Yale University
Department of Computer Science

Reliable Communication Over Unreliable Channels

Yehuda Afek Hagit Attiya Alan Fekete
Michael Fischer Nancy Lynch Yishay Mansour
Da-Wei Wang Lenore Zuck

YALEU/DCS/TR-853
April 1991

This research was supported in part by Research Foundation for Information Technology, University of Sydney; Department of Computer Science, Tel-Aviv University; IBM Fellowship; ONR contracts N00014–85–K–0168, N00014–91–J–1046, N00014–85–K–0445, and N00014–82–K–0154; by NSF grants CCR–8611442, CCR–8915206, IRI–9015570, DCR–8405478, CCR–8910289; and by DARPA contracts N00014–89–J–1988 and N00014–87–K–0825.

Reliable Communication Over Unreliable Channels

Yehuda Afek[*] Hagit Attiya[†] Alan Fekete[‡] Michael Fischer[§] Nancy Lynch[¶]
Yishay Mansour[||] Da-Wei Wang[§] Lenore Zuck[§]

Abstract

Layered communication protocols frequently build a reliable FIFO message facility on top of an unreliable non-FIFO service such as that provided by a packet-switching network. This can be done by including a sequence number in each transmission and acknowledgement. To accommodate the sequence numbers requires in principle an infinite number of different packet headers. This paper investigates the possibility of building a reliable message facility using only a fixed finite number of different packet headers. Protocols to do this are known when the underlying service can lose packets but not reorder them. Here, the more difficult case in which packets might be delivered out of order is considered. A protocol is given that accomplishes this task, but it has the undesirable property that the number of packets needed to deliver a message increases permanently as additional packet-loss and reordering faults occur. A proof is given that no protocol can avoid such performance degradation. The protocol and proof are presented using I/O automata.

Keywords: reliable communication, reliable layer, unreliable layer, protocol, packet-switching network, packet header, I/O automaton.

[*]Computer Science Department, Tel-Aviv University, Tel-Aviv, Israel, and AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.

[†]Department of Computer Science, Technion, Haifa 32000, Israel.

[‡]Department of Computer Science F09, University of Sydney 2006, Australia.

[§]Department of Computer Science, Yale University, Box 2158 Yale Station, New Haven, CT 06520, U.S.A.

[¶]Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge MA 02139, U.S.A.

[||]Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138, U.S.A.

This research was supported in part by Research Foundation for Information Technology, University of Sydney; Department of Computer Science, Tel-Aviv University; IBM Fellowship; ONR contracts N00014-85-K-0168, N00014-91-J-1046, N00014-85-K-0445, and N00014-82-K-0154; by NSF grants CCR-8611442, CCR-8915206, IRI-9015570, DCR-8405478, CCR-8910289; and by DARPA contracts N00014-89-J-1988 and N00014-87-K-0825.

1 Introduction

1.1 Reliable Message Transmission Problem

To overcome the great engineering complexity involved in designing communication networks, designers typically organize them as a series of layers. Each layer is viewed as a "black box" that can be used by the next higher layer. Typical higher layers provide communication services with "nicer" properties than the lower layers upon which they are implemented. (See [Tan89, BG77] for more details.)

One of the most important functions of a higher layer is to provide reliable interprocess communication using a less reliable lower layer. For example, at the lower layers of a layered communication system, individual messages might be lost, duplicated, or corrupted, and sequences of messages might be delivered out of order; higher layers often mask these anomalies, guaranteeing to their users that the messages are delivered reliably, exactly once, and in the intended order.

In this paper, we consider a special case of this problem, namely, that of implementing a reliable, order-preserving communication layer between two processes in terms of an underlying unreliable layer that can lose and reorder (but not duplicate) messages. This problem arises, for example, at the network layer of a packet-switching network, as the problem of implementing a reliable transport layer in terms of an underlying sub-network transmission medium that permits message reordering. It also could arise at the lowest layers, as the problem of implementing a reliable data link layer in terms of an underlying physical transmission medium that permits message reordering.

We formulate the problem abstractly as follows. First, we give an abstract definition for the reliable communication layer. We assume that the reliable layer interacts with its environment (which we generally think of as a higher layer protocol) at two endpoints called *stations* or *sites*. The reliable layer accepts a sequence of data items that we call *messages* from the environment at one station and delivers them to the environment at the other station. All messages accepted at one end of the reliable layer are eventually delivered intact at the other end, and they are delivered in the order in which they are accepted. Thus, a reliable layer provides reliable two-way communication between two stations.

In more detail, from the point of view of the environment, the reliable layer supports two kinds of activities, which we call *actions*. A $\mathsf{send\text{-}rl}(m)$ action is performed by the environment at one station and causes the reliable layer to accept message m; $\mathsf{send\text{-}rl}(m)$ is an *input* action to the reliable layer. A $\mathsf{recv\text{-}rl}(m)$ action is performed by the reliable layer at the other station and causes message m to be delivered to the environment; $\mathsf{recv\text{-}rl}(m)$ is an *output* action of the reliable layer. Thus, the reliable layer looks to the environment like a black box with two kinds of actions: $\mathsf{send\text{-}rl}(m)$ and $\mathsf{recv\text{-}rl}(m)$, which take place at opposite stations.

Next, we define an abstract unreliable layer. As does the reliable layer, an unreliable layer interacts with its environment (usually a higher layer protocol) at two endpoints. It accepts sequences of data items from one station and delivers them to the other station. We call the data items transmitted by the unreliable layer *packets*, to distinguish them from the "messages" of the reliable layer.

Unlike the reliable layer, the unreliable layer may deliver packets incorrectly. It may lose (i.e., fail to deliver) a packet, and it may deliver packets out of order. (We do not consider an unreliable layer that corrupts or duplicates packets.) In order to exclude the uninteresting case in which the unreliable layer loses all or nearly all packets, we impose a minimal liveness requirement on the unreliable layer by assuming that any packet that is accepted by the unreliable layer infinitely many times is also delivered infinitely many times.

In more detail, the unreliable layer supports two actions. A **send-ul**(p) action is issued by the unreliable layer's environment at one station and causes the unreliable layer to accept packet p; **send-ul**(p) is an *input* action to the unreliable layer. A **recv-ul**(p) action is issued by the unreliable layer at the other station and causes packet p to be delivered to the environment; **recv-ul**(p) is an *output* action of the unreliable layer. Thus, the external interface of the unreliable layer consists of actions with very similar names and functions to those of the reliable layer. We use different names (e.g., **send-rl**(m) vs. **send-ul**(p)) in order to distinguish actions of the two layers.

We next define the notion of a *channel*. A channel is a device (i.e., a machine or program) that supports the actions of an unreliable layer and whose interactions with its environment have the properties specified for an unreliable layer; thus, a channel is an implementation of the unreliable layer. An *implementation* of the reliable layer consists of a pair of protocols to be executed by processes at the two stations which communicate via a channel. The protocols allow for the passage of messages between a process and the environment, and for the passage of packets between a process and channel as appropriate. The implementation is *correct* if the combination of the transmitter and receiver protocols with any channel has the properties of a reliable layer, that is, every message sent by the environment at one station is eventually received by the environment from the other station, and the order in which messages are received is the same as the order in which they are sent. We call the problem of implementing a reliable layer on an unreliable layer the *reliable message transmission problem (RMTP).*

In discussing solutions to RMTP, it is convenient to focus on the problem of implementing a *one-way* reliable layer, that is, a layer in which all of the **send-rl** actions occur at one station t, which we call the *transmitting* station, and all **recv-rl** actions occur at the other station r, which we call the *receiving* station. The processes at the transmitting and receiving stations are denoted by A^t and A^r, respectively, and are called *nodes.* An implementation of a full two-way reliable layer is easily obtained by combining two "copies" of the one-way reliable layer implementation.

1.2 Solutions to RMTP

RMTP is easily solved with a reliable channel—one that preserves packet order and eventually delivers each packet. If the packet alphabet is sufficiently large, the transmitter simply sends each message in turn as a separate packet on the channel. The receiver waits passively for each packet from the channel and delivers it to the environment when it arrives. In any case, if there are at least two packets, then the transmitter can encode each message in a variety of ways by a sequence of packets and send the packets one at a time across the channel. The receiver collects the packets that represent a message, decodes the message, and delivers it to the environment. Therefore, any solution to RMTP for a two-message alphabet can be used to solve RMTP for an arbitrary denumerable message alphabet.

Solutions to RMTP for certain kinds of unreliable channels date back to the early work on communication protocols (cf. [BSW69, Ste76, AUWY82]). Much of the early theoretical work was concerned with optimizing the number of states or number of packets under various assumptions about the channel. For example, [AUWY82] consider RMTP using *synchronous* channels in which the loss of a packet can be detected by the recipient at the next time step.

In this paper we consider RMTP in asynchronous systems where the channels are subject to arbitrary packet loss and reordering faults. We note that if the channel is subject to either one of these kinds of faults but not both, or if we allow for infinite channel alphabets, then there are easy solutions to the problem [Ste76, BSW69].

A channel that can both lose and reorder packets can cause serious problems. For example, if the transmitting station sends the sequence 1011210001 of one-digit packets, the receiving station might get 0011 or 1100 or 0000111112 or even nothing at all. It is not clear how the receiving station can derive any useful information from what it receives. In the presence of such channels, it is not hard to see that no protocol derived from the Unbounded Counter protocol simply by keeping sequence numbers to some finite modulus will work; the basic difficulty is that the processes can misinterpret old packets that are delivered out of order.

Indeed, it has often been conjectured informally (by practitioners in the communication network field) that solving RMTP with such channels and finite packet alphabets is impossible; we originally set out to prove this conjecture formally. Instead, we discovered a solution, which we present in Section 5.

Our solution is based on the Alternating Bit protocol [BSW69], augmented by a technique that prevents the processes from misinterpreting old packets. The main part of our solution involves implementing another kind of abstract communication layer that we call a *FIFO layer*, which can lose and duplicate but not reorder packets, using an unreliable layer that can lose and reorder but not duplicate. At first glance it is not clear that this represents progress toward a solution, since in return for the FIFO property that we gain, we introduce the possibility of duplication. However, the Alternating Bit Protocol can be used to achieve reliable communication using such a FIFO layer.

The main ideas of the construction of a FIFO layer are as follows. We say that a packet is "in transit" in an unreliable layer if it has been sent over the unreliable layer but not yet received at the other end. Since the unreliable layer can lose packets, packets in transit might never arrive, but since the unreliable layer can also reorder packets, a packet in transit might be delivered very late. Thus, there is no point in time at which the receiving process can determine that a given packet in transit will *never* be delivered in the future.

The basic strategy used by the protocol for dealing with packets in transit is for the receiving process to maintain a conservative estimate of the number of packets in transit toward it. Then if more copies of a particular packet are received after a certain point in time than are in transit at that time, the receiver knows that at least one of those copies must have been sent after that time. Our protocol uses two different kinds of packets: *queries* and *values*. A process only sends a value in response to a query; thus, the receiver knows that the number of values in transit toward it is at most the difference between the number of queries made and the number of values already received.

Although the protocol solves RMTP, there is a problem with its efficiency. Namely, after certain executions α, a large number of packets are required to convey each future message. We show, however, that for some function f, if the unreliable layer starts behaving "nicely" after α, then the number of packets required to convey each future message is bounded by $f(\alpha)$. The function f is called the *bound* of the protocol.

An obvious question is whether there exist solutions to RMTP which are bounded by a constant function k. Intuitively, such a protocol would be able to recover from any fault history and convey each future message using at most k packets. In Section 7, we show that no such constant-bounded protocol exists.

Results related to ours appear in several other papers. A preliminary version of the protocol is in [AFWZ89]. A preliminary version of the impossibility result appears in [LMF88]. Extensions of the protocol and impossibility result appear in [MS89, TL90, WZ89].

1.3 I/O Automata

Our results are presented using the Input/Output automaton formalism introduced in [LT87] and summarized in [LT89]. The formalism is used for several purposes. First, following the usual style for I/O automata, the allowed observable behavior of the reliable layer and those of the unreliable layer are specified in terms of sets of sequences of input and output actions. Second, the transmitter and receiver protocols of our algorithm in Section 5 are described formally as I/O automata. This makes unambiguous just how the scheduling of events is handled and how the protocols interact with the channels and their environment. Third, the space of possible devices from which channels may be chosen is taken to be the set of I/O automata. Finally, I/O automata are taken to be the space of possible solutions to RMTP for the purposes of the impossibility proof in Section 7. While there is no way of proving that I/O automata are sufficiently general to model any "reasonable" protocol, they have been shown to be adequate for modeling a large number of interesting and sophisticated protocols (e.g., [FLMW90, Blo87, WLL88, LG89]), and we believe that they are sufficiently powerful to model anything that could reasonably be considered a protocol (as well as some things that might not be considered reasonable, e.g., devices with non-computable steps).

The rest of the paper is organized as follows. Section 2 contains a summary of the needed definitions and facts about I/O automata. Section 3 defines the abstract notion of a communication layer and gives the properties of the reliable, FIFO, and unreliable layer with which we are concerned with in this paper. Section 4 defines what it means for a protocol to implement one layer on another. It then defines the Reliable Message Transmission Problem (RMTP) as the problem of implementing a reliable layer on an unreliable layer. Finally, it gives general theorems that allow the modular construction of a protocol that implements one layer on another. Section 5 gives constructions of a one-way reliable layer on a FIFO layer and of a one-way FIFO layer on an unreliable layer. These solutions are combined using general theorems to give a solution to RMTP. Section 6 contains our definition of boundedness or "recoverability" of a protocol. Section 7 shows that no constant bounded solution to RMTP with a finite packet alphabet exists. Section 8 discusses possible practical implications of these results.

2 The I/O Automaton Model

We use the Input/Output Automaton model to specify and describe system components. The I/O automaton model was first defined in [LT87], and we refer the reader to [LT87, LT89] for a complete development of the model. Here, we present a variant of the model which is adequate for our needs.

We assume a universal set of *actions* which describe the activities that occur during a computation. We refer to a particular occurrence of an action as an *event*. A finite or infinite sequence of actions is called a *behavior.* If α is any sequence and Π is any set of actions, then we let $\alpha|\Pi$ denote the subsequence of α consisting of all events in Π.

2.1 Specifications

We provide formal specifications for the allowable behavior of certain system components. Each specification describes the possible interactions of a component with its environment. In describing these interactions, it is helpful to classify the actions involving the component as "input" or "output" actions. Input actions originate in the environment and are imposed by the environment on the component, whereas output actions are generated by the component and imposed by the component on the environment. The interactions of the component with its environment are described by sequences over the component's actions, termed *behaviors*. Formally, each *specification* S is a triple $(in(S), out(S), beh(S))$, where

1. $in(S)$ and $out(S)$ are disjoint sets of actions;
2. $beh(S)$ is a set of behaviors over $in(S) \cup out(S)$.

The elements of $in(S)$ and $out(S)$ are the *input* and *output* actions of S, respectively; we denote their union by $acts(S)$. The set $beh(S)$ is the *behavior* of S, that is, the set of action sequences permitted by S.

For any sequence α and specification S, we write $\alpha|S$ as shorthand for $\alpha|acts(S)$. We say that S and S' are *independent* specifications if $acts(S) \cap acts(S') = \emptyset$.

2.2 I/O Automata

In order to model protocols, we use state machines called I/O automata. Like specifications, they have input and output actions, and they may also have *internal actions.* In addition, they have *states* and *steps* (i.e., *transitions*). Formally, an *I/O automaton A* (which we often call simply an *automaton*) is described by:

1. Three mutually disjoint sets of actions: $in(A)$, $out(A)$, and $internal(A)$. We denote $acts(A) = in(A) \cup out(A) \cup internal(A)$, $loc(A) = internal(A) \cup out(A)$, and $ext(A) = in(A) \cup out(A)$, i.e., $acts(A)$ is the set of A's actions, $loc(A)$ is the set of A's local actions, namely, the actions that A controls, and $ext(A)$ is the set of A's external actions.
2. A set $states(A)$ and a set $start(A) \subseteq states(A)$ of A's *start states.*
3. A transition relation, $steps(A) \subseteq states(A) \times acts(A) \times states(A)$, that is *input enabled,* i.e., for every input action π and state s, there exists some state s' such that $(s, \pi, s') \in steps(A)$. (In general, an action π is *enabled* from a state s if for some s', $(s, \pi, s') \in steps(A)$.)
4. A fairness condition, $fair(A)$, described as a partition on A's local actions with countably many equivalence classes.

An *execution* α of A is a (possibly infinite) sequence of the form:

$$s_0 \xrightarrow{\pi_1} s_1 \xrightarrow{\pi_2} \cdots$$

where s_0 is an initial state of A, and for every $i \geq 0$, $(s_i, \pi_{i+1}, s_{i+1})$ is a transition of A. If α is finite then it terminates in a state. The execution α is *fair* if one of the following holds:

1. α is finite and no local action is enabled from the final state of α.
2. α is infinite, and for every set of local actions $L \in fair(A)$, either actions from L are taken infinitely many times in α (i.e., for infinitely many i's, $\pi_i \in L$), or actions from L are disabled infinitely many times in α (i.e., for infinitely many i's, no action of L is enabled from s_i).

A fair execution should be thought of as giving "fair turns" to each class of $fair(A)$. Informally, one class of $fair(A)$ typically consists of all the actions that are controlled by a single component within the system modeled by automaton A, so fairness means giving each component regular opportunities to take a step under its control, if any is enabled.

The following proposition says that if A is an I/O automaton, then for every finite execution α of A there is a fair execution of A which has α as a prefix and for which the input actions occurring after α are exactly those in a prescribed sequence γ.

Proposition 2.1 *Let A be an I/O automaton and let γ be a sequence of input actions of A. Suppose α is a finite execution of A. Then there exists a fair execution α' of A such that α' is an extension of α and $\alpha'|ext(A) = (\alpha|ext(A))\gamma$.*

Proof: (Sketch) The basic idea is to construct the sequence inductively, interleaving transitions that involve successive input events from γ with transitions that involve locally controlled actions. The successive locally controlled transitions are obtained by dovetailing among the countably many partition classes. A detailed proof appears in [LS89]. ■

The *behavior* of an execution α of A (and more generally, of any sequence of actions and states of A), $beh(\alpha)$, is defined to be the sequence $\alpha|ext(A)$. A *fair behavior* of an automaton A is any sequence $beh(\alpha)$, where α is a fair execution of A. The set of all fair behaviors of A is denoted by $fairbeh(A)$.

2.3 Composition

Let A and B be I/O automata. We say that A and B are *composable* if the only mutual actions are input of one and output of the other, or input of both. If A and B are composable, their *composition*, written as $A \circ B$, is an automaton C such that:

1. C's output and internal actions are the union of the output and internal actions respectively of A and B, and C's input actions are the union of A's and B's input actions that are not C's output actions (i.e., $in(C) = in(A) \cup in(B) - (out(A) \cup out(B))$).
2. C's state set is the Cartesian product of its component state sets, i.e., $states(C) = states(A) \times states(B)$, and C's initial state set is the Cartesian product of its components' initial state sets.
3. C's transitions are such that only the components to which the action belongs are affected, i.e., $((s_A, s_B), \pi, (s'_A, s'_B)) \in steps(C)$ iff:

$$\begin{cases} (s_A, \pi, s'_A) \in steps(A) \text{ and } s_B = s'_B & \text{if } \pi \in acts(A) - acts(B), \\ s_A = s'_A \text{ and } (s_B, \pi, s'_B) \in steps(B) & \text{if } \pi \in acts(B) - acts(A), and \\ (s_A, \pi, s'_A) \in steps(A) \text{ and } (s_B, \pi, s'_B) \in steps(B) & \text{if } \pi \in acts(A) \cap acts(B). \end{cases}$$

4. The fairness condition of C, $fair(C) = fair(A) \cup fair(B)$. (In other words, actions are in the same class in C's partition exactly if they are in the same class in either A's or B's partition.) Note that this is a partition of $loc(C)$ since A and B do not have any locally controlled actions in common.

We say that A and B are *independent* if $acts(A) \cap acts(B) = \emptyset$. Independent A and B are always composable, and $in(A \circ B) = in(A) \cup in(B)$. (Thus, no input actions are "captured" in the composition.)

Let α be an execution of C. Then α defines an execution of A obtained by deleting from α every occurrence of $\xrightarrow{\pi} s$ for actions $\pi \notin acts(A)$, and replacing every remaining state in α with its A component. We denote the resulting execution by $A[\alpha]$. Similarly, α defines an execution of B denoted by $B[\alpha]$. Note that α is a fair execution of C iff $A[\alpha]$ and $B[\alpha]$ are fair executions of A and B respectively. The following propositions are proved in [LT87]. The first one establishes that composition is associative and commutative, modulo renaming of states of the resulting automata. The second shows how fair executions of A and B can be combined to yield a fair execution of C:

Proposition 2.2 *$A \circ (B \circ C) = (A \circ B) \circ C$ and $A \circ B = B \circ A$ modulo renaming of states.*

Proposition 2.3 *Let $C = A \circ B$. Let β be a sequence of actions in $ext(C)$, and suppose that α_1 and α_2 are fair executions of A and B respectively, such that $\beta|ext(A) = beh(\alpha_1)$ and $\beta|ext(B) = beh(\alpha_2)$. Then there is a a fair execution α of C such that $\beta = beh(\alpha)$, $A[\alpha] = \alpha_1$ and $B[\alpha] = \alpha_2$.*

3 Layered Communication Systems

We define several communication layers which form the specifications both for the reliable layer, which is to be achieved by the Reliable Message Transmission Problem, and for the underlying unreliable communication system upon which a solution is built. A third kind of layer, the FIFO layer, serves as an intermediary in our construction of a solution to RMTP.

3.1 Communication Layers

A *communication layer* is a particular kind of specification in which the input actions of the layer represent requests to send data and the output actions represent the receipt of data. Formally, a *communication layer* L consists of:

1. A specification $(in(L), out(L), beh(L))$.

2. A data item domain D_L.

3. A mapping $data_L: acts(L) \rightarrow D_L$, such that each element $d \in D_L$ is the image under $data_L$ of exactly one input action and exactly one output action.

4. An *orientation* function $direction_L: D_L \rightarrow \{tr, rt\}$.

The mapping $data_L$ indicates the data item sent or received by each action. Note that the condition given for $data_L$ above implies that $data_L|in(L)$ is a bijection from $in(L)$ to D_L and $data_L|out(L)$ is a bijection from $out(L)$ to D_L.

If $\pi \in in(L)$ is the unique input action such that $data_L(\pi) = d$, we may write $\mathsf{send}_L(d)$ to denote π, and we call π a **send**-action. Similarly, if $\pi \in out(L)$ is the unique output action such that $data_L(\pi) = d$, we may write $\mathsf{recv}_L(d)$ to denote π, and we call π a **recv**-action. We omit subscripts when no confusion will occur. This notation is elaborated later when several distinct layers come under simultaneous discussion.

The orientation function on the data item domain captures our notion of locality. We assume two sites: t—a fixed *transmitting site* and r—a fixed *receiving site*. The data items d such that $direction_L(d) = tr$ are those that travel from t to r. We will refer to this set as D_L^{tr}. Similarly the set D_L^{rt} consists of those where $direction_L(d) = rt$, that is, they travel from r to t. Thus we assume in our model that each data item moves in one direction only. We define

$$acts_L^t = \{\mathsf{send}_L(d) : direction_L(d) = tr\} \cup \{\mathsf{recv}_L(d) : direction_L(d) = rt\},$$

and

$$acts_L^r = \{\mathsf{send}_L(d) : direction_L(d) = rt\} \cup \{\mathsf{recv}_L(d) : direction_L(d) = tr\}.$$

The actions in $acts^t(L)$ and $acts^r(L)$ are the actions of the transmitting site and receiving site, respectively. The partition of $acts(L)$ into $acts^t(L)$ and $acts^r(L)$ induces corresponding partitions on subsets of actions, e.g., $in^t(L) = acts^t(L) \cap in(L)$ and $in^r(L) = acts^r(L) \cap in(L)$ form a partition of $in(L)$, etc.

A layer is diagrammed in Figure 1. The two boxes represent the sites t and r. The arrows represent actions. The wiggly line represents the network connection between the two sites.

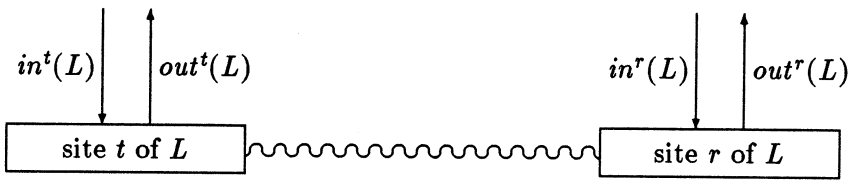

Figure 1: A Communication Layer.

A sequence $\beta \in beh(L)$ is called an *L-behavior.* We define an arbitrary sequence β to be *L-consistent* provided that $\beta|L$, the restriction of β to the actions of L, is an L-behavior.

3.2 L-channels

An I/O automaton A is called an *L-channel* if $in(L) \subseteq in(A)$, $out(L) \subseteq out(A)$, and $fairbeh(A)|L \subseteq beh(L)$. Thus an L-channel has all the actions required by the specification (and possibly more), and any fair behavior of an L-channel is L-consistent. However, an L-channel is not required to exhibit all possible behaviors permitted by L. A is called a *universal L-channel* if $in(A) = in(L)$, $out(A) = out(L)$, $internal(A) = \emptyset$, and $fairbeh(A)|L = beh(L)$.

3.3 One-way Layers

A layer L is said to be *one-way from t to r* if $in(L) = in^t(L)$ and $out(L) = out^r(L)$. Hence, in a one-way layer from t to r, **send** actions take place at the transmitting site and **recv** actions take place at the receiving site. A one-way layer in the reverse direction, from r to t, is similarly defined.

A layer L can be *decomposed* into two one-way layers. L^{tr} is the restriction of L to the t-to-r direction, and L^{rt} is the restriction of L to the r-to-t direction. L^{tr} is obtained by taking $in(L^{tr}) = in^t(L)$, $out(L^{tr}) = out^r(L)$, $beh(L^{tr}) = beh(L)|L^{tr}$, and $D_{L^{tr}} = D_L^{tr}$. The mapping $data_{L^{tr}}$ is the restriction of the mapping $data_L$ to the actions in $acts(L^{tr})$, and the orientation function $direction_{L^{tr}}$ is the restriction of $direction_L$ to the domain $D_{L^{tr}}$. Thus, $direction_{L^{tr}}(d) = tr$ for each $d \in D_{L^{tr}}$ as required for a one-way layer. The layer L^{rt} is similarly defined.

3.4 Properties of Communication Layers

Let L be a layer and let α be a sequence of actions over $acts(L)$. Let *cause* be a total function from **recv** events to **send** events of α. Intuitively, this mapping indicates the **send** event that "causes" each **recv** event. We define the following properties of the pair $(\alpha, cause)$:

(LC1) *[No prescience]* For each **recv** event π in α, the corresponding event $cause(\pi)$ occurs prior to π in α.

(LC2) *[No corruption]* For each **recv** event π in α, $data_L(cause(\pi)) = data_L(\pi)$.

(LC3) *[No duplication]* The *cause* mapping is one-to-one.

(LC4) *[No reordering]* If π and ϕ are **recv** events and $cause(\pi)$ precedes $cause(\phi)$ in α, then π precedes ϕ in α.

(LC5) *[Progress]* For each $d \in D_L$, if α contains infinitely many **send**(d) events, then α contains infinitely many **recv**(d) events.

(LC6) *[No losses]* The *cause* mapping is onto.

Note that (LC5) depends only on α. We shall use these conditions to describe the three different kinds of layers that we use in this paper: the reliable layer, the unreliable layer, and the FIFO layer.

3.5 Reliable Layers

The behaviors that are considered appropriate for a reliable layer are those in which every data item sent is eventually received exactly once. Data items are received in the same order as they are sent, and no item is received before it is sent. It follows that each data item that is sent infinitely many times is received infinitely many times.

Let L be a communication layer. A sequence α of actions over $acts(L)$ is said to be a *reliable layer sequence* if there exists a total function *cause* from **recv** events to **send** events of α such that the pair $(\alpha, cause)$ satisfies conditions (LC1)–(LC6). A layer L is *reliable* if $beh(L)$ is the set of *all* reliable layer sequences over $acts(L)$.

We often use the notation RL to refer to a reliable layer L. When discussing a reliable layer RL, we generally refer to data items as *messages*, and we denote the domain D_{RL} of messages simply by M. The input action π for which $data_{RL}(\pi) = m$ is denoted by send-rl(m), and the output action π for which $data_{RL}(\pi) = m$ is denoted by recv-rl(m).

Note that there are many reliable layers, differing in data item domains, orientation functions, and action alphabets. Moreover, for a given reliable layer RL, there are many possible RL-channels. Recall that a reliable layer RL is intended to specify all of the behavior sequences that could reasonably be described as reliably transferring data from the given domain in the direction given by the orientation function. An RL-channel is intended to model an actual implementation of a reliable layer and in general can exhibit only a subset of the behavior sequences allowed by the reliable layer. The requirement that an RL-channel be an I/O automaton rules out degenerate implementations which cannot handle all possible inputs.

3.6 Unreliable Layers

The behaviors that are considered appropriate for an unreliable layer are those in which every data item sent is received at most once, and no item is received before it is sent. Moreover, each data item that is sent infinitely many times is received infinitely many times.

Let L be a communication layer. A sequence α of actions over $acts(L)$ is said to be an *unreliable layer sequence* if there exists a total function *cause* from **recv** events to **send** events of α such that the pair $(\alpha, cause)$ satisfies conditions (LC1), (LC2), (LC3), and (LC5). A layer L is *unreliable* if $beh(L)$ is the set of *all* unreliable layer sequences over $acts(L)$.

We often use the notation UL to refer to an unreliable layer L. When discussing an unreliable layer UL, we generally refer to data items as *packets*, and we denote the domain D_{UL} of packets simply by P. The input (resp. output) action π for which $data_{UL}(\pi) = p$ is denoted by send-ul(p) (resp. recv-ul(p)).

Let α be a finite prefix of a UL-consistent sequence. Then the multiset[1] of data items received in α is a submultiset[2] of the multiset of data items sent in α. We say that a multiset Q over P is *in transit after* α if Q is a submultiset of the multiset of packets sent and not received in α. We say that Q is *in transit from t to r (resp. from r to t) after* α if Q is in transit after $\alpha | UL^{tr}$ (resp. after) $\alpha | UL^{rt}$). Note that if Q is in transit from t to r, then Q is a multiset of packets in P^{tr} and similarly for the reverse orientation.

We state two lemmas about the set of UL-behaviors for an unreliable layer UL with packet alphabet P. The proofs are obvious from the definitions and are omitted. The first lemma says that any sequence of packets in transit can be delivered at any time. The second lemma says that after any finite period of activity, an unreliable layer may act just like an unreliable layer starting from the initial state.

[1]A multiset is a collection of elements with multiplicities. Formally, a *multiset over a universe* U is a function $Q : U \rightarrow \mathbb{N}$. For every element $u \in U$, we define $mult(u, Q) = Q(u)$, which denotes the number of occurrences of u in Q.

[2]For two multisets Q and Q' over the same universe U, we say that Q' is a *submultiset* of Q, written $Q' \sqsubseteq Q$, if $mult(u, Q') \leq mult(u, Q)$ for every $u \in U$.

Lemma 3.1 *Let β be a finite UL-behavior, and let Q be a multiset of packets that is in transit after β. Let $\gamma = p_1, p_2, \ldots, p_k$ be a finite sequence of packets such that Q is the multiset defined by γ. Then β, $\mathsf{recv\text{-}ul}(p_1), \ldots, \mathsf{recv\text{-}ul}(p_k)$ is a finite UL-behavior.*

Lemma 3.2 *Let β be a finite UL-behavior and γ any UL-behavior. Then $\beta\gamma$ is a UL-behavior. Moreover, if Q is a multiset of packets that is in transit after β, then Q is in transit after $\beta\gamma$.*

3.7 FIFO layers

The behaviors that are considered appropriate for a FIFO layer are those in which (possibly multiple copies of) data items are received in the same order as they were sent, and no item is received before it is sent. Moreover, each data item that is sent infinitely many times is received infinitely many times. That is, a FIFO layer is a communication layer that can lose and duplicate messages, but not reorder them.

Let L be a communication layer. A sequence α of actions over $acts(L)$ is said to be a *FIFO layer sequence* if there exists a total function *cause* from **recv** events to **send** events of α such that the pair $(\alpha, cause)$ satisfies conditions (LC1), (LC2), (LC4), and (LC5). We say that L is a *FIFO layer* if $beh(L)$ is the set of *all* FIFO layer sequences over $acts(L)$.

We often use the notation FL to refer to a FIFO layer L. When discussing a FIFO layer FL, we generally refer to data items as *values*, and we denote the domain D_{FL} of values by V. The input (output) action π for which $data_{FL}(\pi) = p$ is denoted by $\mathsf{send\text{-}fl}(p)$ ($\mathsf{recv\text{-}fl}(p)$).

4 Implementation of Layers

Our goal is to implement a reliable layer on an unreliable layer. By an implementation, we mean protocols (expressed as I/O automata) for the transmitting and receiving stations that communicate, using the unreliable layer, to achieve the properties of a reliable layer. As with real-life protocols, we find it convenient to modularize the protocols by introducing an intermediate layer. We therefore need to formally define the notion of a multilayer implementation as well as what it means to implement one layer on another.

4.1 Definition of Layer Implementation

An implementation of one layer X on another layer Y consists of two I/O automata A^t and A^r, called *nodes*, which correspond to the two sites t and r. The two nodes must be independent; the only influence they can have on each other is through their interactions with the lower-level layer. Each action of the layers X and Y belongs to exactly one of the two nodes A^t and A^r, as indicated by the specification of layers X and Y. For example, if $\mathsf{send}(d)$ is in $in^t(X)$, and therefore $\mathsf{recv}(d)$ is in $out^r(X)$, then $\mathsf{send}(d)$ is in $in(A^t)$ and $\mathsf{recv}(d)$ is in $out(A^r)$. Also, if $\mathsf{send}(d)$ is in $in^t(Y)$, and therefore $\mathsf{recv}(d)$ is in $out^r(Y)$, then $\mathsf{send}(d)$ is in $out(A^t)$ and $\mathsf{recv}(d)$ is in $in(A^r)$.

More precisely, let X and Y be independent layers and let A^t and A^r be I/O automata.[3] Then we say that the pair (A^t, A^r) is *compatible with X on Y* if the following conditions are satisfied:

[3]Here and in the remainder of the paper, we assume without explicit mention that the internal action set of any automaton is disjoint from all other sets of actions under consideration.

1. $acts(A^t) \cap acts(A^r) = \emptyset$.
2. $in(A^t) \supseteq in^t(X) \cup out^t(Y)$ and $in(A^r) \supseteq in^r(X) \cup out^r(Y)$.
3. $out(A^t) \supseteq out^t(X) \cup in^t(Y)$ and $out(A^r) \supseteq out^r(X) \cup in^r(Y)$.

These definitions are illustrated in Figure 2.

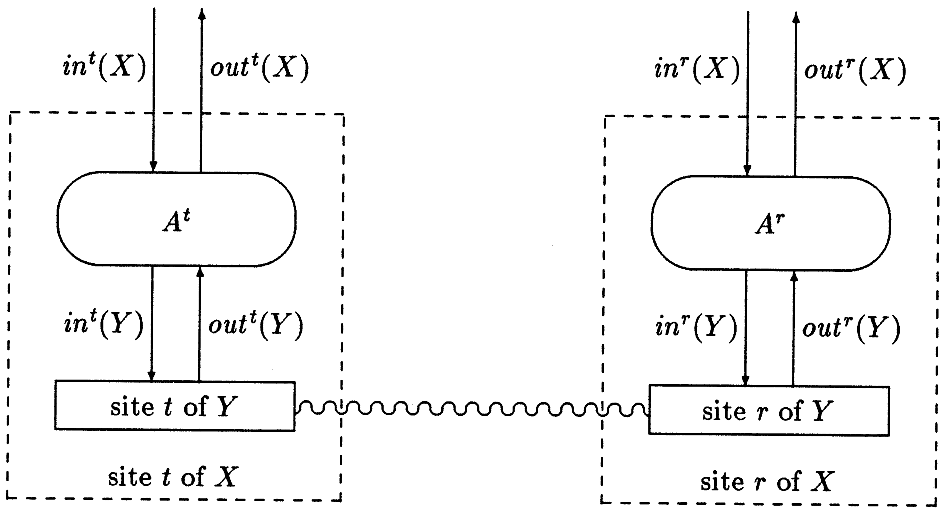

Figure 2: Implementation of Layer X on Layer Y.

The behavior of an implementation depends not only on A^t and A^r but also on the particular device that gives rise to the behavior of the lower level layer Y. In general, all we can assume about that behavior is that it is allowed by the specification of Y and that it is generated by some I/O automaton A_Y. The latter assumption ensures that the device has certain minimal behavior properties, for example, that it is always able to accept an input action to Y.[4] Once the behavior of the lower-level device is determined, the behavior of the implementation is just the behavior of the I/O automaton that results from the composition of A^t, A^r, and A_Y. That composition is itself an I/O automaton; hence, the implementation of the higher-level layer exhibits the same minimal behavior properties that were assumed for the lower-level layer. To ensure that the composition $A^t \circ A^r \circ A_Y$ is defined and that unanticipated capture of actions does not occur, we require that no actions of A_Y except those in $acts(Y)$ be shared with either A^t or A^r. We refer to this assumption by saying that "A_Y is without inappropriate actions".

More precisely, we say that the pair (A^t, A^r) *implements* X *on* Y if

1. (A^t, A^r) is compatible with X on Y;[5]
2. for every Y-channel A_Y without inappropriate actions, then $A^t \circ A^r \circ A_Y$ is an X-channel.

[4]There could be device models whose behavior cannot be modeled by I/O automata but whose behavior might nevertheless provide a suitable environment for implementing a higher-level layer. We exclude such devices from consideration and leave open the question of characterizing "reasonable" devices of greater generality than I/O automata.

[5]Note that this condition implies that X and Y are independent.

In other words, (A^t, A^r) "works" in any suitable environment to give the desired behavior of X, where a "suitable environment" is an I/O automaton A_Y whose behaviors are allowed by Y.

4.2 Multilayer Implementations

It is often convenient to implement one layer on another by means of intermediate layers. The following theorem justifies this approach.

Theorem 4.1 *Let X, Y, and Z be pairwise independent layers. Suppose the pair (A^t_{XY}, A^r_{XY}) implements X on Y, and (A^t_{YZ}, A^r_{YZ}) implements Y on Z. Then*

$$(A^t_{XY} \circ A^t_{YZ}, A^r_{XY} \circ A^r_{YZ})$$

implements X on Z.

Proof: Let A_Z be a Z-channel without inappropriate actions. Since (A^t_{YZ}, A^r_{YZ}) implements Y on Z, then $A_Y = A^t_{YZ} \circ A^r_{YZ} \circ A_Z$ is a Y-channel. Since (A^t_{XY}, A^r_{XY}) implements X on Y, then $A_X = A^t_{XY} \circ A^r_{XY} \circ A_Y$ is an X-channel. By Proposition 2.2,

$$\begin{aligned} A_X &= (A^t_{XY} \circ A^r_{XY}) \circ (A^t_{YZ} \circ A^r_{YZ} \circ A_Z) \\ &= (A^t_{XY} \circ A^t_{YZ}) \circ (A^r_{XY} \circ A^r_{YZ}) \circ A_Z. \end{aligned}$$

Hence, $(A^t_{XY} \circ A^t_{YZ}, A^r_{XY} \circ A^r_{YZ})$ implements X on Z. ■

4.3 Using One-Way Implementations

General layers are "two-way" in that data items can be sent in both directions between the two sites of the layers; indeed, two-way layers are needed to support most protocols. However, it is generally easier to implement a one-way layer and to use two "copies" of that implementation to implement a two-way layer.

We first need the notion of the "merge" of two layers. Formally, given independent layers Y_1 and Y_2, their *merge* is a single layer $Y = Y_1 \cup Y_2$, defined by $in(Y) = in(Y_1) \cup in(Y_2)$, $out(Y) = out(Y_1) \cup out(Y_2)$, and $beh(Y) = \{\alpha : \alpha|Y_1 \in beh(Y_1) \text{ and } \alpha|Y_2 \in beh(Y_2)\}$. Also, $D_Y = D_{Y_1} \cup D_{Y_2}$, $data_Y = data_{Y_1} \cup data_{Y_2}$, $direction_Y = direction_{Y_1} \cup direction_{Y_2}$. Thus, $acts^t(Y) = acts^t(Y_1) \cup acts^t(Y_2)$ and $acts^r(Y) = acts^r(Y_1) \cup acts^r(Y_2)$.

The following theorem describes a parallel composition of two independent layer implementations.

Theorem 4.2 *Let X_1, X_2, Y_1, and Y_2 be pairwise independent layers. Suppose (A^t_1, A^r_1) implements X_1 on Y_1 and (A^t_2, A^r_2) implements X_2 on Y_2. Then the pair $(A^t_1 \circ A^t_2, A^r_1 \circ A^r_2)$ implements $X_1 \cup X_2$ on $Y_1 \cup Y_2$.*

Proof: (Sketch) Immediate from the properties of composition of I/O automata. ■

Our main interest in Theorem 4.2 is to allow a layer X to be implemented by first decomposing X into its two one-way components X^{tr} and X^{rt}, implementing each separately on independent layers Y_1 and Y_2, respectively, and then combining the two implementations to yield an implementation of X on $Y_1 \cup Y_2$. The following corollary justifies this method.

Corollary 4.3 *Let X, Y_1, and Y_2 be pairwise independent layers. Suppose (A_1^t, A_1^r) implements X^{tr} on Y_1 and (A_2^t, A_2^r) implements X^{rt} on Y_2. Then the pair $(A_1^t \circ A_2^t, A_1^r \circ A_2^r)$ implements X on $Y_1 \cup Y_2$.*

Proof: Obvious from Theorem 4.2 and the fact that $X = X^{tr} \cup X^{rt}$. ■

4.4 A Characterization of Layer Implementation

The definition of implementing a layer X on a layer Y is somewhat difficult to work with, because the notion of "implements" involves universal quantification over all possible Y-channels. In case $beh(Y)$ is exactly the set of fair behaviors of some I/O automaton, then we can restate this definition directly in terms of Y rather than in terms of universal quantification over I/O automata.

Theorem 4.4 *Let A^t, A^r be I/O automata, and let X and Y be layers such that (A^t, A^r) is compatible with X on Y. Suppose that there exists a universal Y-channel. Then (A^t, A^r) implements X on Y iff every fair execution of $A^t \circ A^r$ that is Y-consistent is also X-consistent.*

Proof: Suppose that A^t, A^r, X, and Y satisfy the conditions of the theorem. Let $A = A^t \circ A^r$.

For the forward direction, let U be a universal Y-channel, and assume that (A^t, A^r) implements X on Y. Let α be a fair execution of A that is Y-consistent and let $\beta = beh(\alpha)$. Since β is Y-consistent, it follows that $\beta|Y \in beh(Y)$. Since U is universal, there is a fair execution α_1 of U such that $beh(\alpha_1) = \beta|Y$. Then Proposition 2.3 gives a fair execution α' of $A \circ U$ such that $\beta = beh(\alpha')$, $A[\alpha'] = \alpha$, and $U[\alpha'] = \alpha_1$. By assumption, since U is a Y-channel without inappropriate actions, then $A \circ U$ is an X-channel, so α' is X-consistent. But since $beh(\alpha) = beh(\alpha')$, it follows that $\alpha|X = \alpha'|X$, so that α is also X-consistent.

Conversely, suppose that every fair execution of A that is Y-consistent is also X-consistent. Let A_Y be any Y-channel without inappropriate actions. Let α be a fair execution of $A \circ A_Y$, and let $\alpha' = A[\alpha]$. Then α' is a fair execution of A. Since $\alpha|A_Y$ is a fair behavior of A_Y, it is Y-consistent. Thus $(\alpha|A_Y)|Y$ is a Y-behavior. However, since $acts(Y) \subseteq ext(A_Y)$, $(\alpha|A_Y)|Y = \alpha|Y$; thus $\alpha|Y$ is a Y-behavior. Since $\alpha'|Y = \alpha|Y$, it follows that α' is Y-consistent. Then the hypothesis of the theorem implies that α' is X-consistent. Since $\alpha|X = \alpha'|X$, α is also X-consistent, as needed. ■

The following lemma shows that there are universal channels for the reliable, unreliable, and FIFO layer specifications; hence Theorem 4.4 applies to these layers.

Lemma 4.5 *Let Y be a reliable, unreliable, or FIFO layer. Then there exists a universal Y-channel.*

Proof: Such an I/O automaton is not difficult to construct—for instance, it can choose nondeterministically, at the start of execution, exactly which submitted data items will be received, and in what order. Details are left to the reader. (Cf. [LMF88].) ■

4.5 The Reliable Message Transmission Problem

We finally define the reliable message transmission problem (RMTP) for a reliable layer RL and an unreliable layer UL to be the problem of finding a pair of I/O automata (A^t, A^r) that implements

RL on UL. Note that by our definitions, this problem will have a solution only if $acts(RL)$ and $acts(UL)$ are disjoint.

The following is an immediate consequence of Theorem 4.4 and Lemma 4.5. It says that the notion of implementability can be replaced by a condition on the fair executions of $A^t \circ A^r$ in defining RMTP.

Lemma 4.6 *Let (A^t, A^r) be a pair of I/O automata, let RL be a reliable layer, let UL be an unreliable layer, and assume that (A^t, A^r) is compatible with RL on UL. Then (A^t, A^r) implements RMTP for RL on UL iff every fair execution of $A^t \circ A^r$ that is UL-consistent is also RL-consistent.*

4.6 Properties of RMTP Solutions

Let (A^t, A^r) be an arbitrary solution to RMTP for RL and UL. We establish some properties of $A = A^t \circ A^r$.

Consider a system composed of A and an arbitrary UL-channel. The following lemma asserts that it is possible for the system to run from any point onward, with no further inputs, in such a way that no packets sent before that point are delivered after it. Note that the notion of "UL-consistent" enables the lemma to be stated without explicit reference to the underlying channel.

Lemma 4.7 *Let α be a finite execution of A that is UL-consistent. Then there exists a fair execution $\alpha\beta$ of A such that*

1. *β contains no* send-rl *events, and*
2. *β is UL-consistent.*

Proof: The proof is similar to that of Proposition 2.1. The main difference is that here, while dovetailing among the fairness classes of A, we ensure that whenever a **send-ul**(p) event is added to the execution, it is immediately followed by a corresponding **recv-ul**(p) event. This is allowed by A since **recv-ul**$(p) \in in(A)$, and UL-consistency is obviously maintained. The dovetail ensures that the execution $\alpha\beta$ constructed is a fair execution of A. Since every **send-ul** event is followed by its corresponding **recv-ul** event, it follows that the suffix β is UL-consistent. ■

Let α be a finite execution of A that is both RL- and UL-consistent, and let Q be a multiset of packets that is in transit from t to r after α. Consider any extension of α in which the multiset of packets received after α is a submultiset of Q. The following lemma shows that such an extension has no **recv-rl** events after α. Intuitively, this is true since otherwise the packets in Q could be delivered prior to any subsequent **send-rl** event so as to allow another **recv-rl** event to occur, which would violate RL-consistency.

Lemma 4.8 *Let α be a finite execution of A that is both RL- and UL-consistent, and let Q be a multiset of packets that is in transit from t to r after α. Consider a finite UL-consistent execution $\alpha\beta$ of A such that for some $m \in M^{tr}$, β contains an* recv-rl(m) *event. Then the multiset of P^{tr} packets received in β is not a submultiset of Q.*

Proof: Let α, Q, β, and m satisfy the conditions of the lemma. Assume, by way of contradiction, that the multiset of P^{tr} packets received in β is a submultiset of Q. Consider the sequence $\alpha\gamma$, where γ is constructed by deleting from β all steps involving actions in $acts(A^t)$, and changing the A^t components of the remaining states to be the A^t component of the last state in α.

We show now that $\alpha\gamma$ is a finite *UL*-consistent execution of A and that there exists a fair *UL*-consistent extension of $\alpha\gamma$ which is not *RL*-consistent, contradicting our assumption that A solves RMTP.

Since β is finite, so is γ. Moreover, from our construction it follows that $A^t[\alpha\gamma] = A^t[\alpha]$, which is an execution of A^t. It also follows that $A^r[\alpha\gamma] = A^r[(\alpha\beta)]$, which is an an execution of A^r. Consequently, $\alpha\gamma$ is an execution of $A^t \circ A^r$. The sequence of P^{tr} packets received in γ is exactly the sequence of P^{tr} packets received in β. By assumption, the multiset of P^{tr} packets received in β is a submultiset of Q. It therefore follows from Lemma 3.1 that $\alpha\gamma$ is UL^{tr}-consistent. Finally, the only events in $\gamma | UL^{rt}$ are **send-ul** events (since the corresponding **recv-ul** events of β were deleted). Since *UL* allows a finite number of packets to be lost, $\alpha\gamma$ is UL^{rt}-consistent. Consequently, $\alpha\gamma$ is a finite *UL*-consistent execution of A.

Let γ' be such that $\alpha\gamma\gamma'$ is a fair execution of A, γ' contains no **send-rl** events, and γ' is *UL*-consistent. The existence of γ' is guaranteed by Lemma 4.7. By Lemma 3.2, $\alpha\gamma\gamma'$ is *UL*-consistent. Since A solves RMTP, $\alpha\gamma\gamma'$ is *RL*-consistent. Thus, there is a function *cause* that satisfies (LC1)–(LC6). By property (LC1) and the fact that γ contains no **send-rl** events, *cause* must map each **recv-rl** event in γ to some **send-rl** event in α. But this is impossible by the pigeon hole principle, for *cause* is one-to-one by property (LC3), and α, being *RL*-consistent, contains the same number of **send-rl** and **recv-rl** events. This contradicts our assumption that A solves RMTP.

■

5 A Protocol With Finite Packet Alphabet

We construct a solution to RMTP, i.e., a pair of I/O automata that implements an arbitrary reliable layer *RL* with finite message alphabet on an unreliable layer *UL* with finite packet alphabet. The packet alphabet and actions of *UL* depend on the message alphabet and actions of *RL* in a way to be described. As in [AG88], we find it convenient to present our solution in a modular fashion.

Our solution is obtained from two basic constructions, one that implements an arbitrary *one-way reliable layer* on a suitable *two-way* FIFO layer, and one that implements an arbitrary *one-way* FIFO layer on a suitable *two-way* unreliable layer. These constructions are used twice each in forming the solution to RMTP.

Let *RL* be a given reliable layer. We decompose *RL* into its two one-way components, RL^{tr} and RL^{rt}. We use the first construction twice, once to implement RL^{tr} on FIFO layer FL_1 and once to implement RL^{rt} on FIFO layer FL_2. We may assume without loss of generality that FL_1 and FL_2 are independent and that the automata implementing RL^{tr} on FL_1 are independent of the automata implementing RL^{rt} on FL_2.[6] Then Corollary 4.3 gives an implementation (A^t_{RL}, A^r_{RL}) of *RL* on $FL = FL_1 \cup FL_2$.

Next, decompose *FL* into one-way layers FL^{tr} and FL^{rt}. Use the second construction to implement them on unreliable layers UL_1 and UL_2, respectively. As before, we may assume without loss of generality that UL_1 and UL_2 are independent and that the automata implementing FL^{tr} on UL_1

[6]This follows from the fact that RL^{tr} and RL^{rt} are independent.

are independent of the automata implementing FL^{rt} on UL_2. Corollary 4.3 gives an implementation (A^t_{FL}, A^r_{FL}) of FL on $UL = UL_1 \cup UL_2$.

Thus, we have constructed implementations of RL on FL and of FL on UL. Moreover, we can assume without loss of generality that A^t_{RL} is composable with A^t_{FL} and that A^r_{RL} is composable with A^r_{FL}. By Theorem 4.1, the pair $(A^t_{RL} \circ A^t_{FL}, A^r_{RL} \circ A^r_{FL})$ is an implementation of RL on UL, as desired.

5.1 Implementation of a Reliable Layer

Our first construction implements a one-way reliable layer on a FIFO layer. This is accomplished by a version of of the Alternating Bit Protocol [BSW69], expressed as a pair (B^t, B^r) of I/O automata that implements a one-way RL on FL.

In this protocol, the transmitter keeps sending each message to the receiver until it gets an acknowledgement from the receiver for that message. It then begins sending the next message. Meanwhile, the receiver keeps acknowledging the previous message until the current message is received, at which point it begins acknowledging it. A single bit header suffices for distinguishing consecutive messages, and likewise a single bit acknowledgement suffices to distinguish acknowledgements for consecutive messages.

The transmitter has two local variables, *queue*, which holds a queue of messages, initially empty, and a Boolean *flag*, initially equal to 1. The *queue* is used to buffer the messages to be sent by the reliable layer, and the *flag* is used to record the Boolean value being used as the header for the current message (i.e., the message that is first on *queue*). Likewise, the receiver has two local variables, *queue*, which holds a queue of messages, initially empty, and a Boolean *flag*, initially equal to 0. The *queue* is used to buffer the messages received from the transmitter until they are output to the environment via recv-rl events, and the *flag* is used to record the Boolean value that has been attached to the latest message received from the transmitter (if the *queue* is nonempty, then this is the last message on *queue*).

If the transmitter queue is nonempty, the transmitter is constantly enabled to send copies of the first message on its queue, tagged with a header consisting of the transmitter's flag; likewise, the receiver is constantly enabled to send acknowledgements, in the form of the receiver's flag. If the receiver receives a message with a header unequal to the receiver's flag, the receiver accepts the new message and changes its flag; otherwise, the receiver ignores the message. Analogously, if the transmitter receives an acknowledgement equal to its own flag, the transmitter accepts the acknowledgement as an acknowledgement to its current message, goes on the next message and changes its flag.

The code B^t and B^r is given in Figure 3. The fairness condition has three classes: one for all the send-fl(m, b) actions, one for all the send-fl(b) actions, and one for all the recv-rl actions.

Standard arguments about the Alternating Bit Protocol (see, for example, [HZ87]) can be used to show the following correctness theorem.

Theorem 5.1 *Let RL be a one-way reliable layer with message set M. Let $V_1 = \{(m, b) : b \in \{0, 1\}$ and $m \in M\}$ and $V_2 = \{0, 1\}$. Let FL be a FIFO layer with value set $V = V_1 \cup V_2$, where $\{$send-fl$(m, b) : (m, b) \in V_1\} \cup \{$recv-fl$(b) : v \in V_2\} \subseteq acts^t(FL)$ and $\{$recv-fl$(m, b) : (m, b) \in V_1\} \cup \{$send-fl$(b) : v \in V_2\} \subseteq acts^r(FL)$. Then every fair execution of $B^t \circ B^r$ that is FL-consistent is also RL-consistent.*

```
Transmitter $B^t$

Variables:
    queue, a finite queue of elements of $M$,
        initially empty
    flag, a Boolean, initially 1

send-rl($m$), $m \in M$:
    effect:
        add $m$ to queue

send-fl($m, b$) $m \in M$, $b$ a Boolean:
    precondition:
        $m$ is first on queue
        $b = flag$

recv-fl($b$), $b$ a Boolean:
    effect:
        if $b = flag$ then
            remove first element from queue
            $flag := 1 - flag$
```

```
Receiver $B^r$

Variables:
    queue, a finite queue of elements of $M$,
        initially empty
    flag, a Boolean, initially 0

recv-rl($m$), $m \in M$:
    precondition:
        $m$ is first on queue
    effect:
        remove first element from queue

send-fl($b$), $b$ a Boolean:
    precondition:
        $b = flag$

recv-fl($m, b$), $m \in M$, $b$ a Boolean:
    effect:
        if $b \neq flag$ then
            add $m$ to queue
            $flag := 1 - flag$
```

Figure 3: Protocol B: An implementation of the reliable layer using FIFO links.

It follows immediately from Theorems 4.4 and 5.2 that the pair (B^t, B^r) implements RL on FL.

5.2 Implementation of a FIFO layer

We implement a one-way FIFO layer FL on an unreliable layer UL with appropriate packet alphabet. The implementation consists of a pair of I/O automata (F^t, F^r) that are compatible with FL on UL.

The transmitter F^t maintains a local variable, *latest* which is set to v whenever **send-fl**(v) occurs; the receiver continuously tries to obtain new values of *latest* from the transmitter. After any prefix of an execution, we define as *new* any value that has been in *latest* at any time since the last **recv-fl** event. The algorithm ensures that the value that is contained in any **recv-fl** event new relative to the execution prefix ending just prior to the **recv-fl** event. The receiver, F^r, continues to perform **recv-fl** events.

Consider an execution α of $F^t \circ F^r$ that is UL-consistent. Because the unreliable layer can lose, reorder, and delay packets, the receiver needs a way to recognize new values. Suppose that α contains at least one **recv-fl** event. Let α' be the prefix of α that ends with the last **recv-fl** event, and let $old(\alpha)$ be the maximum size of any multiset of values that is in transit from t to r after

α'. If for some value v, the number of **recv-ul**(v) actions in α after α' is more then than $old(\alpha)$, then, since the unreliable layer cannot duplicate packets), at least one **send-ul**(v) action must have occurred after α'. Therefore, v must be a new value.

The receiver cannot compute $old(\alpha)$ exactly; it instead maintains an upper bound on this value. The protocol is designed so that the transmitter only sends packets in response to *queries* by the receiver. Thus, the difference between the number of **send-ul**$(query)$ events and the number of **recv-ul**(v) events at any time is an upper bound on the number of packets in transit from t to r at that time. The receiver keeps track of this difference in its local variable *intransit*, which is copied into another local variable *old* whenever a **recv-fl** occurs. Thus, *old* is always an upper bound on the number of values that were in transit from t to r at the time of the most recent **recv-fl** event.

In somewhat more detail, the protocol proceeds as follows. The receiver continuously sends queries to the transmitter. For each query received, the transmitter sends the value of *latest*. The receiver records the number of copies of each value v received in a local variable $count[v]$. At any point in the execution, if $count[v] > old$ then **recv-fl**$^{n}(v)$ is enabled; as argued above, v is guaranteed to be a new value in this case. Whenever a **recv-fl** occurs, *old* is set to the current value of *intransit* and the counters are reset to zero, thus ensuring that the next **recv-fl** event also contains a new value (with respect to the current **recv-fl** event).

The code for protocol *FIFO* is given in Figure 4. The fairness partition contains three classes: one for the **send-ul**(v) actions, one for the **send-ul**$(query)$ actions, and one for all the **recv-fl** actions.

The arguments above now allow us to claim the following correctness result for this implementation.

Theorem 5.2 *Let FL be a one-way FIFO layer with value set V. Let UL be an unreliable layer with packet alphabet $P = V \cup \{query\}$, where $\{\textsf{send-ul}(v) \mid v \in V\} \cup \{\textsf{recv-ul}(query)\} \subseteq acts^{t}(UL)$ and $\{\textsf{recv-ul}(v) \mid v \in V\} \cup \{\textsf{send-ul}(query)\} \subseteq acts^{r}(UL)$. Let α be a fair execution of protocol FIFO that is UL-consistent. Then α is FL-consistent.*

It follows immediately from Theorems 4.4 and 5.2 that protocol *FIFO* implements *FL* on *UL*.

5.3 A Solution to RMTP

As discussed in the introduction to this section, we can combine the above constructions to yield a pair of I/O automata (A^t, A^r) that implements *RL* over a layer *UL*, thereby solving RMTP. The solution actually consists of two copies of protocol (B^t, B^r), one in each direction, and two copies of protocol (F^t, F^r), one in each direction.

Let M^{tr} and M^{rt} be the message alphabets in each direction of *RL*. The first copy of (B^t, B^r) implements RL^{tr} over FIFO layer FL_1, and the second copy (in which the roles of t and r are reversed) implements RL^{rt} over FIFO layer FL_2. FL_1 has value set $V_1 = V_1^{tr} \cup V_1^{rt}$, where $V_1^{tr} = M^{tr} \times \{0,1\}$ and $V_1^{rt} = \{0,1\}$. The second copy has value set $V_2 = V_2^{rt} \cup V_2^{tr}$, where $V_2^{rt} = M^{rt} \times \{0,1\}$ and $V_2^{tr} = \{0', 1'\}$. (Primes have been used to make the sets disjoint.) The merged FIFO layer, $FL = FL_1 \cup FL_2$ thus has value set $V = V^{tr} \cup V^{rt}$, where $V^{tr} = (M^{tr} \times \{0,1\}) \cup \{0', 1'\}$ and $V^{rt} = (M^{rt} \times \{0,1\}) \cup \{0,1\}$.

Each direction of *FL* is implemented by a copy of (F^t, F^r). The first copy requires an unreliable layer UL_1 with packet alphabet $P_1 = P_1^{tr} \cup P_1^{rt}$, where $P_1^{tr} = V^{tr}$ and $P_1^{rt} = \{query\}$. The second copy (again, with the roles of t and r reversed) requires an unreliable layer UL_2 with

Transmitter F^t

```
Variables:
    latest, an element of $V \cup \{nil\}$,
        initially nil
    pending, a nonnegative integer,
        initially 0

send-fl($v$):
    effect:
        $latest := v$

recv-ul($query$):
    effect:
        $pending := pending + 1$

send-ul($v$):
    precondition:
        $pending > 0$
        $v = latest \neq nil$
    effect:
        $pending := pending - 1$
```

Receiver F^r

```
Variables:
    intransit, a nonnegative integer,
        initially 0
    old, a nonnegative integer, initially 0
    for each $v \in V$, $count[v]$,
        a nonnegative integer, initially 0

recv-fl($v$):
    precondition:
        $count[v] > old$
    effect:
        $count[w] := 0$ for all $w$
        $old := intransit$

send-ul($query$):
    effect:
        $intransit := intransit + 1$

recv-ul($v$):
    effect:
        $intransit := intransit - 1$
        $count[v] := count[v] + 1$
```

Figure 4: Protocol F: An implementation of FL over UL.

packet alphabet $P_2 = P_2^{rt} \cup P_2^{tr}$, where $P_2^{rt} = V^{rt}$ and $P_2^{tr} = \{query'\}$. The merged unreliable layer, $UL = UL_1 \cup UL_2$, has packet alphabet $P = P^{tr} \cup P^{rt}$, where $P^{tr} = V^{tr} \cup \{query'\} = (M^{tr} \times \{0,1\}) \cup \{0', 1', query'\}$ and $P^{rt} = V^{rt} \cup \{query\} = (M^{rt} \times \{0,1\}) \cup \{0, 1, query\}$. Thus, $P = (M \times \{0,1\}) \cup \{0, 1, query, 0', 1', query'\}$, so $|P| = 2|M| + 6$.

In conclusion, we have the following result.

Theorem 5.3 *Let RL be a reliable layer with message set M. Let UL be an unreliable layer with packet alphabet* $P = P^{tr} \cup P^{rt}$, *where* $P^{tr} = (M^{tr} \times \{0,1\}) \cup \{0', 1', query'\}$ *and* $P^{rt} = (M^{rt} \times \{0,1\}) \cup \{0, 1, query\}$, *and where* $\{\text{send-ul}(p) : p \in P^{tr}\} \cup \{\text{recv-ul}(p') : p' \in P^{rt}\} \subseteq acts^t(UL)$ *and* $\{\text{send-ul}(p') : p' \in P^{rt}\} \cup \{\text{recv-ul}(p) : p \in P^{tr}\} \subseteq acts^r(UL)$. *Then there is a protocol that implements RL over UL.*

It is possible to reduce the size of P by using a variant of the Alternating Bit protocol. The $2|M|$ term in the size of the packet alphabet arises in the Alternating Bit protocol because the protocol attaches an extra bit to each message in order to distinguish the current and previous messages. A more efficient encoding accomplishes the same end with only a single additional message for each of the two directions. This yields a solution using only $|M| + 8$ packet types.

6 Boundedness

In Section 5, we presented a solution to RMTP. The solution suffers from a serious performance problem: the more packets there are in transit, the more packets are needed in order to convey a message. Every lost packet appears forever to be in transit. Hence, as more and more packets are lost, the protocol runs slower and slower.

A natural questions is, "Can one do better?" To answer it, we formally define a complexity measure that allows us to capture the intuitive notion of the rate at which messages are processed by the protocol, i.e., the number of packets that the transmitter must send in order for the receiver to output the next message to the environment. Since the unreliable layer is permitted to lose and reorder packets, there is no worst-case upper bound on the rate. We instead measure performance according to the best case, i.e., the minimum number of packets that will suffice, assuming the unreliable layer performs in the best way possible.

6.1 f-Boundedness

Fix a pair of automata (A^t, A^r) that implements layer X on layer Y, and let $A = A^t \circ A^r$. We measure the performance of A in terms of its "f-boundedness", where f is a function mapping finite executions of A and positive integers to positive integers.

f-boundedness is an attempt to measure the ability of a protocol to recover from faults. An f-bounded protocol has the property that, after any finite faulty execution α, if the layer Y "co-operates" in the future, then the number of packets sent in order to convey the i^{th} subsequent X-message is at most $f(\alpha, i)$.

Formally, we say that A is *f-bounded* if for every finite X- and Y-consistent execution α of A and every sequence μ of send_X events, there exists some extension $\alpha\beta$ of α such that the following all hold:

1. $\alpha\beta$ is a fair execution of A.
2. β is Y-consistent.
3. $\beta|\{\mathsf{send}_X(d) : d \in D_X\} = \mu$,
4. Let β be divided into segments $\beta = \beta_1 \ldots \beta_{|\mu|}\gamma$ such that each β_i ends with a recv_X event. Then, for every $i \leq |\mu|$, there are at most $f(\alpha, i)$ send_Y events in β_i.

In the above definition, α represents the initial execution during which layer Y may be arbitrarily faulty. β represents a "good" execution that might occur after layer Y has recovered from its "faulty" behavior. We require that a good β exist for each possible sequence μ, since recovery should work for all possible future inputs. The Y-consistency of β prevents the protocol's recovery from being dependent on the receipt of an old Y-packet.

If A is f bounded for $f(\alpha, i) = k$, we say that A is *k-bounded*; if A is k-bounded for some constant k, then we say that A is *constant-bounded.*

6.2 Analysis of the RMTP Solution

In Section 5.3, we presented a solution to RMTP. We now show that the solution is f-bounded, where f depends only on α.

Since the solution is composed of copies of B and copies of F, we analyze B and F separately. Recall that B implements a one-way reliable layer on a FIFO layer, and F implements a one-way FIFO layer on an unreliable layer.

At any point in B's execution, in order to receive the next message, the receiver needs only to receive a single **send-fl** value (of the appropriate parity) from the transmitter. If the two sites' *flag* values are currently equal, then the transmitter will be able to send this value after it has received one new acknowledgement from the receiver. On the other hand, if the *flag* values are currently unequal, then the new value can be sent immediately. Thus, if the underlying FIFO layer cooperates in delivering values as soon as they are sent, and if the scheduler cooperates in giving transmitter and receiver turns as needed, then only 2 packets are needed to deliver a new message. Hence, B is 2-bounded.

At any point in F's execution, in order to receive each subsequent value, the receiver needs to receive *intransit* copies of the current packet. These copies can be sent if *intransit* queries are received. Consequently, F is bounded by f where $f(\alpha, i)$ is twice the value of *intransit* at the end of α. F is bounded by a non-constant function which depends only on α.

It is easy to see that once several copies of B and F are composed, the resulting protocol is bounded by a non-constant function which depends only on α. In fact, it is f-bounded for a function f such that $f(\alpha, i)$ is linear in $|\alpha|$. We would much prefer a constant-bounded solution. We show in Section 7 that no such solution exists.

7 An Impossibility Proof

We now show that there is no constant-bounded solution to RMTP that uses a finite packet alphabet. We first define a strict partial order $<_k^U$ on multisets over a universe U, where k is a positive integer. When U is finite, $<_k^U$ has no infinite increasing chains. We then show that every k-bounded solution to RMTP has an infinite sequence of *UL*- and *RL*-consistent executions such that the multiset of packets in transit at the end of each is increasing in the ordering $<_k^P$. This implies that P is infinite.

Let k be a positive integer and U a set. For every multiset Q over U, let Q^k be the multiset defined by $mult(Q^k, u) = \min(k, mult(Q, u))$ for every $u \in U$. For multisets Q_1 and Q_2 over U, define $Q_1 <_k^U Q_2$ if $Q_1^k \sqsubseteq Q_2^k$ and $Q_1^k \neq Q_2^k$. That is, $Q_1 <_k^U Q_2$ if for every $u \in U$, $\min(k, mult(u, Q_1)) \leq \min(k, mult(u, Q_2))$, and there exists a u for which equality does not hold. Note that $Q <_k^U Q'$ and $Q' \sqsubseteq Q''$ imply $Q <_k^U Q''$. Note also that if U is finite, then every increasing chain of $<_k^U$ has at most $k|U| + 1$ elements. We omit explicit mention of U when it is clear from context.

Let (A^t, A^r) be a solution to RMTP for which $M^{tr} \neq \emptyset$. For every *UL*-consistent execution α of A, let Q_α denote the maximum multiset of P^{tr} packets that is in transit at the end of α.

Lemma 7.1 *Let (A^t, A^r) be a k-bounded solution to RMTP for which $M^{tr} \neq \emptyset$, and let $A = A^t \circ A^r$. Then, for every UL- and RL-consistent finite execution α of A, there exists a UL- and RL-consistent* *finite execution α' of A, such that $Q_\alpha <_k Q_{\alpha'}$.*

Proof: Let α be a *UL*- and *RL*-consistent finite execution of A. Let $m \in M^{tr}$. From the definition of k-boundedness, it follows that there exists some extension $\alpha\beta$ of α such that the following all hold:

1. $\alpha\beta$ is a fair execution of A.
2. β is *UL*-consistent.
3. $\beta|\{\textsf{send-rl}(m') : m' \in M^{tr}\} = \textsf{send-rl}(m)$,
4. There are at most k **send-ul** events in β_1, where $\beta = \beta_1\gamma$ and β_1 ends with a **recv-rl** event.

Let R denote the multiset of P^{tr} packets received in β_1. From 2 and 4 it follows that $mult(p, R) \leq k$ for every $p \in P$. From Lemma 4.8 it follows that R is not a submultiset of Q_α. Hence, for some $p \in P^{tr}$, $mult(p, R) > mult(p, Q_\alpha)$. From 2 it follows that β_1 is *UL*-consistent. Since β_1 contains a **recv-ul**(p) event, there is some **send-ul**(p) event in β_1. Let β_1' be the shortest prefix of β_1 such that $\alpha\beta_1'$ is a finite execution (i.e., ends with a state) and the last event of β_1' is **send-ul**(p). Since *UL*-consistent sequences are closed under prefix and, by Lemma 3.2, concatenation, $\alpha\beta_1'$ is *UL*-consistent. From 2, it follows that packets in transit in α are not received in β, so $Q_\alpha \sqcup \{p\} \sqsubseteq Q_{\alpha\beta_1'}$.[7] Since $Q_\alpha <_k Q_\alpha \sqcup \{p\} \sqsubseteq Q_{\alpha\beta_1'}$, it follows that $Q_\alpha <_k Q_{\alpha\beta_1'}$.

If $\beta_1'|RL$ is empty, then $\alpha' = \alpha\beta_1'$ is *RL*-consistent and we are done. Else, $\beta_1'|RL = \textsf{send-rl}(m)$. From Lemma 4.7 it follows that there exists a fair execution $\alpha\beta_1'\beta_2$ of A such that β_2 contains no **send-rl** events and β_2 is *UL*-consistent. Since $\alpha\beta_1'\beta_2$ is fair, it is *RL*-consistent. Because α is *RL*-consistent and $\beta_1'|RL = \textsf{send-rl}(m)$, we deduce $\beta_2|RL = \textsf{recv-rl}(m)$. Let β_2' be some prefix of β_2 that includes the **recv-rl**(m) event and such that $\alpha' = \alpha\beta_1'\beta_2'$ is a finite execution of A. α' is a finite *UL*- and *RL*-consistent execution of A, and $Q_\alpha <_k Q_{\alpha'}$. ■

Lemma 7.2 *Let A be a k-bounded solution to RMTP such that $M^{tr} \neq \emptyset$. Then there exists an infinite sequence $\alpha_0, \alpha_1, \ldots$ of finite UL- and RL-consistent executions of A, such that for every $i \geq 0$, $Q_{\alpha_i} <_k Q_{\alpha_{i+1}}$.*

Proof: The claim follows immediately from Lemma 7.1 by induction, when we define α_0 to be the empty execution, which is trivially *RL*- and *UL*-consistent. ■

We can therefore conclude:

Theorem 7.3 *Let A be a k-bounded solution to RMTP such that $M^{tr} \neq \emptyset$ (resp. $M^{rt} \neq \emptyset$). Then P^{tr} (resp. P^{rt}) is infinite.*

Proof: From Lemma 7.2, it follows that A defines an infinite increasing chain of multisets of packets according to $<_k^{P^{tr}}$. This implies that P^{tr} is infinite. ■

Corollary 7.4 *There is no constant-bounded solution to RMTP that uses finite packet alphabets.*

[7] $Q \sqcup \{p\}$ is the multiset Q' such that $mult(p, Q') = mult(p, Q) + 1$ and $mult(u, Q') = mult(u, Q)$ for all $u \neq p$.

Notes

Theorem 7.3 can be extended in various technical ways by weakening some of the assumptions of the theorem that are not needed in the proof.

1. The sequences α_i constructed in the proof of Lemma 7.2 have alternating sends and receives, i.e., every sequence $\alpha_i = \alpha_i^1, \ldots, \alpha_i^n$, where every $\alpha_i^j|RL$ is of the form $\mathsf{send\text{-}rl}(m)\mathsf{recv\text{-}rl}(m)$. Thus, f-boundedness need only hold for alternating α's.

2. In the proof of Lemma 7.1, we only look at extensions of one message, as opposed to the general μ. Thus, we can weaken the definition of f-boundedness by requiring the conditions to hold only for μ of length 1.

3. The proof only considers packets going in one direction, namely, a direction for which the set of messages is non-empty. We can therefore weaken the definition of f-boundness by restricting attention to that one direction.

8 Conclusion

In this paper we have considered the problem of reliable communication over unreliable channels. We have presented both an algorithm and an impossibility result. On the one hand we have demonstrated that, seemingly contrary to popular belief, there exists a correct protocol that uses only finite packet alphabets. On the other hand, we have demonstrated that any such protocol must exhibit serious degradation of performance, as more and more messages are lost and delayed. This raises the question of whether *practical* finite-alphabet protocols can exist for channels that can lose and reorder packets. The answer to this questions probably lies in the interpretation of the term "practical".

If "practical" means maintaining a bandwidth similar to the underlying channels, then the performance of our protocol is horrendous. Moreover, this is not simply a shortcoming of our protocol, but, as our impossibility result shows, it is an inherent limitation. The impossibility result says that any finite-alphabet protocol must require a large number of packets to send each message; this imposes a large penalty on the bandwidth of the channel. Later theoretical work has strengthened the claim that communicating with bounded headers over a channel that can reorder packets must incur a severe bandwidth penalty. The interested reader is referred to [MS89, TL90, WZ89] where a variety of impossibility results related to ours are shown.

On the other hand, the development of newer, extremely high bandwidth, communication channels raises the serious possibility that a communication protocol could be considered reasonably efficient even though it reduces the bandwidth of the underlying channel. Even then, our impossibility result shows that no *fixed* reduction in bandwidth can be maintained; rather, the reduction must worsen over time.

As usual, it is necessary to be cautious in making practical inferences from the theoretical results, for the theoretical results are based on a set of assumptions that might be weakened in practice. For example, we have assumed that the protocols must be *asynchronous*; however, simple and efficient protocols can be constructed that use information about real time, in the form of local processor clocks and bounds on the lifetime of packets (e.g., [SD78]). Also, we have assumed that the protocols must always work correctly; however, efficient randomized protocols can be constructed that allow a small fixed probability of error (e.g., [HGM89]). A challenging problem is to find models that are realistic, yet are simple enough to admit theoretical analysis.

Acknowledgments

We would like to thank Baruch Awerbuch and Ewan Tempero for useful discussions on this work. We also would like to thank Jennifer Welch for her helpful comments on early drafts of our paper. Special thanks are also due to the designers of the (usually) reliable Internet, over which many of our conversations about this research were held.

References

[AFWZ89] H. Attiya, M. J. Fischer, D. Wang, and L. D. Zuck. Reliable communication using unreliable channels. Manuscript, 1989.

[AG88] Y. Afek and E. Gafni. End-to-end communication in unreliable networks. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 117–130, 1988.

[AUWY82] A. V. Aho, J. D. Ullman, A. D. Wyner, and M. Yannakakis. Bounds on the size and transmission rate of communication protocols. *Comp. & Maths. with Appls.*, 8(3):205–214, 1982. This is a later version of [AUY79].

[AUY79] A. V. Aho, J. D. Ullman, and M. Yannakakis. Modeling communication protocols by automata. In *Proc. 20th IEEE Symp. on Foundations of Computer Science*, pages 267–273, 1979.

[BG77] G. V. Bochmann and J. Gecsei. A unified method for the specification and verification of protocols. In B. Gilchrist, editor, *Information Processing 77*, pages 229–234. North-Holland Publishing Co., 1977.

[Blo87] B. Bloom. Constructing two-writer atomic registers. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 249–259, Vancouver, British Columbia, Canada, August 1987. Also, to appear in special issue of *IEEE Transactions On Computers on Parallel and Distributed Algorithms.*

[BSW69] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, May 1969.

[FLMW90] A. Fekete, N. Lynch, M. Merritt, and W. Weihl. Commutativity-based locking for nested transactions. *Journal of Computer and System Science*, 41(1):65–156, August 1990.

[HGM89] A. Herzberg, O. Goldreich, and Y. Mansour. Source to destination communication in the presence of faults. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 85–102, 1989.

[HZ87] J. Y. Halpern and L. D. Zuck. A little knowledge goes a long way: Simple knowledge-based derivations and correctness proofs for a family of protocols. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 269–280, 1987. Journal version to appear in J. ACM.

[LG89] N. Lynch and K. Goldman. Distributed algorithms. Research Seminar Series MIT/LCS/RSS–5, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1989.

[LMF88] N. A. Lynch, Y. Mansour, and A. Fekete. Data link layer: Two impossibility results. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 149–170, August 1988.

[LS89] N. Lynch and E. Stark. A proof of the Kahn principle for Input/Output automata. *Information and Computation*, 82(1):81–92, July 1989.

[LT87] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 137–151, August 1987.

[LT89] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[MS89] Y. Mansour and B. Schieber. The intractability of bounded protocols for non-FIFO channels. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 59–72, August 1989.

[SD78] C. A. Sunshine and Y. K. Dalal. Connection management in transport protocols. *Computer Networks*, 2:454–473, 1978.

[Ste76] M. V. Stenning. A data transfer protocol. *Computer Networks*, 1:99–110, 1976.

[Tan89] A. Tannenbaum. *Computer Networks.* Prentice Hall, 1989.

[TL90] Ewan Tempero and Richard E. Ladner. Tight bounds for weakly bounded protocols. In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, pages 205–218, August 1990.

[WLL88] J. Welch, L. Lamport, and N. Lynch. A lattice-structured proof of a minimum spanning tree algorithm. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 28–43, August 1988.

[WZ89] Da-Wei Wang and Lenore D. Zuck. Tight bounds for the sequence transmission problem. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 73–83, August 1989.