The Impossibility of Implementing Reliable Communication in the Face of Crashes

ALAN FEKETE, NANCY LYNCH, YISHAY MANSOUR, AND JOHN SPINELLI

Massachusetts Institute of Technology, Cambridge, Massachusetts

Abstract. An important function of communication networks is to implement reliable data transfer over an unreliable underlying network. Formal specifications are given for reliable and unreliable communication layers, in terms of I/O automata. Based on these specifications, it is proved that no reliable communication protocol can tolerate crashes of the processors on which the protocol runs.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism and concurrency*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*specification techniques*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Connection reset, lower bounds

---

Earlier versions of the result of this paper appear in LYNCH, N. A., MANSOUR, Y., AND FEKETE, A. Data link layer: Two impossibility results. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ont. Canada, Aug. 15–17). ACM, New York, 1988, pp. 149–170, and SPINELLI, J. M. Reliable data communication in faulty computer networks. Ph.D. dissertation. Dept. Elect. Eng. Comput. Sci., Massachusetts Institute of Technology, Cambridge, Mass., and MIT Laboratory for Information and Decision Systems report LIDS-TH-1882, June 1984.

A. Fekete and N. Lynch were supported in part by the National Science Foundation (NSF) under grants CCR 86-11442 and CCR 89-15206, by the Office of Naval Research (ONR) under contracts N00014-85-K-0168 and N00014-91-J-1046, and by the Defense Advanced Research Projects Agency (DARPA) under contracts N00014-83-K-0125 and N00014-89-J-1988.

A. Fekete was also supported by the Research Foundation for Information Technology, University of Sydney.

Y. Mansour was supported in part by a grant of ISEF and by the NSF under grants CCR 86-11442 and CCR 86-57527.

J. Spinelli was supported in part by an NSF graduate fellowship, by NSF grant ECS 83-1698, and by the Army Research Office (ARO) under grant DAAL03-86-K-0171.

Authors' present addresses: A. Fekete, Department of Computer Science, University of Sydney, 2006 Australia; N. Lynch, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge MA 02139; Y. Mansour, I.B.M. T. J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 19598; J. Spinelli, Electrical Engineering Dept., Union College, Schenectady, NY 12308.

1. *Introduction*

Modern computers do not usually operate in isolation, but are connected to other computers by data communication media. Networking software is provided to enable users and application programs located at different machines to interact. This software is often complicated—in fact, it sometimes occupies more of the resources used by system software than does the operating system kernel. In order to control the complexity of networking software, and also to enable different machines in a network to run different networking software, a *layered architecture* is often used. There are many different layered architectures in use in proprietary, governmental, and international networks [6, 12, 17, 18]. Although the exact choice of function for each layer differs in the various networks, the general framework is always the same: Each layer acts according to a protocol that uses the services of the next lower layer, in order to provide enhanced features. For example, in the OSI architecture, the network layer uses a service providing reliable communication between directly connected machines, and provides communication between machines that are connected only indirectly. A general account of layering can be found in [14].

Reliable delivery of information is one important service that is provided in at least one layer in most layered networks. For example, the HDLC protocol for the data link layer of the OSI architecture [18] provides reliable transfer of data between directly connected machines, using the physical layer service of an unreliable bit channel: The physical layer can generally corrupt, lose, or duplicate messages, but the HDLC protocol guarantees exactly-once, FIFO delivery. In layered architectures, data corruption is often detected using checksums, and the loss of a message is compensated for by retransmission. Such retransmissions can lead to the arrival of duplicate messages. Since a reliable service must not pass duplicate messages to the higher layers, each message is usually tagged with a sequence number, which is also mentioned in the corresponding acknowledgment. Many algorithms have been developed based on these ideas, such as the Alternating Bit Protocol [4], in which only the low-order bit of the sequence number is actually used. Common protocols such as HDLC use these algorithms.

Protocols based on tagging messages with a sequence number require each end station to remember the current sequence number. If this information is kept in volatile storage, and if a crash destroys that storage at one station, then the protocol will be restarted at that station in its initial state, and therefore will assign sequence number 1 (as initially) to the next message. If the other station was still expecting a different sequence number, the first message after the crash might not be delivered. (It might be treated like a retransmission of a previous message and ignored.) Thus, some mechanism is needed in the protocol for one station to cause the other to reinitialize its sequence number also.

One such mechanism is for the station on the machine that has crashed to send a special control message to the other station. (In HDLC, this is a "Set Normal Response Mode" (SNRM) message.) When this control message is received, the other station reinitializes its sequence number and other data structures. The control message is acknowledged by its recipient, and data messages (or data acknowledgments) are sent by the station on the crashed machine only after the reinitialization acknowledgment has been received. Of course, the reinitialization message itself might be lost; to handle this possibil-

ity, the crashed station uses a timeout to determine when to resend the reinitialization message. The HDLC reinitialization protocol is based on the ideas just sketched. In [3], Baratz and Segall examine this protocol, and find it to be incorrect in that reliable delivery is not guaranteed even for messages sent after reinitialization has completed. That is, it is possible for a pattern of failure and message delay to cause an execution of the protocol in which a sequence of data items is accepted from the higher layer at one end after reinitialization, but the sequence delivered at the other end is different.

In [3], Baratz and Segall present an alternative mechanism for reinitializing the sequence numbers and other data structures; their mechanism is applicable to a wide range of reliable communication protocols. Their method involves tagging the reinitialization control messages and their acknowledgments with a bit whose value alternates between reinitialization episodes. This bit must be remembered across crashes, and therefore it cannot be stored in volatile memory.[1] Baratz and Segall conjecture that some nonvolatile storage is needed in *any* protocol that reinitializes values so as to provide reliable data transfer after reinitialization has completed. This paper is devoted to formalizing this impossibility claim and proving it rigorously.

Formal correctness proofs for particular communication protocols are fairly common in the study of computer networks, but there are few examples so far of impossibility results. A survey of such results in distributed computation can be found in [8]. Proving an impossibility result requires a formal model for specifications in which one can describe the task being considered, a formal model for implementations in which one can express any conceivable protocol to perform the given task, and a definition of when a protocol (as described in the model) is correct according to a specification (as described in the model). In this paper, we use the input/output automaton model from [9] and [10] for these purposes.

In order to state an impossibility result in the strongest form, one should specify the task to be performed in as *weak* a fashion as possible; that is, the specification should place few requirements on the protocol. (Of course, the task must not be described so weakly that it becomes possible to accomplish it!) In this paper, the task is reliable data communication using the unreliable service of a lower layer. We use a weak specification for reliable data communication, which states that each message is delivered at most once, and that every message sent after the last crash is delivered exactly once. This specification does not include stronger guarantees such as reliable delivery of messages sent before a crash, or FIFO delivery of those messages that are delivered. Although such properties are desirable for *users* of a reliable communication service, they are not necessary for proving our impossibility result. The impossibility result we give for the weak specification immediately implies corresponding impossibility results for specifications with stronger guarantees.

Since the reliable layer uses the lower unreliable layer without knowledge of the details of the lower layer's implementation, a correct protocol is required to work correctly with *every* implementation of the unreliable layer. Thus, to make the impossibility result as strong as possible, one should make the description of the lower layer as *strong* as possible; this places fewer require-

[1] Since the value of this bit is not used during normal operation, there is little practical disadvantage in keeping it on disk.

ments on the protocol, since it is then required to work with fewer implementations of the unreliable layer, that is, those having strong constraints. We use a strong specification for unreliable data communication, which allows messages to be lost, but does guarantee at-most-once FIFO delivery. The impossibility result we state in terms of an unreliable layer with these strong guarantees applies a fortiori to situations where the reliable layer must cope with a larger range of faults in the unreliable layer.

As an example of the application of the impossibility result, the ISO transport protocol class 4 and Internet TCP protocols provide *ordered* reliable end-to-end service using a network service that may lose *or reorder* data. Since the requirements for reliable message delivery are stronger than those in our result and the assumptions about the unreliable layer are weaker than those in our result, our impossibility result applies to this situation. It implies that for these protocols to guarantee to correctly initialize a connection after a crash, there must be some information that survives the crash.

In practice, there are several ways in which systems cope with the limitation expressed by the impossibility result. First, some existing protocols (such as HDLC at the data link layer) simply behave incorrectly in some cases. The "reliable" layer may lose a message in the face of certain (unlikely) combinations of requests, crashes, and message delays. This is often accepted by system designers on the basis that the errors only happen infrequently, and even when they occur, higher layers of the system may be able to recover from the problem. Second, some systems keep data that is not volatile, and so will survive a crash of a machine on which the protocol is running. For transport protocols, a hardware clock is sometimes used. This provides information about the current time, and therefore does not return to the initial state when a crash occurs. Another strategy involves keeping a counter known as an *incarnation number* in nonvolatile disk storage, and incrementing it after each crash. Transport layer control messages are tagged with the incarnation number, which enables the protocol to recognize old connection requests. Third, some systems require still stronger assumptions about the unreliable layer than we use. For instance, some existing transport protocols insist that the network layer enforce a known maximum time within which each message must be delivered or destroyed. When the network layer is restricted in this way, correct transport initialization protocols can be obtained, but at the cost of introducing dependencies between the settings of time parameters in different layers. Several of these techniques are described in more detail in [15].

There are several other impossibility results in the literature for communication problems. A sketch of a proof that no protocol can reliably provide either delivery or notification of nondelivery for all messages, including those sent before a crash, is given in [5]. There is a proof [7] that correct connection establishment is impossible when the protocol has a particular form: A single resynchronizing message is sent and acknowledged if no data message is successfully delivered within a fixed timeout period, and each data message is retransmitted after a (possibly different) timeout, until it is acknowledged. The paper by Aho et al. [2] contains a number of impossibility results for synchronous protocols, specifically, lower bounds for the number of states required to solve various communication problems. Afek et al. [1] provide an impossibility proof for reliable transmission using a number of messages that is bounded in the best case, regardless of past faults, when the messages have

bounded headers and the unreliable layer can reorder data messages. Related impossibility results concerning the use of bounded headers with non-FIFO unreliable layers are found in [11], [15], and [16].

The rest of the paper is organized as follows: Section 2 contains a summary of the relevant definitions from the input/output automaton model. Section 3 contains a specification of a *reliable layer*, which represents the reliable communication task to be performed. Section 4 contains a specification of the *unreliable layer*, which the protocol is assumed to have available for its use. Section 5 defines what it means for a protocol to be correct according to the given specifications. Finally, Section 6 contains the impossibility result.

2. *The I/O Automaton Model*

The *input/output automaton* model was defined in [9] as a tool for modeling concurrent and distributed systems. We refer the reader to [9] and to the expository paper [10] for a complete development of the model, plus motivation and examples. Here, we provide a brief summary of those aspects of the model that are needed for our results.

2.1. Actions and Action Signatures.

Fundamental to the model is the identification of the actions possible between an entity and its environment, and the separation of those actions into types depending on where the occurrence is controlled. An entity has inputs that are under the control of the environment, outputs that are under the control of the entity and detectable by the environment, and internal actions that are controlled by the entity but not detectable by the environment.

Formally, an *action signature* S is an ordered triple consisting of three pairwise-disjoint sets of actions. We write $in(S)$, $out(S)$, and $int(S)$ for the three components of S, and refer to the actions in the three sets as the *input actions*, *output actions*, and *internal actions* of S, respectively. We let $ext(S) = in(S) \cup out(S)$ and refer to the actions in $ext(S)$ as the *external actions* of S. We let $acts(S) = in(S) \cup out(S) \cup int(S)$, and refer to the actions in $acts(S)$ as the *actions* of S.

2.2. Input / Output Automata.

In the I/O automaton model, a computational entity (either a whole system, or a process or node within a system) is modeled by a state machine. Formally, an *input/output automaton* A (also called an *I/O automaton* or simply an *automaton*) consists of five components:

(1) an action signature $sig(A)$,
(2) a set $states(A)$ of *states*,
(3) a nonempty set $start(A) \subseteq states(A)$ of *start states*,
(4) a transition relation $steps(A) \subseteq (states(A) \times acts(sig(A)) \times states(A))$, with the property that for every state s' and input action π there is a transition (s', π, s) in $steps(A)$, and
(5) an equivalence relation $part(A)$ on $out(sig(A)) \cup int(sig(A))$, having at most countably many equivalence classes.

For brevity, we write $in(A)$ for $in(sig(A))$, $out(A)$ for $out(sig(A))$, and so on.

We refer to an element (s', π, s) of $steps(A)$ as a *step* of A. If (s', π, s) is a step of A, then π is said to be *enabled* in s'. Since every input action is enabled in every state, automata are said to be *input-enabled*. The partition

part(A) is an abstract description of the underlying components of the automaton, and is used to define fairness.

An *execution fragment* of A is a finite sequence $s_0\pi_1 s_1\pi_2 \cdots \pi_n s_n$ or an infinite sequence $s_0\pi_1 s_1\pi_2 \cdots \pi_n s_n \cdots$ of alternating states and actions of A such that $(s_i, \pi_{i+1}, s_{i+1})$ is a step of A for every i. An execution fragment beginning with a start state is called an *execution*.

A *fair execution* of an automaton A is defined to be an execution α of A such that the following condition holds for each class C of *part*(A): If α is finite, then no action of C is enabled in the final state of α, while if α is infinite, then either α contains infinitely many events from C, or else α contains infinitely many occurrences of states in which no action of C is enabled. Thus, a fair execution gives "fair turns" to each class of *part*(A). Informally, one class of *part*(A) typically consists of all the actions that are controlled by a single subsystem within the system modeled by the automaton A, and so fairness means giving each such subsystem regular opportunities to take a step under its control, if any is enabled. In the common case that there is no lower level of structure to the system modeled by A (when *part*(A) consists of a single class), a fair execution is an execution in which infinitely often the automaton is given an opportunity to take an action under its control if any is enabled.

The *behavior* of an execution fragment α of A is the subsequence of α consisting of external actions, and is denoted by *beh*(α). That is, *beh*(α) is formed by removing from the sequence α all states and also those actions in *int*(A). We say that β is a *behavior* of A if β is the behavior of an execution of A. We say that β is a *fair behavior* of A if β is the behavior of a fair execution of A. When an algorithm is modeled as an I/O automaton, it is the set of fair behaviors of the automaton that reflect the activity of the algorithm that is important to users.

We say that a finite behavior β of A *can leave A in state s* if there is a finite execution α with β as its behavior, such that the final state in α is s.

A fundamental operation that we sometimes apply to sequence β of actions (or other elements), such as a behavior, is to take the subsequence consisting of those actions that are in a set Φ of actions. We call this the *projection* of β on Φ, and denote it by $\beta|\Phi$. For brevity, we write $\beta|A$ for $\beta|acts(A)$.

2.3. COMPOSITION. The most useful way of combining I/O automata is by means of a composition operator, as defined in this subsection. This models the way algorithms interact, as for example when the pieces of a communication protocol at different nodes and a lower-level protocol all work together to provide a higher-level service.

A collection $\{A_i\}_{i \in I}$ of automata is said to be *strongly compatible* if no action is an output of more than one automaton in the collection, any internal action of any automaton does not appear in the signature of another automaton in the collection, and no action occurs in the signatures of an infinite number of automata in the collection. Formally, we require that for all $i,j \in I$, $i \neq j$, we have

(1) $out(A_i) \cap out(A_j) = \emptyset$,
(2) $int(A_i) \cap acts(A_j) = \emptyset$, and
(3) no action is in $acts(A_i)$ for infinitely many i.

The *composition* $A = \Pi_{i \in I} A_i$ of a strongly compatible collection of automata $\{A_i\}_{i \in I}$ has the following components:

(1) $in(A) = \bigcup_{i \in I} in(A_i) \setminus \bigcup_{i \in I} out(A_i)$, $out(A) = \bigcup_{i \in I} out(A_i)$, and $int(A) = \bigcup_{i \in I} int(A_i)$,
(2) $states(A) = \Pi_{i \in I} states(A_i)$
(3) $start(A) = \Pi_{i \in I} start(A_i)$
(4) $steps(A)$ is the set of triples (s_1, π, s_2) such that for all $i \in I$, if $\pi \in acts(A_i)$ then $(s_1[i], \pi, s_2[i]) \in steps(A_i)$, and if $\pi \notin acts(A_i)$ then $s_1[i] = s_2[i]$,[2] and
(5) $part(A) = \bigcup_{i \in I} part(A_i)$.

Since the automata A_i are input-enabled, so is their composition, and hence their composition is an automaton. Each step of the composition automaton consists of all the automata that have a particular action in their signatures performing that action concurrently, while the automata that do not have that action in their signatures do nothing. The partition for the composition is formed by taking the union of the partitions for the components. Thus, a fair execution of the composition gives fair turns to all of the classes within all of the component automata. In other words, all component automata in a composition continue to act autonomously. If $\alpha = s_0 \pi_1 s_1 \cdots$ is an execution of A, let $\alpha | A_i$ be the sequence obtained by deleting $\pi_j s_j$ when π_j is not an action of A_i, and replacing the remaining s_j by $s_j[i]$.

The following basic results relate executions and behaviors of a composition to those of the automata being composed. The first result says that the projections of executions of a composition onto the components are executions of the components, and similarly for behaviors, etc. The parts of this result dealing with fairness depend on the fact that at most one component automaton can impose preconditions on each action.

LEMMA 2.1. *Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata, and let $A = \Pi_{i \in I} A_i$. If α is an execution of A, then $\alpha | A_i$ is an execution of A_i for all $i \in I$. Moreover, the same result holds for fair executions, behaviors, and fair behaviors in place of executions.*

Certain converses of the preceding lemma are also true. Behaviors of component automata can be patched together to form schedules or behaviors of the composition.

LEMMA 2.2. *Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata, and let $A = \Pi_{i \in I} A_i$. Let β be a sequence of actions in $acts(A)$. If $\beta | A_i$ is a fair behavior of A_i for all $i \in I$, then β is a fair behavior of A. Also, if $\beta | A_i$ is a behavior of A_i that can leave A_i in state s_i, for all $i \in I$, then β is a behavior of A that can leave A in a state s where $s[i] = s_i$ for all $i \in I$.*

2.4. HIDING OUTPUT ACTIONS. We now define an operator that hides a designated set of output actions in a given automaton to produce a new automaton in which the given actions are internal. Namely, suppose A is an I/O automaton and $\Phi \subseteq out(A)$ is any subset of the output actions of A. Then we define a new automaton, $hide_\Phi(A)$, to be exactly the same as A except for its signature component. For the signature component, we have

[2] We use the notation $s[i]$ to denote the ith component of the state vector s.

$in(hide_\Phi(A)) = in(A)$, $out(hide_\Phi(A)) = out(A) \setminus \Phi$, and $int(hide_\Phi(A)) = int(A) \cup \Phi$.

2.5. SPECIFICATIONS. To specify an entity, we give a set of acceptable patterns of interaction between the entity and its environment. Formally,[3] a *specification* T consists of two components:

(1) an action signature $sig(T)$ having no internal actions, and
(2) a set $behs(T)$ of sequences (finite or infinite) of elements of $acts(sig(T))$, called the *behaviors* of T.

For brevity, we write $in(T)$ for $in(sig(T))$ and so on. We also write $\beta|T$ for $\beta|acts(T)$.

2.6. AN AUTOMATON SATISFYING A SPECIFICATION. To express the fact that an entity modeled by an automaton A is satisfactory for a task modeled by a specification T, we use the following definition: We say that A *satisfies* T provided $in(A) = in(T)$, $out(A) = out(T)$, and also every fair behavior of A is an element of $behs(T)$.

3. *The Reliable Layer*

In this section, we give a specification for the weak type of reliable layer that we wish to implement.

We assume that the reliable layer interacts with higher layers at two endpoints, a *transmitting station* and a *receiving station*. The reliable layer accepts messages from the higher layer at the transmitting station, and delivers some of them to the higher layer at the receiving station. In this paper, we consider the situation in which nodes may crash, losing the information in their state. Therefore, the specification includes events that model these crashes, and the reliability provided is only conditional on no later crash occurring. That is, the reliable layer guarantees that every message that is sent is eventually received, assuming that the end stations remain active. We do not insist that the order of the messages be preserved, as discussed in Section 1.

We describe the reliable layer formally as a specification RL. Let M be a fixed alphabet of "*messages*." The action signature $sig(RL)$ is illustrated in Figure 1, and is given formally as follows:

Input actions
  $send(m)$, $m \in M$
  $crash^t$
  $crash^r$

Output actions:
  $rcv(m)$, $m \in M$

The $send(m)$ action represents the sending of message m on the reliable layer by the transmitting station, and the $rcv(m)$ represents the receipt of message m by the receiving station. The $crash^t$ and $crash^r$ actions represent notification that the transmitting or receiving station, respectively, has suffered a hardware crash failure. In the distributed implementations of the reliable layer to be considered later in the paper, these events will trigger the return to

[3] This is a special case of a *schedule module* as defined in [9].

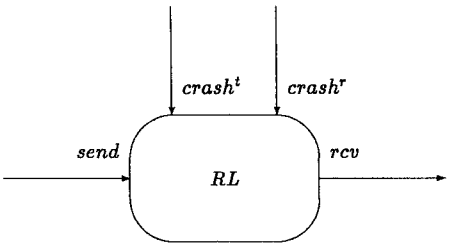


FIG. 1. The reliable layer.

initial state in the appropriate automaton. We refer to the actions in $acts(RL)$ as *reliable layer actions*.

In order to define the set $behs(RL)$, we define a collection of auxiliary properties. These properties are defined with respect to $\beta = \pi_1\pi_2 \cdots$, a (finite or infinite) sequence of reliable layer actions, and a total function *cause* from the indices in β of *rcv* events to the indices of *send* events. This function is intended to model the association that can be set up between the event modeling the receipt of a packet and the event modeling the sending of the same packet. This function is needed to deal carefully with the fact that the same data might be sent repeatedly, and in that case the sequence will contain multiple occurrences of the same action.

The first property expresses the idea that an effect (i.e., a *rcv* event) must occur after its cause (i.e., a corresponding *send* event).

(RL1) If $\pi_i = rcv(m)$, $\pi_j = send(n)$, and $cause(i) = j$, then $j < i$ (i.e., the event π_j precedes π_i in β).

The next property indicates that messages are not corrupted.

(RL2) If $\pi_i = rcv(m)$, $\pi_j = send(n)$, and $cause(i) = j$, then $m = n$.

The next property indicates that messages are not duplicated.

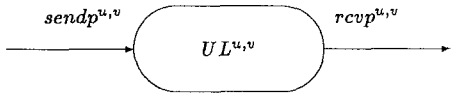
(RL3) The function *cause* is one-to-one (i.e., $cause(i_1) \neq cause(i_2)$ for $i_1 \neq i_2$).

So far, the properties listed have been safety properties; that is, when they hold for a sequence they also hold for any prefix of that sequence. The final property is a liveness property asserting when messages are required to be delivered by the reliable layer. It says that all messages that are sent are eventually delivered, provided no later crashes occur. We use the following terminology: A *crash interval* is a maximal contiguous subsequence of β not containing any $crash^t$ or $crash^r$ events; thus, the crash intervals of β are the sequences of events between successive crash events, together with the sequence of events before the first crash and the sequence of events after the last crash. We say that a crash interval of β is *unbounded* if it is not followed in β by a crash event.

(RL4) If π_i is a $send(m)$ event occurring in an unbounded crash interval in β, then there is an index j of an *rcv* event in β such that $cause(j) = i$.

We say that a sequence β of reliable layer actions is *RL-consistent* provided there exists a function *cause* such that all the conditions (RL1)–(RL4) are satisfied. We extend the use of the term, and say that any sequence (possibly including actions other than reliable layer actions, and possibly including

FIG 2. The unreliable layer.



states) is RL-consistent provided that the subsequence consisting of reliable layer actions is.

Now we can define the specification RL. We have already defined $sig(RL)$. Let $behs(RL)$ be the set of sequences β of reliable layer actions that are RL-consistent.

4. *The Unreliable Layer*

In this section, we define the strong type of unreliable layer that we assume is available for our protocols to use.

We again assume that there are two endpoints, a *transmitting station* and a *receiving station*. The unreliable layer accepts messages, which we call *packets* in order to distinguish them from the messages of the reliable layer, from the higher layer at the transmitting station, and delivers some of them at the receiving station. We do not consider corruption, duplication, or reordering of packets; the only faulty behavior we consider is loss of packets.

4.1. DEFINITIONS. We describe the unreliable layer formally as a specification. Since construction of a reliable layer will generally need *two* unreliable channels, carrying packets in opposite directions, we parameterize the specification by an ordered pair (u, v) of names for the transmitting and receiving stations, respectively. The specification is denoted by $UL^{u,v}$. Let P be a fixed alphabet of "*packets*." The action signature $sig(UL^{u,v})$ is illustrated in Figure 2 and given formally as follows:

Input actions:
$sendp^{u,v}(p)$, $p \in P$

Output actions:
$rcvp^{u,v}(p)$, $p \in P$

The $sendp^{u,v}(p)$ action represents the sending of packet p on the unreliable layer by the transmitting station, and the $rcvp^{u,v}(p)$ represents the receipt of packet p by the receiving station. We refer to the actions in $acts(UL^{u,v})$ as *unreliable layer actions* (for (u, v)).

In order to define the set of behaviors for the specification $UL^{u,v}$, we again define a collection of auxiliary properties. The properties are defined with respect to a sequence $\beta = \pi_1\pi_2 \cdots$ of unreliable layer actions, and a function *cause* from the indices in β of the $rcvp^{u,v}$ events to the indices of $sendp^{u,v}$ events. The first three properties are analogous to those for the unreliable layer.

(UL1) If $\pi_i = rcvp^{u,v}(p)$, $\pi_j = sendp^{u,v}(q)$, and $cause(i) = j$, then $j < i$ (i.e., the event π_j precedes π_i in β).
(UL2) If $\pi_i = rcvp^{u,v}(p)$, $\pi_j = sendp^{u,v}(q)$, and $cause(i) = j$, then $p = q$.
(UL3) The function *cause* is one-to-one (i.e., $cause(i_1) \neq cause(i_2)$ for $i_1 \neq i_2$).

The next property is the FIFO property. It says that those packets that are delivered have their *rcvp* events occurring in the same order as their *sendp*

events. Note that (UL4) may be true even if a packet is delivered and some packet sent earlier is not delivered; there can be gaps in the sequence of delivered packets representing lost packets.

(UL4) (FIFO) Suppose that $cause(i) = j$ and $cause(k) = l$. Then $i < k$ if and only if $j < l$.

The remaining property is the liveness property for the unreliable layer. It says that if repeated send events occur for a particular packet value, then eventually some copy is delivered.

(UL5) For any p, if infinitely many $sendp^{u,v}(p)$ actions occur in β, then infinitely many $rcvp^{u,v}(p)$ actions occur in β.

We say that a sequence β of unreliable layer actions is $UL^{u,v}$-*consistent*, provided there exists a function *cause* such that all the conditions (UL1)–(UL5) are satisfied. As before, we extend the use of the term, and say that any sequence is $UL^{u,v}$-consistent provided that the subsequence consisting of unreliable layer actions is. We have the following simple consequences of the definitions.

LEMMA 4.1

(1) *Suppose* β *and* γ *are* $UL^{u,v}$*-consistent. Then* $\beta\gamma$ *is* $UL^{u,v}$*-consistent.*
(2) *Suppose* β *is* $UL^{u,v}$*-consistent and* β' *is a prefix of* β*. Then* β' *is* $UL^{u,v}$*-consistent.*

Now we define the specification $UL^{u,v}$. We have already defined $sig(UL^{u,v})$. Let $behs(UL^{u,v})$ be the set of sequences β of unreliable layer actions that are $UL^{u,v}$-consistent.

We define an *unreliable channel* from u to v to be any I/O automaton that satisfies $UL^{u,v}$. Thus, C is an unreliable channel if it has the external actions appropriate for the specification, and also every fair behavior satisfies the conditions above (for some choice of the function *cause*). An unreliable channel with the largest set of fair behaviors is called "universal;" formally, a *universal unreliable channel* is an unreliable channel whose set of fair behaviors is exactly the set of $UL^{u,v}$-consistent sequences.

4.2. PROPERTIES OF THE UNRELIABLE LAYER. In this subsection, we give some basic properties of the unreliable layer and of unreliable channels.

We first define the idea of a sequence of packets being "in transit" after a behavior of the unreliable layer. If $\beta = \pi_1\pi_2 \cdots$ is a finite $UL^{u,v}$-consistent sequence, we say that a sequence of packets $Q = q_1q_2 \cdots q_k$ is *in transit* after β, provided there is a function *cause* such that properties (UL1)–(UL5) hold for β and *cause*, and also there are indices $i_1, i_2, \ldots, i_k$ with the following properties:

—$i_1 < i_2 < \cdots < i_k$,
—$\pi_{i_j} = sendp^{u,v}(q_j)$ for each j, $1 \le j \le k$, and
—for any index j of a $rcvp^{u,v}$ event in β, $cause(j) < i_1$.

That is, a sequence of packets is in transit after β if it is a subsequence of the collection of packets sent after the sending of the last packet that is success-

fully delivered. Notice, as a consequence of this definition, that if a sequence Q is in transit after β, then so is any subsequence of Q.

LEMMA 4.2. *If β is a finite $UL^{u,v}$-consistent sequence of unreliable layer actions, Q is a sequence of packets that is in transit after β, and Q' is a subsequence of Q, then Q' is in transit after β.*

Another immediate consequence of the definition is the following lemma, which says that as further packets are sent, they can be added to the sequence in transit.

LEMMA 4.3. *If β is a finite $UL^{u,v}$-consistent sequence of unreliable layer actions, $q_1 q_2 \cdots q_k$ is in transit after β, and $q'_1 q'_2 \cdots q'_l$ is a finite sequence of packets, then the sequence*

$$\beta' = \beta sendp^{u,v}(q'_1) sendp^{u,v}(q'_2) \cdots sendp^{u,v}(q'_l)$$

is a $UL^{u,v}$-consistent sequence and the sequence of packets $q_1 q_2 \cdots q_k q'_1 \cdots q'_l$ is in transit after β'.

The following lemma says that, any sequence of packets in transit can be delivered without violating the specification of an unreliable layer.

LEMMA 4.4. *If β is a finite $UL^{u,v}$-consistent sequence of unreliable layer actions, and $Q = q_1 q_2 \cdots q_k$ is a sequence of packets that is in transit after β, then $\beta rcvp^{u,v}(q_1) \cdots rcvp^{u,v}(q_k)$ is a $UL^{u,v}$-consistent sequence.*

Recall that a universal unreliable channel is an unreliable channel whose fair behaviors are all the sequences allowed by the specification $UL^{u,v}$, rather than merely a subset of these. For our later work, it will be important to know that a universal unreliable channel exists. We give the construction here, and leave it to the reader to check that this automaton has the required behaviors. Note that no property of the automaton is used in this paper other than the fact that it is universal.

The I/O automaton $\hat{C}^{u,v}$ has the inputs and outputs of $UL^{u,v}$, and no internal actions. The state of $\hat{C}^{u,v}$ consists of a sequence *queue* of packets, an array *count* of integers indexed by packet values, and a array *keep* of infinite sets of positive integers indexed by packet values. The initial states of the automaton are those states in which q is empty and each entry *count*[p] is zero. Thus, each initial state is determined by a value for the array *keep*.

The transition relation for the automaton $\hat{C}^{u,v}$ consists of all triples (s', π, s) described by the following code.[4]

$sendp^{u,v}(p)$
Effect: *count*[p] $\leftarrow$ *count*[p] + 1
if *count*[p] $\in$ *keep*[p], then append p to *queue*

$rcvp^{u,v}(p)$
Precondition: p is at head of *queue*
Effect: delete p from front of *queue*

The partition puts all the output actions of $\hat{C}^{u,v}$ in a single class.

[4] This style of describing I/O automata by giving preconditions (i.e., conditions on s') and effects (i.e., imperatives to be executed sequentially to transform s' to give s) is used in [10]. It is not fundamental to the model, but is rather a notational convenience for describing sets of triples.

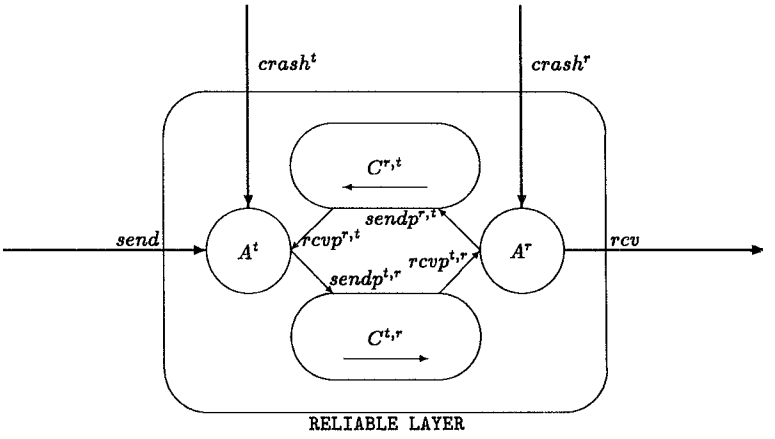


FIG. 3. A reliable layer implementation.

Thus, $i \in keep[p]$ means that the ith time packet value p is sent; it will succeed in being delivered. The fact that each $keep[p]$ is infinite ensures that (UL5) is satisfied by fair behaviors of $\hat{C}^{u,v}$.

LEMMA 4.5. *The automaton $\hat{C}^{u,v}$ is a universal unreliable channel.*

5. *Reliable Layer Implementation*

In this section, we define a "reliable communication protocol," which is intended to be used to implement the reliable layer using the services provided by the unreliable layer. A reliable communication protocol consists of two automata, one at the transmitting station and one at the receiving station. These automata communicate with each other using two unreliable channels, one in each direction. They also communicate with the outside world, through the reliable layer actions we defined in Section 3.

Figure 3 shows how two protocol automata and two unreliable channels should be connected, in a reliable layer implementation.

5.1. RELIABLE COMMUNICATION PROTOCOLS.

We define a reliable communication protocol syntactically, as two automata that have the correct action names to be used in a system connected as in Figure 3.

A *transmitting automaton* is any I/O automaton having an action signature as follows:

Input actions:
  $send(m)$, $m \in M$
  $rcvp^{t',t}(p)$, $p \in P$
  $crash^t$

Output actions:
  $sendp^{t,r}(p)$, $p \in P$

In addition, there can be any number of internal actions. That is, a transmitting automaton receives requests from the environment of the reliable layer to send messages to the receiving station. It also receives packets over the unreliable channel from the receiving station r, and notification of crashes at the transmitting station. It sends packets over the unreliable channel to r.

Similarly, a *receiving automaton* is any I/O automaton having an action signature as follows:

Input actions:
$rcvp^{t,r}(p)$, $p \in P$
$crash^r$

Output actions:
$sendp^{r,t}(p)$, $p \in P$
$rcv(m)$, $m \in M$

Again, there can also be any number of internal actions. That is, a receiving automaton receives packets over the unreliable channel from the transmitting station t, and notification of crashes at the receiving station. It sends packets to t over the unreliable channel to t, and it delivers messages to the environment of the reliable layer.

A *reliable communication protocol* is a pair (A^t, A^r), where A^t is a transmitting automaton and A^r is a receiving automaton.

We close this subsection with a lemma describing a useful property of reliable communication protocols interacting with an unreliable layer. It says that from any point in an execution, the system can continue to run in some way, with no further crashes nor requests for message transfer, so that no packets sent before that point are delivered after it.

Recall that for any specification T and sequence β we write $\beta|T$ for the subsequence of β consisting of actions of T. For brevity, we say that β is UL-consistent provided $\beta|UL^{t,r}$ is $UL^{t,r}$-consistent and $\beta|UL^{r,t}$ is $UL^{r,t}$-consistent.

LEMMA 5.1. *Let (A^t, A^r) be a reliable communication protocol. Let α be a finite UL-consistent execution of $A = A^t \circ A^r$. Then there exists a fair UL-consistent execution $\alpha\beta$ of A such that*

(1) *β contains no send or crash events, and*
(2) *β is UL-consistent*

PROOF. (Sketch) The sequence β is constructed inductively, interleaving transitions that involve actions from each equivalence class of the fairness partition of A. However, whenever a $sendp(p)$ event is added to the execution, it is immediately followed by a corresponding $rcvp(p)$ event. This is allowed by A since $rcvp(p)$ is an input to the composition, and UL-consistency is obviously maintained. The dovetail ensures that the execution $\alpha\beta$ constructed is a fair execution of A. Since every *sendp* event is followed by its corresponding *rcvp* event, it follows that the suffix β is UL-consistent. □

5.2. CORRECTNESS OF RELIABLE COMMUNICATION PROTOCOLS. Now we are ready to define correctness of reliable communication protocols. Informally, we say that a reliable communication protocol is "correct" provided that when it is composed with any pair of unreliable channels (from t to r and from r to t, respectively), the resulting system yields correct reliable layer behavior. This reflects the fundamental idea of layering, that the implementation of one layer should not depend on the details of the implementation of other layers, so that each layer can be implemented and maintained independently. Formally, we say that a reliable communication protocol (A^t, A^r) is *correct* provided that the following is true. For all $C^{t,r}$ and $C^{r,t}$ that are unreliable channels from t to r

and from r to t, respectively, $hide_{\Phi}(D)$ satisfies RL, where D is the composition of A^t, A^r, $C^{t,r}$, and $C^{r,t}$, and Φ is the subset of $acts(D)$ consisting of *sendp* and *rcvp* actions. We need to hide the actions between the protocol and the unreliable channels in order that the composition should have the signature required for the reliable layer.[5]

The definition of correctness just given is somewhat difficult to work with, because it involves universal quantification over all possible unreliable channels. We actually work with an alternative characterization, using only behaviors of the composition of A^t and A^r.

THEOREM 5.2. *Let (A^t, A^r) be a reliable communication protocol. Then the following are equivalent.*

(1) *(A^t, A^r) is correct.*
(2) *For every fair behavior β of $A = A^t \circ A^r$, if β is UL-consistent, then β is RL-consistent.*

PROOF. Let Φ be the set of all *sendp* and *rcvp* actions. For one direction of implication, assume that (A^t, A^r) is correct. Let β be a fair behavior of A that is UL-consistent. Let $\hat{C}^{t,r}$ and $\hat{C}^{r,t}$ be the unreliable channels defined in Section 4; Lemma 4.5 implies that these are universal unreliable channels.

Since β is $UL^{t,r}$-consistent, and $\hat{C}^{t,r}$ is a universal unreliable channel, it must be that $\beta|UL^{t,r}$ is a fair behavior of $\hat{C}^{t,r}$. Likewise, $\beta|UL^{r,t}$ is a fair behavior of $\hat{C}^{r,t}$. Then Lemma 2.2 gives that β is a fair behavior of $D = A \circ \hat{C}^{t,r} \circ \hat{C}^{r,t}$. Therefore, $\beta|RL$ is a fair behavior of $hide_{\Phi}(D)$, since the actions of RL are exactly the external actions of D that are not in Φ. Since (A^t, A^r) is correct and $\hat{C}^{t,r}$ and $\hat{C}^{r,t}$ are unreliable channels from t to r and r to t, respectively, any fair behavior of $hide_{\Phi}(D)$ is RL-consistent. Thus, $\beta|RL$ is RL-consistent, which implies that β is RL-consistent, as required.

Conversely, suppose that for every fair behavior β of A, if β is UL-consistent, then β is RL-consistent. Let $C^{t,r}$ and $C^{r,t}$ be arbitrary unreliable channels from t to r and from r to t, respectively, and let $D = A \circ C^{t,r} \circ C^{r,t}$. We must show that $hide_{\Phi}(D)$ satisfies RL.

Let β' be an arbitrary fair behavior of $hide_{\Phi}(D)$. Then there is a fair behavior β of D such that $\beta' = \beta|RL$. By Lemma 2.1, $\beta|C^{t,r}$ is a fair behavior of $C^{t,r}$, and since $C^{t,r}$ is an unreliable channel, $\beta|C^{t,r}$ is $UL^{t,r}$-consistent. That is, $\beta|UL^{t,r}$ is $UL^{t,r}$-consistent. Likewise, $\beta|UL^{r,t}$ is $UL^{r,t}$-consistent. Thus, β is UL-consistent. By hypothesis, β is RL-consistent, and so β' is RL-consistent. Thus, $\beta' \in behs(RL)$, as required. □

5.3. CRASHING PROTOCOLS. In this subsection, we define a constraint for reliable communication protocols: a "crashing" property, which says that a crash at either the transmitting or receiving station causes the corresponding protocol automaton to revert back to its start state (thereby losing all information in its memory). This property models the absence of nonvolatile storage.

We say that a transmitting automaton A^t is *crashing*, provided that there is a unique start state q_0, that $(q, crash^t, q_0)$ is a step of A^t, for every $q \in states(A^t)$, and that these are the only $crash^t$ steps. Similarly, we say that a receiving automaton A^r is *crashing*, provided that there is a unique start state q_0, that

[5] Recall that in the I/O automaton model, actions between components of a system are outputs of the system as a whole.

$(q, crash^r, q_0)$ is a step of A^r, for every $q \in states(A^r)$, and that these are the only $crash^r$ steps. A reliable communication protocol (A^t, A^r) is said to be *crashing*, provided that A^t and A^r are both crashing.

6. *The Impossibility Proof*

A useful property for a reliable communication protocol would be the ability to tolerate crashes of the machines on which it runs. We consider the case in which a crash causes all the memory at the site to be lost; we model this by having a crash cause the automaton at that site to revert to its initial state. In this section, we present our impossibility result, that no correct reliable communication protocol can tolerate arbitrary crashes (without access to some nonvolatile memory).

The main idea of our proof is to assume the existence of a reliable communication protocol that is both correct and crashing, and to find two finite executions, α and $\hat{\alpha}$, that leave both the transmitting and receiving automata in the same states, although in α every message has been delivered and in $\hat{\alpha}$ there is an undelivered message. The protocol must eventually deliver the missing message in any fair extension of $\hat{\alpha}$ in which no more crashes occur, even if no further messages are submitted by the environment. Then a corresponding extension of α will cause some message to be delivered, although every message sent had already been delivered. This contradicts the claimed correctness of the protocol.

In our proof, α contains the sending and delivery of a single message, while $\hat{\alpha}$ contains many crash events and ends with the sending of a message that is not delivered. The construction of $\hat{\alpha}$ from α is given in Lemma 6.3, using the following observation: It is possible to find a behavior that can leave the end stations in the same states that they have after step k of the execution α, but where a particular sequence of packets (which are received by one station in the first k steps of α) are in transit. This is shown carefully in Lemma 6.2 by induction. The induction step (which is Lemma 6.1) uses the fact that the inputs, up to step k of α, of a given station depend on outputs of the other station up to step $k - 1$.

We now begin the rigorous proof, following the sketch above. We first establish some notation. For $x \in \{t, r\}$, we define $\bar{x}$ so that $\bar{x} \in \{t, r\}$ and $x \neq \bar{x}$, that is, $\bar{t} = r$ and $\bar{r} = t$. For a finite execution $\alpha = s_0 \pi_1 s_1 \cdots \pi_n s_n$ of $A^t \circ A^r$, $x \in \{t, r\}$, and an integer k, $0 \leq k \leq n$, we define the following:

—$in(\alpha, x, k)$ is the sequence of packets received by A^x during $\pi_1 \pi_2 \cdots \pi_k$, the first k steps of α,
—$out(\alpha, x, k)$ is the sequence of packets sent by A^x during the first k steps of α,
—$state(\alpha, x, k)$ is the state of A^x in s_k,
—$ext(\alpha, x, k)$ is the sequence of external actions of A^x during the first k steps of α.

Note that if α is *UL*-consistent, then $in(\alpha, x, k)$ is a subsequence of $out(\alpha, \bar{x}, k - 1)$.

The first lemma is used for the inductive step in the inductive proof of Lemma 6.2. Speaking informally, we use it to "pump up" the sequence of packets waiting in the channels, as illustrated in Figure 4. If a behavior can leave the system so that in transit from $\bar{x}$ to x, there is a sequence of packets

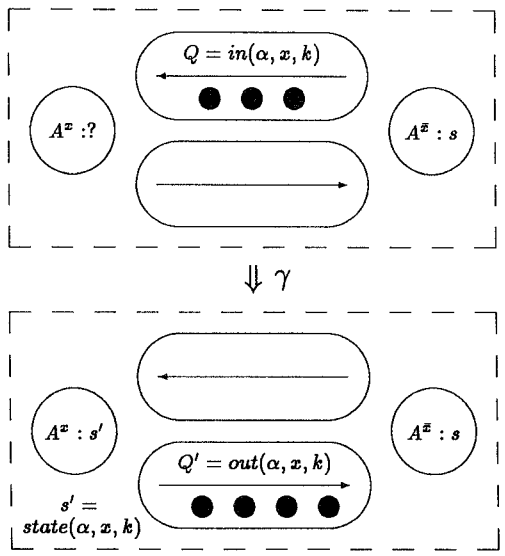


FIG. 4. Illustration for Lemma 6.1.

that is the same as the sequence of packets delivered across that channel in a reference execution, then we can extend the behavior by crashing the destination station A^x and replaying that station's part of the reference execution, and this can leave the system so that a sequence of packets is in transit in the other direction, equal to the packets sent by $A^{\bar{x}}$ in the reference execution.

LEMMA 6.1. *Let (A^t, A^r) be a crashing reliable communication protocol. Let $\alpha = s_0\pi_1 s_1 \cdots \pi_n s_n$ be a finite UL-consistent execution of $A = A^t \circ A^r$ such that no crash events occur in $\pi_1 \cdots \pi_n$. Suppose $x \in \{t, r\}$, k is an integer with $0 \le k \le n$ and β is a finite UL-consistent behavior of A with the following properties*:

(1) *β can leave A in a state where the state of $A^{\bar{x}}$ is s, and*
(2) *the sequence $in(\alpha, x, k)$ of packets is in transit from $\bar{x}$ to x after β.*

Let $\gamma = crash^x ext(\alpha, x, k)$, a sequence of actions of A^x. Then we have the following properties of $\beta\gamma$:

(1) *$\beta\gamma$ is a finite UL-consistent behavior of A,*
(2) *$\beta\gamma$ can leave A in the state where the state of $A^{\bar{x}}$ is s, and the state of A^x is $state(\alpha, x, k)$, and*
(3) *the sequence $out(\alpha, x, k)$ of packets is in transit from x to $\bar{x}$ after $\beta\gamma$.*

PROOF. As notation, let q_1, q_2 etc. denote the packets such that $in(\alpha, x, k) = q_1 q_2 \cdots q_l$. We consider the sequence $\beta\gamma$.

Now $\beta\gamma|A^x$ is just $(\beta|A^x)crash^x(ext(\alpha, x, k))$. Since $\beta|A^x$ is a behavior of A^x, $crash^x$ is an input of A^x that takes A^x to its initial state, and $ext(\alpha, x, k)$ is the behavior of an execution fragment of A^x that starts in the initial state of A^x and ends in $state(\alpha, x, k)$, we deduce that $\beta\gamma|A^x$ is a finite behavior of A^x that can leave A^x in state $state(\alpha, x, k)$.

Also, $\beta\gamma|A^{\bar{x}}$ is just $\beta|A^{\bar{x}}$ which is a finite behavior of $A^{\bar{x}}$ that can leave $A^{\bar{x}}$ in state s. By Lemma 2.2, $\beta\gamma$ is a finite behavior of A that can leave A in the state where the state of $A^{\bar{x}}$ is s and the state of A^x is $state(\alpha, x, k)$.

Now $\gamma|UL^{\bar{x},x}$ is $rcvp^{\bar{x},x}(q_1)\cdots rcvp^{\bar{x},x}(q_l)$ by construction. Since Q is in transit from $\bar{x}$ to x after β, we see by Lemma 4.4 that $\beta\gamma|UL^{\bar{x},x}$ is $UL^{\bar{x},x}$-consistent. Also, $\gamma|UL^{x,\bar{x}}$ consists of the sequence of $sendp^{x,\bar{x}}$ actions in $\pi_1\pi_2\cdots\pi_l$. By Lemma 4.3, $\beta\gamma|UL^{x,\bar{x}}$ is $UL^{x,\bar{x}}$-consistent; thus, $\beta\gamma$ is UL-consistent. Lemmas 4.3. and 4.2 together imply that the sequence $out(\alpha, x, k)$ of packets is in transit from x to $\bar{x}$ after $\beta\gamma$. □

The next lemma says that we can find a behavior that can leave the protocol in the same state as in any suitable execution α, and with the same sequence of packets as those sent in α in transit in one of the channels.

LEMMA 6.2. *Let (A^t, A^r) be a crashing reliable communication protocol. Let $\alpha = s_0\pi_1 s_1\cdots\pi_n s_n$ be a finite UL-consistent execution of $A = A^t \circ A^r$ such that no crash events occur in $\pi_1\cdots\pi_n$. Suppose $x \in \{t, r\}$ and k is an integer, with $0 \le k \le n$, such that either $k = 0$ or $\pi_k \in acts(A^x)$. Then there is a finite sequence β with the following properties:*

(1) *β is a UL-consistent behavior of A,*
(2) *β can leave A in the state where the state of A^x is $state(\alpha, x, k)$, and the state of $A^{\bar{x}}$ is $state(\alpha, \bar{x}, k)$, and*
(3) *the sequence $out(\alpha, x, k)$ of packets is in transit from x to $\bar{x}$ after β.*

PROOF. We use induction on k.

The base case, when $k = 0$, is trivial, as $state(\alpha, x, 0)$ is the initial state of A^x, $state(\alpha, \bar{x}, 0)$ is the initial state of $A^{\bar{x}}$, and $out(\alpha, \bar{x}, 0)$ is the empty sequence. Thus, we may take β to be the empty sequence of actions.

Now we suppose that $k > 0$, and we assume inductively that the lemma is true for all smaller values of k.

If all the actions $\pi_1,\cdots,\pi_k$ are in $acts(A^x)$, then $out(\alpha, \bar{x}, k)$ must be the empty sequence, and therefore we deduce that $in(\alpha, x, k)$ is also empty. Also, $state(\alpha, \bar{x}, k)$ must be equal to $state(\alpha, \bar{x}, 0)$. Thus, the empty sequence β_1 is a finite UL-consistent behavior of A, β_1 can leave $A^{\bar{x}}$ in state $state(\alpha, \bar{x}, k)$, and $in(\alpha, x, k)$ is in transit from $\bar{x}$ to x after β_1. We can therefore apply Lemma 6.1 to obtain β as an extension of β_1.

Otherwise, let j be the greatest integer such that $1 \le j \le k$ and $\pi_j \in acts(A^{\bar{x}})$. Notice that in fact $j < k$, since $\pi_k \in acts(A^x)$. Then $in(\alpha, x, k)$ is a subsequence of $out(\alpha, \bar{x}, j)$, and $state(\alpha, \bar{x}, k)$ must equal $state(\alpha, \bar{x}, j)$. By using the inductive hypothesis, we get a finite UL-consistent behavior β_1 of A, where β_1 can leave $A^{\bar{x}}$ in state $state(\alpha, \bar{x}, j)$, and the sequence $out(\alpha, \bar{x}, j)$ is in transit from $\bar{x}$ to x after β_1. By Lemma 4.2, the subsequence $in(\alpha, x, k)$ is also in transit from $\bar{x}$ to x after β_1. We can therefore apply Lemma 6.1 to obtain β as an extension of β_1. □

We can now use Lemma 6.2 to find a behavior of a crashing reliable communication protocol that can lead to states identical to those at the end of a given execution, but in which a message has been sent but not received.

LEMMA 6.3. *Let (A^t, A^r) be a crashing reliable communication protocol. Let $\alpha = s_0\pi_1 s_1\cdots\pi_n s_n$ be a finite UL-consistent execution of $A = A^t \circ A^r$ such that*

$$beh(\alpha)|RL = send(m)rcv(m).$$

Then there is a finite UL-consistent execution, $\hat{\alpha}$, of A with the following properties:

(1) *$\hat{\alpha}|RL$ ends in send(m).*
(2) *$\hat{\alpha}$ ends in a state in which the state of A^t is state(α, t, n) and the state of A^r is state(α, r, n).*

PROOF. Let k denote the greatest integer less than or equal to n such that $\pi_k \in acts(A^r)$. That is, k is the index of the last event in α that occurs at the receiving station (since $rcv(m)$ is an action of A^r, there is some k satisfying this description). Lemma 6.2 yields a finite UL-consistent behavior β' of A with the following properties: β' can leave A in a state where the state of A^r is $state(\alpha, r, k)$, and the sequence $out(\alpha, r, k)$ of packets is in transit from r to t after β'.

Since the sequence $in(\alpha, t, n)$ is a subsequence of $out(\alpha, r, k)$, Lemma 4.2 implies that $in(\alpha, t, n)$ is in transit from r to t after β'.

We now apply Lemma 6.1 to see that, for $\gamma = crash^t ext(\alpha, t, n)$, $\beta'\gamma$ is a finite UL-consistent behavior of A, $\beta'\gamma$ can leave A in the state where the state of A^r is $state(\alpha, r, k)$, and the state of A^t is $state(\alpha, t, n)$. We set $\beta = \beta'\gamma$.

We now note, using the definition of k, that $state(\alpha, r, k) = state(\alpha, r, n)$. Since γ is $crash^t ext(\alpha, t, n)$ and $ext(\alpha, t, n)|RL = (beh(\alpha)|A^t)|RL = send(m)$, we have that $\beta|RL$ ends in $crash^t send(m)$. Let $\hat{\alpha}$ be any finite execution of A with $beh(\hat{\alpha}) = \beta$, that ends in the state where the state of A^r is $state(\alpha, r, k)$ and the state of A^t is $state(\alpha, t, n)$. We know that such $\hat{\alpha}$ must exist, because β can leave A in the indicated state. □

Finally, we can use the results above to prove our impossibility theorem.

THEOREM 6.4. *There is no crashing reliable communication protocol that is correct.*

PROOF. Assume that (A^t, A^r) is such a protocol and let $A = A^t \circ A^r$.

First, we claim that there is a finite UL-consistent execution $\alpha = s_0\pi_1 s_1 \cdots \pi_n s_n$ of A such that $beh(\alpha)|RL = send(m)rcv(m)$. The existence of such an α is proved by starting with an execution of A containing the single action $send(m)$ (which exists since A is input-enabled), and then using Lemma 5.1 to get a fair UL-consistent execution of A whose behavior contains $send(m)$ and no other *send* or *crash* events. By Theorem 5.2, the execution's behavior must be RL-consistent. Since the action $send(m)$ occurs in the behavior and is followed by no *crash* events, property (RL4) implies that an rcv action appears, and (RL2) shows that the action must be $rcv(m)$. By (RL1), it must follow the $send(m)$ action, and (RL3) implies that no other rcv event can appear. We obtain the finite execution α by truncating this fair execution after the state following the $rcv(m)$ event. It follows that $beh(\alpha)|RL$ is $send(m)rcv(m)$.

Next we appeal to Lemma 6.3 to obtain a finite UL-consistent execution $\hat{\alpha} = \hat{s}_0\hat{\pi}_1\hat{s}_1 \cdots \hat{\pi}_k\hat{s}_k$ of A with the following properties: $beh(\hat{\alpha})$ ends in $send(m)$, and $state(\hat{\alpha}, x, k) = state(\alpha, x, n)$ for $x \in \{t, r\}$.

By Lemma 5.1, there is a fair UL-consistent execution of A that extends $\hat{\alpha}$ and contains no additional *send* or *crash* events. The projection of this extension on the reliable layer actions must satisfy (RL4). Since the final $send(m)$ of $\hat{\alpha}$ occurs in the extension in an unbounded crash interval, by (RL4)

and (RL1) the suffix of the extension after $\hat{\alpha}$ contains a *rcv* event. Let α_2 be the subsequence of this extension, starting at the action following the end of $\hat{\alpha}$ and ending at the state after the first following *rcv* event. We see that $\alpha_2|RL = rcv(m')$ for some m' (since the extension contains no *send* or *crash* events), and that α_2 is UL-consistent. Also, the sequence consisting of the final state of $\hat{\alpha}$ following by α_2 is an execution fragment of A.

Since α and $\hat{\alpha}$ end in the same state both in the transmitter and the receiver, the sequence $\alpha_1 = \alpha\alpha_2$ is a finite execution of A. It is UL-consistent since each of α and α_2 are (using Lemma 4.1). Now $beh(\alpha_1)|RL = send(m)rcv(m)rcv(m')$.

Now we use Lemma 5.1 to get a fair UL-consistent extension of α_1 with no additional *send* or *crash* events. The behavior of this extension contains exactly one *send* event and at least two *rcv* events. Clearly no function *cause* can be found for this behavior that satisfies (RL3), so this behavior is not RL-consistent. By Lemma 5.2, this contradicts the assumption that A is a correct crashing reliable communication protocol. □

ACKNOWLEDGMENTS. We thank Baruch Awerbuch and Robert Gallager for many useful discussions. We also thank Jennifer Welch and Boaz Patt-Shamir for their comments on several versions of the paper. Michael Fischer and Lenore Zuck gave us many helpful ideas for the modeling of communication service specifications.

REFERENCES

1. AFEK, Y., ATTIYA, H., FEKETE, A., FISCHER, M., LYNCH, N., MANSOUR, Y., WANG, D., AND ZUCK, L. Reliable communication over unreliable channels. Tech Rep. YALE/DCS/TR-853. Yale Univ., New Haven, Conn Also, *J. ACM*, to appear.
2. AHO, A., ULLMAN, J., WYNER, A., AND YANNAKAKIS, M. Bounds on the size and transmission rate of communication protocols. *Comput. Math Appl. 8* (1982), 205–214.
3. BARATZ, A., AND SEGALL, A Reliable link initialization procedures. *IEEE Trans. Commun. COM-36* (Feb. 1988), 144–152
4. BARTLETT, K., SCANTLEBURY, R., AND WILKINSON, P. A note on reliable full-duplex transmission over half-duplex links. *Commun ACM 12*, 5 (May 1969), 260–261
5. BELSNES, D. Single-message communication. *IEEE Trans. Commun. COM-24* (Feb 1976), 190–193
6. CYPER, R. J. *Communications Architecture for Distributed Systems*. Addison-Wesley, Reading, Mass., 1978.
7. LE LANN, G., AND LE GOFF, H. Verification and evaluation of communication protocols. *Comput. Netw. 2* (Feb. 1978), 50–69.
8. LYNCH, N. A. A hundred impossibility proofs for distributed computing In *Proceedings of 8th Annual ACM Symposium on Principles of Distributed Computing* (Edmonton, Alberta, Canada, Aug. 14–16). ACM, New York, 1989, pp. 1–28.
9. LYNCH, N. A., AND TUTTLE, M. R. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, 1987, pp. 137–151.
10. LYNCH, N. A., AND TUTTLE, M. R. An introduction to input/output automata. *CWI Q. 2*, 3 (Sept. 1989), 219–246
11. MANSOUR, Y., AND SCHIEBER, B. The intractability of bounded protocols for non-FIFO channels *J ACM 39*, 4 (Oct. 1992), 783–799.
12. MCQUILLAN, J. M., AND WALDEN, D C. The ARPA network design decisions. *Comput. Netw. 1* (Aug 1977), 243–289.
13. SUNSHINE, C., AND DALAL, Y. Connection management in transport protocols. *Comput Netw. 2* (Dec. 1978), 454–473.
14. TANENBAUM, A. *Computer Networks*. 2nd ed. Prentice-Hall, Englewood Cliffs, N.J., 1988

15. TEMPERO, E., AND LADNER, R. Tight bounds for weakly bounded protocols. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing* (Quebec City, Que., Canada, Aug. 22–24). ACM, New York, 1990, pp. 205–218.
16. WANG, D., AND ZUCK, L. Tight bounds for the sequence transmission problem. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing* (Edmonton, Alberta, Canada, Aug. 14–16). ACM, New York, 1989, pp. 73–84.
17. WECKER, S. DNA: The Digital Network Architecture. *IEEE Trans. Commun. COM-28* (Apr. 1980), 510–526.
18. ZIMMERMANN, H. OSI reference model–The ISO model of architecture for open systems interconnection. *IEEE Trans. Commun. COM-28* (Apr. 1980), 425–432.

RECEIVED SEPTEMBER 1990; REVISED APRIL 1992; ACCEPTED APRIL 1992