Lynch, Y. Mansour and A. Fekete, "The Data Link Layer: Two Impossibility Results," *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing,* pp. 149–170, August 1988.

[24] N. Lynch and M. Tuttle, "An Introduction to Input/Output Automata," *CWI Quarterly*, Vol. 2, No. 3, pp. 219–246, September 1989.

[25] Y. Mansour and B. Schieber, "The Intractability of Bounded Protocols for non-FIFO Channels," *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing,* pp. 59–72, August 1989.

[26] S. L. Murphy and A. U. Shankar, "Connection Management for the Transport Layer: Service Specification and Protocol Verification," *IEEE Trans. on Communications,* Vol. 39, No. 12, pp. 1762–1775, December 1991.

[27] R. Rappoport, personal communication, August 1993.

[28] C. A. Sunshine and Y. K. Dalal, "Connection Management in Transport Protocols," *Computer Networks,* Vol. 2, pp. 454–473, 1978.

[29] A. Tanenbaum, *Computer Networks*, 2nd ed., Prentice Hall, 1988.

[30] E. Tempero and R. Ladner, "Tight Bounds for Weakly Bounded Protocols," *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing,* pp. 205–209, August 1990.

[31] R. S. Tomlinson, "Selecting Sequence Numbers," *Proc. ACM SIGCOMM/SIGOPS Interprocess Communications Workshop,* pp. 11–23, 1975; in *ACM Operating Systems Review,* Vol. 9, No. 3, 1975.

[32] *Transmission Control Protocol*, DARPA Network Working Group Report RFC-793, University of Southern California, September 1981.

[33] D.-W. Wang and L. Zuck, "Tight Bounds for the Sequence Transmission Problem," *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing,* pp. 73–83, August 1989.

[34] R. W. Watson, "Timer-based Mechanisms in Reliable Transport Protocol Connection Management," *Computer Networks,* Vol. 5, pp. 47–56, 1981.

[35] R. W. Watson, "The Delta-t Transport Protocol: Features and Experience," *Proc. IEEE Conf. on Local Computer Networks,* pp. 399–407, 1989.