A LITTLE KNOWLEDGE GOES A LONG WAY: SIMPLE KNOWLEDGE-BASED DERIVATIONS AND CORRECTNESS PROOFS FOR A FAMILY OF PROTOCOLS

Joseph Y. Halpern and Lenore D. Zuck

YALEU/DCS/TR-571
October, 1987

A Little Knowledge Goes a Long Way: Simple knowledge-based derivations and correctness proofs for a family of protocols

Joseph Y. Halpern
IBM Almaden Research Center
halpern@almvma (BITNET); halpern@ibm.com (ARPA/CSNET)

Lenore D. Zuck*
Department of Computer Science, Yale University
zuck@yalecs (BITNET); zuck@yale (ARPA)

Abstract: We use a high-level, knowledge-based approach for deriving a family of protocols for the *sequence transmission* problem. The protocols of Aho, Ullman, and Yannakakis [AUY79,AUWY82], the Alternating Bit protocol [BSW69], and Stenning's protocol [Ste76] are all instances of one of the knowledge-based protocols that we derive. Our derivation leads to easy and uniform correctness proofs for all these protocols.

*This author was supported in part by NSF grant DCR-8405478

1 Introduction

Designing and proving the correctness of protocols in distributed systems is a notoriously difficult problem. The potential for faulty behavior makes the problem even more difficult. Subtle bugs are often found in seemingly correct protocols (see, for example, [GS80,SH86]). Consequently, researchers have looked for good tools to analyze distributed systems. Temporal logic [OL82,Pnu77], the state machine approach [BG77,BS80,Mer76,Sun79], Floyd-Hoare-style methods [HO83], model checking [CES83], and interval logic [SM82], have all been advocated, and indeed, have been used successfully to verify a number of distributed protocols.

While the proofs in the cited papers do indeed demonstrate correctness, they do not usually help the reader to understand *why* the protocol is correct. Reading the step-by-step details of these proofs, one loses the global picture of what is happening in the protocol. It is not obvious which of the protocol's features are important and what is the role of each of the steps in the protocol. This understanding is crucial if we want to redesign the protocol so that it still works correctly in a slightly different environment or if we want to optimize the protocol in some way. Ideally, the design and verification of a protocol would be closely related and one could straightforwardly derive correctness proofs from the design methodology. In practice, design and verification are often done separately, in fact by different groups of people.

Recently it was suggested [HM84] that a useful way to analyze distributed systems is in terms of knowledge and how communication changes the processes' state of knowledge. Since then the role of knowledge in distributed systems has been extensively studied [CM86,DM86,FI86,FHV86, Had87,HF85,HV86,LR86,MT86,NT87,PR85] (see [Hal87] for an overview). In this paper, we use reasoning about knowledge to design and verify a family of protocols that deal with a standard problem of data communication that we call the *sequence transmission* problem. We believe that our analysis provides further evidence of the usefulness of a knowledge-based approach.

The problem is easily stated: Consider two processes, called the *sender* and the *receiver*. The sender S has an input tape with an infinite sequence $X = \langle x_0, x_1, \ldots \rangle$ of data elements. S reads these data elements and tries to transmit them to the receiver R. R must write these data elements onto an output tape. We require that (a) at any time the sequence of data elements written by R is a prefix of X (this is the *safety* property) and (b) if the communication medium satisfies appropriate *fairness* conditions, then every data element x_i in the sequence X is eventually written by R (this is the *liveness* property).

The sequence transmission problem clearly has a trivial solution if we assume that messages sent by S cannot be lost, corrupted, duplicated, or reordered. S simply sends $x_0, x_1, \ldots$ in order, and R writes them out as it receives them. However, once we consider a faulty communication medium, the problem becomes far more complicated. A number of different communication models have been extensively studied in the literature. For example, [AUWY82,AUY79] decribe protocols that solve the problem in the case of a completely synchronous communication channel which allows only one-bit messages (we call this communication model the *AUY model*). They consider various types of faulty behavior, including message deletion and corruption. The famous *Alternating Bit protocol* [BSW69] is a solution to the sequence transmission problem for an asynchronous channel where messages cannot be reordered or duplicated, but may be

lost or (detectably) corrupted. *Stenning's protocol* [Ste76] deals with the problem in the case where messages may be duplicated, lost, (detectably) corrupted, or reordered.

The solutions to the sequence transmission problem that appear in the literature were all designed individually, on an *ad hoc* basis. As we show here, they can all be viewed as instances of the same high-level *knowledge-based* protocol. (A knowledge-based protocol is one with explicit tests for knowledge; cf. [HF85].) By considering these protocols from the viewpoint of knowledge, we obtain simple, transparent derivations and correctness proofs for them all. We urge the reader to compare our correctness proofs to those that can be found, for example, in [BG77,Gou85,Hai82,Hai85,HO83].

It is interesting that the idea of thinking about such protocols in terms of knowledge appears in an informal way quite early in the literature. For example, in [BG77] it says that "Verification ... will correspond ... to finding out whether and in which circumstances the sender subsystem (and its user) can 'know' that all data obtained from the user have been received correctly and in sequence to the user in the receiver subsystem." However, knowledge is not explicitly used to verify correctness in any of the papers cited above.

The rest of the paper is organized as follows: In the next section we describe our formal model of distributed systems. We also discuss formally the notion of knowledge in distributed systems and knowledge-based protocols. In Section 3 we simultaneously derive a high-level knowledge-based protocol for the sequence transmission model and a finite state protocol which is an implementation of the knowledge-based protocol. In Section 4 we show how to implement the finite-state solution in the AUY model by appropriately encoding messages. In Section 5 we derive other high-level protocols for the sequence transmission problem, and then obtain corresponding implementations of these protocols in the AUY model. The solutions we obtain here are essentially the same as those actually presented in [AUWY82,AUY79]. Correctness proofs for all the protocols are almost immediate from our derivation. (The formal details, however, are somewhat tedious and left to the appendix.) In Section 6 we extend our results to the Alternating Bit protocol and Stenning's protocol. In Section 7 we consider necessary and sufficient conditions on the state of knowledge required for the sequence transmission problem, as was done in [HM84] for coordinated attack, [DM86,MT86] for Byzantine agreement, and [Had87] for atomic commitment. We conclude in Section 8 with further discussion of the knowledge-based viewpoint and some directions for further research.

2 The formal model

A detailed description of the model we use can be found in [HF87], so we only sketch the necessary details here, and refer the reader to [HF87] for further motivation and examples.

We assume that our processes are state machines and that the relevant features of a system at a given time are described by the *global state* of the system, where a global state is a tuple describing the local state of each of the processes and the state of the *environment*. We take the environment to consist of everything in the system that is relevant to the analysis that is not part of the state of the processes. (Exactly what is relevant will of course depend on the particular system being analyzed.) For simplicity, we assume that time ranges over the

non-negative integers. (The definitions can easily be extended to other time models.) A *run* (or *execution*) of the system is defined to be a function from the non-negative integers to global states. Intuitively, a run is a description of the relevant features of the system over time. We occasionally refer to a pair (r, m) consisting of a run r and a time m as a *point*. As has been done in numerous previous papers (e.g. [CM86,HF85,HM84,PR85]), we identify a distributed system with a set $\mathcal{R}$ of runs. We say (r, m) is a point in system $\mathcal{R}$ if $r \in \mathcal{R}$.

For protocols solving the sequence transmission problem, the processes are S and R. Thus a global state is a tuple of the form (s_e, s_S, s_R), where s_e is the environment state, s_S is S's local state, and s_R is R's local state. The details of the states will depend on how we choose to analyze the system; we will discuss this in more detail when we formally analyze the protocols presented in the sequel.

We define a *protocol for process j* to be a (possibly nondeterministic or probablistic) function from j's local states to *actions*. Thus a process' protocol describes what actions the process takes as a function of its local state. Usually we think of these actions as coming from a small set of basic actions, such as reading a data element, writing a value, sending a message, or receiving a message. We find it useful to think of the environment as also running a protocol. In the introduction we discussed assumptions on the communication model such as "messages cannot be reordered or duplicated, but may be lost or (detectably) corrupted." Such assumptions, which implicitly describe the environment's behavior, can be captured by the environment's protocol. We take a *joint protocol* P to consist of protocols P_e, P_S, P_R for e, S, and R respectively. (We remark that for all the cases we consider in this paper, both P_S and P_R are deterministic, while P_e is nondeterministic or probabilistic.)

We would like to associate with every (joint) protocol a particular set of runs. To do this, we first must specify the possible local states for each of e, S, and R. Call these sets of states L_e, L_S, and L_R respectively. Let $\mathcal{G} = L_e \times L_S \times L_R$ be the set of possible global states. (Not all the global states in $\mathcal{G}$ will necessarily be reachable when we run the protocol.) The next step is to specify the subset $\mathcal{G}_0$ of $\mathcal{G}$ which consists of the possible initial global states. Finally, we must specify how the actions performed by e, S, and R change the global state. Let Act_e, Act_S, and Act_R be the actions performed in P_e, P_S, and P_R respectively. A *transition function* τ associates with every *joint action* $(a_e, a_S, a_R) \in Act_e \times Act_S \times Act_R$ a global state transformer $\tau(a_e, a_S, a_R)$, i.e., a deterministic function from $\mathcal{G}$ to $\mathcal{G}$. (We could allow $\tau(a_e, a_S, a_R)$ to be nondeterministic, but we do not need this level of generality in this paper.) Thus we can think of $\tau(a_e, a_S, a_R)$ as describing the effect of simultaneously having e perform action a_e, S perform a_S, and R perform a_R.

Fix $\mathcal{G}$, $\mathcal{G}_0$ ($\mathcal{G}_0 \subseteq \mathcal{G}$), and τ as above. We say that a run r is *consistent with protocol P (with respect to $\mathcal{G}$, $\mathcal{G}_0$, and τ)* if

1. $r(0) \in \mathcal{G}_0$ (so $r(0)$ is a legal initial state).
2. For all $m \geq 0$, if $r(m) = (s_e, s_S, s_R)$, then there is a joint action $(a_e, a_S, a_R) \in P_e(s_e) \times P_S(s_S) \times P_R(s_R)$ such that $r(m+1) = \tau(a_e, a_S, a_R)(r(m))$ (so $r(m+1)$ is the result of transforming $r(m)$ by a joint action that could have been performed from $r(m)$ according to P).

We use $\mathcal{R}(P)$ to denote the set of all runs consistent with P (with respect to $\mathcal{G}$, $\mathcal{G}_0$, and τ).

Besides the type of protocol defined above (which we occasionaly call a *standard* protocol), we will also be interested in a more high-level notion called a *knowledge-based protocol* ([HF85]), where we allow explicit test for knowledge. To define such protocols formally, we find it convinient to assume that there is some set Φ of *basic facts* about the system. The set Φ can include facts of the type "$x_6 = 0$", "R sent m to S", etc. We define an *interpreted system* $\mathcal{I}$ to be a pair $(\mathcal{R}, \pi)$ consisting of a system $\mathcal{R}$ and an assignment π of truth values to the basic facts for each point in $\mathcal{R}$, so that for every $p \in \Phi$ and point (r, m) in $\mathcal{R}$, we have $\pi(r, m)(p) \in \{\textbf{true}, \textbf{false}\}$. We say that the point (r, m) is in the interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$ if $r \in \mathcal{R}$.

Given an interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$ and a point (r, m) in $\mathcal{I}$, we define a satisfiability relation $\models$ between the tuple $(\mathcal{I}, r, m)$ and a formula φ. For a basic fact $p \in \Phi$, we have

$$(\mathcal{I}, r, m) \models p \text{ iff } \pi(r, m)(p) = \textbf{true}.$$

We extend the $\models$ relation to conjunctions and negations in the obvious way:

$$\begin{array}{rcl} (\mathcal{I}, r, m) \models \neg\varphi & \text{iff} & (\mathcal{I}, r, m) \not\models \varphi \\ (\mathcal{I}, r, m) \models \varphi \wedge \psi & \text{iff} & (\mathcal{I}, r, m) \models \varphi \text{ and } (\mathcal{I}, r, m) \models \psi. \end{array}$$

We want to extend our language to allow formulas of the form $K_j\varphi$, which is read "process j *knows* φ". We ascribe knowledge to processes in a distributed system using ideas first discussed in [HM84], and later amplified in numerous other papers (see [Hal87] for an overview and references). Again, we state our definitions under the assumption that the only processes in the system are S and R, although they clearly extend to the case with an arbitrary (but fixed) set of processes.

Given two global states $s = (s_e, s_S, s_R)$ and $s' = (s'_e, s'_S, s'_R)$, we say s and s' are *indistinguishable to process* j (where j is either S or R) if j has the same state in both s and s', i.e., if $s_j = s'_j$. We say two points (r, m) and (r', m') are *indistinguishable to* j, and write $(r, m) \sim_j (r', m')$, if the global states $r(m)$ and $r'(m')$ are indistinguishable to j. We then define

$$(\mathcal{I}, r, m) \models K_j\varphi \quad \text{iff} \quad (\mathcal{I}, r', m') \models \varphi \text{ for all } r' \text{ and } m' \text{ such that } (r, m) \sim_j (r', m').$$

This definition is designed to capture the intuition that processor j *knows* φ at $r(m)$ if φ is true at time m' in run r' for all points (r', m') indistinguishable to j from (r, m).

An important property of this definition of knowledge is that $K_j\varphi \supset \varphi$. Thus, if $(\mathcal{I}, r, m) \models K_j\varphi$, then $(\mathcal{I}, r, m) \models \varphi$; this easily follows from the observation that $(r, m) \sim_j (r, m)$.

A knowledge-based protocol allows explicit tests for knowledge. Unlike the tests that appear in a standard protocol, the truth value of the test in a conditional statement of the form "if $K_S\varphi$ **then** ..." cannot be determined by looking at the local state in isolation. Its truth depends on the truth of φ at other points (all the ones with global states that S cannot distinguish from its current global state). Thus, whereas a protocol for S is a function from S's local states to actions, a knowledge-based protocol is a function from S's local states *and an interpreted system* to actions. For example, consider a protocol P_S with an instruction of the form "if $K_S\varphi$ **then** send m **else** send m'". Suppose that in state s_S process S is at the step in the protocol

with this instruction. The action performed by S in state s_S is "send m" if S knows φ, and "send m'" otherwise. Thus we have

$$P_S(s_S, I) = \begin{cases} \text{send } m & \text{if } (I, r, m) \models \varphi \text{ for all points } (r, m) \text{ where} \\ & \qquad\qquad S\text{'s state in } r(m) \text{ is } s_S \\ \text{send } m' & \text{otherwise.} \end{cases}$$

Note that the only difference between the formal definition of knowledge-based protocols and standard protocols is that a knowledge-based protocol takes an interpreted system as one of its arguments. Of course, a standard protocol can be viewed as a special case of a knowledge-based protocol where the function is independent of the interpreted system.

Fix a set $\mathcal{G}$ of global states, a subset $\mathcal{G}_0 \subseteq \mathcal{G}$ of initial states, a transition function τ on G, and an interpreted system I with global states in $\mathcal{G}$. We define a run r to be *consistent with (knowledge-based) protocol P relative to I* (with respect to $\mathcal{G}$, $\mathcal{G}_0$, and τ) just as we defined the notion of a run being consistent with a standard protocol P, except that now the joint action (a_e, a_S, a_R) in clause 2 is in $P_e(s_e, I) \times P_S(s_S, I) \times P_R(s_R, I)$ rather than $P_e(s_e) \times P_S(s_S) \times P_R(s_R)$.

An interpreted system $I = (\mathcal{R}, \pi)$ is *consistent* with a knowledge-based protocol P (with respect to $\mathcal{G}$, $\mathcal{G}_0$, and τ) if every run $r \in \mathcal{R}$ is consistent with P relative to I. In general, there is not a unique interpreted system that is consistent with a given protocol.

3 A knowledge-based protocol and a finite-state implementation

In this section we design a knowledge-based protocol that solves the sequence transmission problem and simultaneously derive a finite-state protocol that solves the problem in a situation where we allow messages to be deleted, (detectably) corrupted, duplicated, but disallow reordering of messages.

We start by trying to informally derive a solution. Suppose that the input sequence X is $\langle 0, 0, 1, \ldots \rangle$. (For simplicity we assume throughout the paper that the input sequence consists only of 0s and 1s, although in fact we could deal with any finite language.) Intuitively, S should start by sending 0 (the first bit) to R. Clearly, sending the first bit only once is not enough, since R might not get the message. So how long should S continue to send this message? One approach is for S to continue sending until S knows that R knows the value of x_0, which we write $K_S K_R(x_0)$ (we use $K_R(x_i)$ as an abbreviation for $K_R(x_i = 0) \vee K_R(x_i = 1)$). We can implement this by having R send S an *ack* message when it gets x_0. When S gets the *ack* message, $K_S K_R(x_0)$ holds. Can S safely send R the data element x_1 when it gets R's *ack*? Unfortunately, it cannot. Since x_1 is also 0, if S sends x_1 and R receives it, R will not know whether this is yet another attempt on the part of S to send x_0 (because S did not receive R's acknowledgement), or an attempt by S to send x_1. S must somehow let R know that $K_S K_R(x_0)$ holds. We can implement this by having S acknowledge R's acknowledgement, i.e., send *ack-ack* (which we abbreviate to ack^2) to R. When R receives this message, it can stop sending the *ack* message; moreover, it knows that the next data element it receives will be x_1, not x_0. S must

continue sending ack^2 until it knows that R knows that S knows that R knows x_0, i.e., until $K_S K_R K_S K_R(x_0)$ holds. We implement this here by having R send an ack^3 message so as to acknowledge the receipt of the ack^2 message. When S receives the ack^3 message, S can safely send x_1 to R, since S knows that R will not confuse x_1 with x_0.

Below we describe a knowledge-based protocol A that formalizes the intuitions above. In A, S sends the i^{th} data element until $K_S K_R(x_i)$ holds, then sends "$K_S K_R(x_i)$" until $K_S K_R K_S K_R(x_i)$ holds, and then reads the next data element and repeats the cycle. We also describe a standard protocol A^{fs} that implements A (and solves the sequence transmission problem) provided we restrict the environment's actions appropriately, as we discuss below. We present these two protocols side-by-side to emphasize their relationship. In particular, the reader should note that the ack, ack^2, and ack^3 messages of protocol A^{fs} are simply a way of guaranteeing that the appropriate level of knowledge is attained.

In the appendix, we give formal semantics for these protocols in terms of the formal model described in the previous section. All the theorems stated in this section are also proved in the appendix. We make a few remarks now that suffice for us to state the results and (hopefully) allow the reader to follow the informal presentation and get a high-level picture of what is going on.

The protocols all start with an INIT statement that describes the initial settings of all relevant variables. We assume that S has a local variable y for storing the last data element read. Thus the effect of "read y" (in INIT or line S1 of the figure below) is that S reads the next bit in the input sequence and stores it in the variable y. Similarly, we assume that S and R have local variables, z and z' respectively, for storing the message last received. R does not send anything until it receives the first value from S; we make this explicit in line R1 with the command "send λ" (where we use λ to denote that nothing is sent). In a command such as 'send "$K_S K_R(x_i)$" ' on line S1 of A, we assume that "$K_S K_R(x_i)$" is just some string of symbols (we allow arbitrarily long strings to be transmitted in one time step of the knowledge-based protocol), and that these strings are distinct for every value of i, as well as being distinct from strings of the form "$x_i = y$". Of course, the intention is that when R receives a message such as "$K_S K_R(x_i)$", then in fact R will know that S knows that R knows the value of x_i. Our semantics assumptions will indeed force this to be the case (cf. Lemma 3 in the Appendix).

As we shall show, the knowledge-based protocol A is correct in a wide variety of settings, precisely because it abstracts away the details of how a state of knowledge such as $K_S K_R(x_1)$ is attained. In particular, it is correct even if

1. we allow messages to be duplicated, reordered, and detectably corrupted as well as being deleted.
2. we have an asynchronous system, where S and R perform an action only when they are scheduled (rather than performing an action at every round). Of course, in order to assure the liveness property, we must assume that S and R are scheduled infinitely often.
3. there is some *a priori* knowledge about the sequence X. For example, we might have a situation where every even-numbered data element in X is (commonly) known to be 0. In this case, when running protocol A, R would be able to write x_0 even before it received any messages from S (since the state of knowledge $K_R(x_0)$ would hold at line R1).

S's protocol:

INIT. read y; $i := 0$; $z := \lambda$
S1. send "$x_i = y$"; receive z;
if $K_S K_R(x_i)$
then go to S2 **else** go to S1
S2. send "$K_S K_R(x_i)$"; receive z;
if $K_S K_R K_S K_R(x_i)$
then read y; $i := i + 1$; go to S1
else go to S2

R's protocol:

INIT. $i' := 0$; $z' := \lambda$
R1. send λ; receive z';
if $K_R(x_{i'})$
then write $x_{i'}$; go to R2
else go to R1
R2. send "$K_R(x_{i'})$"; receive z';
if $K_R K_S K_R(x_{i'})$
then go to R3 **else** go to R2
R3. send "$K_R K_S K_R(x_{i'})$"; receive z';
if $K_R(x_{i'+1})$
then $i' := i' + 1$; write $x_{i'}$; go to R2
else go to R3

Protocol A

S's protocol:

INIT. read y; $z := \lambda$
S1. send y; receive z;
if $z = ack$
then go to S2 **else** go to S1
S2. send ack^2; receive z;
if $z = ack^3$
then read y; go to S1
else go to S2

R's protocol:

INIT. $z' := \lambda$
R1. send λ; receive z';
if $(z' = 0 \vee z' = 1)$
then write z'; go to R2
else go to R1
R2. send ack; receive z';
if $z' = ack^2$
then go to R3 **else** go to R2
R3. send ack^3; receive z';
if $(z' = 0 \vee z' = 1)$
then write z'; go to R2
else go to R3

Protocol A^{fs}

To prove the correctness of A we require that the processes do not "forget" their message histories. This is captured in the formal model by taking S's and R's local states to encode all the messages they have sent and received thus far. (See the appendix for details.) To understand the need for no forgetting, suppose R moves from R2 to R3 because $K_R K_S K_R(x_{i'})$ holds. Moreover, suppose this happens because it gets a message of the form "$K_S K_R(x_{i'})$" from S. R will continue sending the message "$K_R K_S K_R(x_{i'})$" until it knows $x_{i'+1}$. But if it "forgets" that it received the message "$K_S K_R(x_{i'})$" from S, it might no longer be the case that $K_R K_S K_R(x_{i'})$ actually holds.

There is one more small technical condition we must require for the correctness of protocol A. We want to restrict our attention to interpreted systems $I = (\mathcal{R}, \pi)$ where the truth assignment π is such that the formula $x_i = 0$ (resp. $x_i = 1$) is true exactly at the points where x_i really does have the value 0 (resp. 1). To make this precise, we restrict attention to systems $\mathcal{R}$ where the environment component at every point has the form $(X, \ldots)$, where $X = \langle x_0, x_1, \ldots \rangle$ is the sequence of data elements, and the sequence X is constant throughout the run. For the purposes of this theorem, we will call an interpreted system $I = (\mathcal{R}, \pi)$ *reasonable* if the environment component does indeed have this form at all points, and $\pi((r, m))(x_i = v) =$ **true** iff the value

of the i^{th} component of X is v. These issues are made precise and discussed in more detail in the appendix, where the following theorem is proved:

Theorem 3.1: *Let I be a reasonable interpreted system where (1) messages may be deleted, reordered, duplicated, or detectably corrupted, and (2) j's local state contains a complete record of the messages sent and received by j (for $j \in \{S, R\}$). If I is consistent with protocol A, then every run of I has the safety property, and those runs of I where S and R are scheduled infinitely often and every message sent by S (resp. R) infinitely often is eventually delivered uncorrupted (when R (resp. S) is scheduled) also have the liveness property.*

Protocol A is an infinite state protocol. This is immediate from condition 2 of Theorem 1, where we assume that process j's state contains the whole message history. As we mentioned in the discussion above, this assumption is necessary for the correctness of the protocol. In practice, however, one does not want to store so much information. If S and R just keep track of the number of data elements they have read (resp. written) thus far, an analogue of protocol A will still work. In this case, we would have to replace the messages "$K_R(x_i)$", "$K_S K_R(x_i)$", and "$K_R K_S K_R(x_i)$" by "R wrote x_i", "$K_S(R$ wrote $x_i)$", and "$K_R K_S(R$ wrote $x_i)$". Similarly, the tests $K_S K_R(x_i)$, $K_R K_S K_R(x_i)$, and $K_S K_R K_S K_R(x_i)$ have to be replaced by $K_S(R$ wrote $x_i)$, etc. In the appendix we discuss how to modify the proof of correctness of protocol A to deal with this variant.

However, even if S and R only record the number of data elements they have read (resp. written) in their local state, they would still requires an unbounded amount of memory. Unbounded memory is not an artifact of our solution; it is a necessary requirement for sufficiently general solutions. If we allow messages to be deleted and reordered (and there is no upper bound on message delivery time for messages that do get delivered), then it is not hard to show that *any* solution to the sequence transmission problem must involve infinitely many distinct states. Similarly, it can be shown that if messages can be duplicated and reordered then no finite state solution to the sequence transmission problem exists (even if messages cannot be lost).

There are finite-state solutions if we restrict the environment's actions. In particular, as we now show, A^{fs} is a solution to the sequence transmission problem if we do not allow messages to be reordered (although they can be deleted, corrupted detectably, and duplicated).

Theorem 3.2: *Protocol A^{fs} solves the sequence transmission problem in systems where messages can be deleted, corrupted detectably, and duplicated (but not reordered). Every run of A^{fs} has the safety property and those runs of A^{fs} where S and R are schedule infinitely often and every message sent by S (resp. R) infinitely often is eventually delivered uncorrupted (when R (resp. S) is scheduled) also have the liveness property.*

We prove correctness of A^{fs} by showing that in a precise sense it is an *implementation* of protocol A, i.e., we show that $\mathcal{R}(A^{fs})$ can be mapped into an instance of an interpreted system consistent with A. The details are provided in the appendix.

4 A solution in the AUY model

In the AUY model [AUWY82,AUY79], message transmission is assumed to proceed in synchronous clocked rounds. We can conceptually think of a round or step as consisting of three phases: a send phase, a receive phase (where all messages sent in that step that are not deleted are received, possibly after being corrupted), and a local computation phase (during which data elements may be be read from the input sequence or written onto the output sequence). Since a message is received on the same round in which it is sent (if it is received at all), messages cannot be reordered or duplicated. As in [AUWY82,AUY79], we assume that the symbols transmitted over the channel are 0, 1, and λ (again λ denotes that nothing is sent), and that the input sequence X consists of 0s and 1s. We consider three types of errors in the communication medium:

- *Deletion* errors: 0 or 1 is sent, but λ (nothing) is received.
- *Mutation* errors: 0 (resp. 1) is sent, but 1 (resp. 0) is received.
- *Insertion* errors: λ is sent, but 0 or 1 is received.

As observed in [AUWY82,AUY79], the sequence transmission problem has no solution if all three error types are present. To see this, observe that if all error types are present, then any sequence σ of messages in $\{0, 1, \lambda\}^*$ transmitted by S can be altered by the channel to any other sequence σ' of the same length. Thus R can gain no information from the messages it receives about the messages that S actually transmitted. It was also shown in [AUWY82,AUY79] that for any combination of two out of the three possible types of errors, the problem is solvable. However, the informal correctness proofs presented in [AUWY82,AUY79] are difficult to follow, and some effort has been expended in constructing more formal proofs [Gou85,Hai85]. Unfortunately, these proofs are also far from transparent.

We provide solutions to the sequence transmission problem in the AUY model by implementing protocol A^{fs}. Note that A^{fs} as it stands does not provide a solution, since messages such as ack^2 are illegal in the AUY model.

We overcome this difficulty by encoding the messages λ, 0, 1, ack, ack^2, and ack^3 using only λ, 0, and 1. This is straightforward under our assumption that the system is synchronous.

Suppose first we restrict attention to deletion and insertion faults. Then we can encode 0 and ack as 00, 1 and ack^3 as 01, and ack^2 and λ as 10. Since S sends only 0, 1, and ack^2, while R sends only ack, ack^3, and λ, there is no conflict in this choice of encoding. We implement a command such as "send ack^2" in A^{fs} by first sending 1 and then sending 0. Similarly, "send 0" is now implemented as "send 0; send 0". With this encoding, we have effectively disabled insertion errors, since a λ is never sent.

We implement receiving by dividing time into *2-blocks*. The first 2-block consists of times 0 and 1, the next one consists of times 2 and 3, etc. Let e^{di} denote the encoding above (the *di* stands for "deletion" and "insertion"), so that $e^{di}(0) = 00$, $e^{di}(1) = 01$, and so on. If $e^{di}(m)$ is received by S (resp. R) in a 2-block, for some element m of the alphabet used in protocol A^{fs}, then we say that m is received. If part of a 2-block gets corrupted (by a message being deleted), then we say that λ is received.

Let us call the resulting protocol A^{di}. We describe A^{di} below. We have numbered the steps so that they correspond to those of A^{fs}. Essentially the only difference between A^{fs} and A^{di} is that a command of the form "send m" in A^{fs} is replaced by "send $e_1^{di}(m)$; send $e_2^{di}(m)$" in A^{di}, where we use $e_j^{di}(m)$, $j = 1, 2$, to denote the j^{th} element of $e^{di}(m)$ (so, for example, $e_1^{di}(1) = 0$ and $e_2^{di}(1) = 1$). We now assume that S uses variables z_1 and z_2 to receive messages, rather than just z; similarly, R uses z'_1 and z'_2. This way S and R can keep track of both messages received in a 2-block.

S's protocol:

INIT. read y; $z_1 := \lambda$; $z_2 := \lambda$
S1a. send $e_1^{di}(y)$; receive z_1; go to S1b
S1b. send $e_2^{di}(y)$; receive z_2;
if $z_1 z_2 = e^{di}(ack)$ **then** go to S2a **else** go to S1a
S2a. send $e_1^{di}(ack^2)$; receive z_1; go to S2b
S2b. send $e_2^{di}(ack^2)$; receive z_2;
if $z_1 z_2 = e^{di}(ack^3)$ **then** read y; go to S1a **else** go to S2a

R's protocol:

INIT. $z'_1 := \lambda$; $z'_2 := \lambda$
R1a. send $e_1^{di}(\lambda)$; receive z'_1; go to R1b
R1b. send $e_2^{di}(\lambda)$; receive z'_2;
if $z'_1 z'_2 = e^{di}(0)$ **then** write 0; go to R2a
else
if $z'_1 z'_2 = e^{di}(1)$ **then** write 1; goto R2a
else go to R1a
R2a. send $e_1^{di}(ack)$; receive z'_1; go to R2b
R2b. send $e_2^{di}(ack)$; receive z'_2;
if $z'_1 z'_2 = e^{di}(ack^2)$ **then** go to R3a **else** go to R2a
R3a. send $e_1^{di}(ack^3)$; receive z'_1; go to R3b
R3b. send $e_2^{di}(ack^3)$; receive z'_2;
if $z'_1 z'_2 = e^{di}(0)$ **then** write 0; goto R2a
else
if $z'_1 z'_2 = e^{di}(1)$**then** write 1; goto R2a
else go to R3a

Protocol A^{di}

The fairness condition required now is that there be infinitely many 2-blocks where both of S's transmissions are delivered, and infinitely many 2-blocks where both of R's transmissions are delivered.[1] There is a straightforward mapping from runs of protocol A^{di} to runs of protocol A^{fs}. Using this mapping, we show in the appendix that:

[1]Note that if we assume a fixed non-zero probability of any given message arriving, then the set of runs satisfying our fairness criterion has measure 1. Our fairness condition is equivalent to that used in [AUWY82,AUY79].

Theorem 4.1: *Protocol A^{di} solves the sequence transmission problem in the AUY model with deletion and insertion faults. Every run of A^{di} has the safety property, and those runs with infinitely many 2-blocks where both of S's messages are delivered and infinitely many 2-blocks where both of R's messages are delivered also have the liveness property.*

A number of optimizations are possible in protocol A^{di}. For example, if R receives a 1 in step R2a, then it knows that S sent an ack^2, independent of what it receives in step R2b. (This is so because e^{di} has the property that if $e_1^{di}(m) = 1$ then m is either ack^2 or λ, and R knows that S never sends λ.) Similarly, if R receives 1 at step R3b, then R knows that S must be trying to send $e^{di}(1)$, even if it does not receive any message at step R3a; R could thus safely write a 1 and go to step R2a. Clearly other optimizations are possible. We could also consider trying to get an encoding that is more efficient in some sense. We have not bothered pursuing these issues here, since our goal is to get an easily understood and easily verifiable protocol, not one that is most efficient. (We remark that [AUWY82] presents a protocol with an optimal transmission rate.)

Although A^{di} solves the sequence transmission problem in the AUY model for the case of deletion and insertion errors, it cannot deal with mutation errors. If 0 and 1 can be mutated, then it is possible that, for example, S sends $e^{di}(0) = 00$ and R receives $01 = e^{di}(1)$. We solve this problem by using a different encoding function when dealing with mutation errors. The key observation here is that the encoding e^{di} we used in A^{di} could be replaced by another encoding e, as long as e has the following two properties (with respect to the faulty behavior allowed):

1. (*Unique decodability*) If $e(x)$ is received uncorrupted, then the recipient knows that it is uncorrupted, and that it is an encoding of x.
2. (*Corruption detectability*) If $e(x)$ is corrupted, then the recipient knows it is corrupted.

An encoding with these properties essentially allows us to treat any type of faulty behavior as a deletion error. If corruption is detected, a message can be ignored and treated as if it were lost. Clearly e^{di} has these properties in the case of deletion and insertion errors. In order to implement A^{fs} in the presence of deletion and mutation errors, we must find an encoding that preserves unique decodability and corruption detectability in the presence of these errors. One encoding that works is e^{dm}, defined by:

$$e^{dm}(0) = e^{dm}(ack) = 1\lambda\lambda,$$
$$e^{dm}(1) = e^{dm}(ack^3) = \lambda 1\lambda,$$
$$e^{dm}(ack^2) = e^{dm}(\lambda) = \lambda\lambda 1.$$

It is easy to check that e^{dm} does indeed have the properties of unique decodability and corruption detectability in the case of deletion and mutation errors. Clearly any uncorrupted sequence of messages received in a 3-block is uniquely decodable. A mutation error is easily detectable and, indeed, easily corrected: Since 0 does not appear in any encoded message, if 0 is received it must be the case that a 1 was sent and mutated. If a message is deleted, then the receipient will get $\lambda\lambda\lambda$ in a 3-block and know that the message was corrupted.

Finally, in order to deal with mutation and insertion errors, we can use the encoding e^{mi} which is obtained by reversing the roles of 1 and λ in e^{dm}:

$$e^{mi}(0) = e^{mi}(ack) = \lambda 11,$$
$$e^{mi}(1) = e^{mi}(ack^3) = 1\lambda 1,$$
$$e^{mi}(ack^2) = e^{mi}(\lambda) = 11\lambda.$$

We leave it to the reader to check that e^{mi} has the required properties in the case of mutation and insertion errors. (Of course, neither e^{dm} nor e^{mi} is unique; there are many possible encodings with the appropriate properties.)

We can now construct protocols A^{dm} and A^{mi} from A^{fe} by doing our encoding using e^{dm} and e^{mi}, respectively. Note that for A^{dm} and A^{mi}, we divide up time into 3-blocks rather than 2-blocks, and, to prove correctness, we modify the fairness condition accordingly. Formally, we have the following extension of Theorem 3:

Theorem 4.2: *Protocol A^{dm} (resp. A^{mi}) solves the sequence transmission problem in the AUY model with deletion and mutation (resp. mutation and insertion) errors. Every run of A^{dm} (resp. A^{mi}) has the safety property, and those runs with infinitely many 3-blocks where all of S's messages are delivered and infinitely many 3-blocks where all of R's messages are delivered also have the liveness property.*

We could deal with other error types too, provided we could find appropriate encoding functions. There is, however, no encoding function with the properties of unique decodability and corruption detectability that can deal simultaneously with deletion, mutation, and insertion errors.

Note that as we move further away from our knowledge-based protocol, we make more and more use of the details of the underlying model. In particular, A^{fe} makes heavy use of the assumption that messages are not reordered, and the protocols A^{di}, A^{dm}, and A^{mi} make heavy use of the synchronous nature of the AUY model, so that messages are received in the same round that they are sent (if they are received at all).

5 Other solutions to the sequence transmission problem

In this section we construct two other families of protocols that solve the sequence transmission problem.

In the solution constructed in Section 3, S waited until it knew that R knew that S knew that R knew x_i (i.e., until $K_S K_R K_S K_R(x_i)$ held) before sending x_{i+1}. Does S really need this *depth 4* knowledge before sending x_{i+1}?

Suppose X is an input sequence with $x_0 = 0$ and $x_1 = 1$. Again S sends x_0 until it gets an acknowledgement. But in this case it would be safe to send x_1 without needing to send "$K_S K_R(x_0)$". Unlike the previous example, where we had $x_0 = x_1 = 0$, there would now be no ambiguity about whether S was trying to resend x_0 (because $K_S K_R(x_0)$ did not hold) or sending x_1. The point is that although in this case S does not know that R knows that S

knows that R knows x_0 before it sends x_1, S does know that when R gets x_1, R will know that S knows that R knows x_0. Intuitively, "$K_S K_R(x_0)$" is "piggy-backed" on top of the x_1.

Below we describe a knowledge-based protocol B that captures this idea. In protocol B, S starts sending "$x_{i+1} = y$" to R when S knows that R knows (the value of) x_i. Since such messages are distinct for all values of i, this message encodes $i+1$, so that when R gets it, R knows that S knows that R knows x_i.

Protocol B^{fs} is a finite-state implementation of B, using the symbols 0, 0′, 1, and 1′. S has a variable i that keeps track of the parity of the data element it is currently sending. If the last data element that S read has value y and parity i, then S sends R the message $var(y,i)$, where $var(y,i)$ is y if i is 0 and y' if i is 1. Assume, for example, that both the 6^{th} and the 7^{th} data elements are 0. S will use 0 to send the 6^{th} and 0′ to send the 7^{th}, thus eliminating ambiguity. Similarly, R has a variable i' that keeps track of the parity of the data element it next wants to receive. If R receives a message z' of the form $var(0,i')$ or $var(1,i')$, then R writes $var_1^{-1}(z')$, the first component of $var^{-1}(z')$. R then sets $i' := i' \oplus 1$ (where $\oplus$ is addition mod 2), and acknowledges by sending $i' \ominus 1$ to S (thus telling S it has received the old value and wants a new one). If, instead, R receives a message of the form $var(0, i' \ominus 1)$ or $var(1, i' \ominus 1)$, then this message is ignored, since it is a data element that R has already written.

S's protocol:

INIT. read y; $i := 0$; $z := \lambda$
S1. send "$x_i = y$"; receive z;
if $K_S K_R(x_i)$
then $i := i + 1$; read y;
go to S1

R's protocol:

INIT. $i' := 0$; $z' := \lambda$
R1. send λ; receive z';
if $K_R(x_{i'})$
then write $x_{i'}$; $i' := i' + 1$;
go to R2
else go to R1
R2. send "$K_R(x_{i'-1})$"; receive z';
if $K_R(x_{i'})$ **then** write $x_{i'}$; $i' := i' + 1$
then write $x_{i'}$; $i' := i' + 1$;
go to R2

Protocol B

S's protocol:

INIT. read y; $i := 0$; $z := \lambda$
S1. send $var(y,i)$; receive z;
if $z = i$
then $i := i \oplus 1$; read y;
go to S1

R's protocol:

INIT. $i' := 0$; $z' := \lambda$
R1. send λ; receive z';
if $(z' = var(0,i') \vee z' = var(1,i'))$
then write $var_1^{-1}(z')$; $i' := i' \oplus 1$;
go to R2
else go to R1
R2. send $i' \ominus 1$; receive z';
if $(z' = var(0,i') \vee z' = var(1,i'))$
then write $var_1^{-1}(z')$; $i' = i' \oplus 1$;
go to R2

Protocol B^{fs}

It is straightforward to prove that protocol B solves the sequence transmission problem in the same wide range of settings as A, again under the assumption that processes' states record their message histories. And, just like A^{fs}, protocol B^{fs} solves the sequence transmission

problem in systems where messages can be deleted, duplicated, or detectably corrupted (but not reordered). We can also convert B^{fs} to protocols B^{di}, B^{dm}, and B^{mi} that deal with any pair of error types in the AUY model by doing encoding just as in the previous section.

It is interesting to note that the protocols presented in [AUWY82,AUY79] are all essentially implementations of protocol B^{fs}. For example, the reader can check that the protocol of [AUWY82,AUY79] that handles only deletion errors is an implementation using the encoding e^d defined by:

$$e^d(0) = 0\lambda,\ e^d(1) = e^d(ack') = 1\lambda,$$
$$e^d(0') = \lambda 0,\ e^d(1') = e^d(ack) = \lambda 1,$$
$$e^d(\lambda) = \lambda\lambda.$$

The two families of protocols discussed so far can be viewed as *sender-driven* protocols: S sends x_i if S does not know that R knows x_i; i.e., S sends x_i when $\neg K_S K_R(x_i)$ holds. We can also consider *receiver-driven* protocols, where S sends x_i only if S knows that R does *not* know x_i; i.e., when $K_S \neg K_R(x_i)$ holds. Receiver-based protocols might be quite practical if S is relatively busy, or if S's messages are quite long rather than only consisting of one bit.

Such a receiver-driven protocol is given below. As usual, we describe a knowledge-based protocol C that captures this idea, and also give a finite-state protocol C^{fs} that implements C. (We discuss below the assumptions under which C and C^{fs} are correct.) The idea behind C^{fs} is again quite simple. As in protocol B^{fs}, S uses a variable i to keep track of the parity of the data element it last sent, and R uses a variable i' to keep track of the parity of the data element it next wants to receive. R keeps sending i' to S (thus telling S the parity of the next data element it wants to receive). If S is at S1, then it sends R its current value and moves to line S2. If S is at S2, it sends λ until it receives a non-λ message z from R. If $z = i \oplus 1$, then S knows that R has received the current value, so that S reads the next value. If $z = i$, then S knows that R still does not know the current value. (Since S sent a λ in the current round, R could not have learned it.) In either case, S now knows what value R does not know, so it moves to line S1 and sends it.

Note that S never sends non-λ messages in two consecutive rounds. The reason is that if S sent the current data element at round m and does not get a message from R at round m saying that R knows that data element (presumably because it was also sent on an earlier round), then S does not know whether or not R knows the value of the current data element at the beginning of round $m+1$. (We could be a bit more efficient by having S send the next value if it knows that R just received a value, i.e., if it receives $i \oplus 1$ in S1, but we do not bother to do this here.) Also note that R can write an element as soon as it receives it; there is no ambiguity. The protocol guarantees that any value received by R is the value of the next data element.

Recall that we want the knowledge-based protocol C to behave properly even in a system where there is some *a priori* knowledge about the values of the data elements. Thus C has tests for knowledge not present in C^{fs}. In particular, it may be possible that $K_R(x_i)$ holds without R receiving any message about x_i from S, in which case R sends λ (in line R0) rather than "$\neg K_R(x_i)$".

It is not the case that C works in the same general setting as A and B. Suppose, for example, that we consider an asynchronous system where messages may take an arbitrary

S's protocol:

INIT. read y; $i := 0$; $z := \lambda$
S1. send "$x_i = y$"; receive z; go to S2
S2. send λ; receive z;
if $K_S K_R(x_i)$
then $i := i + 1$; read y;
if $K_S \neg K_R(x_i)$ **then** go to S1
else go to S2

R's protocol:

INIT. $i' := 0$; $z' := \lambda$
R0. send λ; receive z';
if $K_R(x_{i'})$ **then** write $x_{i'}$; $i' := i' + 1$;
if $\neg K_R(x_{i'})$ **then** go to R1
else go to R0
R1. send "$\neg K_R(x_{i'})$"; receive z';
if $K_R(x_{i'})$
then write $x_{i'}$; $i' := i' + 1$;
if $\neg K_R(x_{i'})$ **then** go to R1
go to R0

Protocol C

S's protocol:

INIT. read y; $i := 0$; $z := \lambda$
S1. send y; receive z; go to S2
S2. send λ; receive z;
if $z = i \oplus 1$
then $i := i \oplus 1$; read y;
if $z = i$ **then** go to S1
else go to S2

R's protocol:

INIT. $i' := 0$; $z' := \lambda$
R1. send i'; receive z';
if $z' \neq \lambda$
then write z'; $i' := i' \oplus 1$;
go to R1

Protocol C^{fs}

number of rounds to be delivered. It is easy to see that C has the safety property, but liveness will not hold in general. The problem is that S only sends "$x_i = y$" when $K_S \neg K_R(x_i)$ holds. However, even if S gets a message from R saying "$\neg K_R(x_i)$", it is not necessarily the case that $K_S \neg K_R(x_i)$ holds. Although this message may have been true when it was sent, it may no longer be true after it is received. S has no way of knowing whether R received an "$x_i = y$" message sometime after R sent its message. While $K_R(x_i)$ is a *stable* formula (once true it remains true), $\neg K_R(x_i)$ is not stable. However, C does have the liveness property in systems where messages have finite lifetimes (i.e., systems where there is some constant T such that messages are delivered within time T if they are delivered at all). Note the AUY model is a special case of a system where messages have finite lifetimes; in this case the lifetime is one round.

On the other hand, just as for A^{fs} and B^{fs}, it is straightforward to prove that protocol C^{fs} solves the sequence transmission problem in systems where messages can be deleted, duplicated, and detectably corrupted (but not reordered). In fact, since the only messages sent in protocol C^{fs} are 0, 1, and λ, C^{fs} actually solves the sequence transmission problem in the AUY model with only deletion errors. We can also find protocols C^{di}, C^{dm}, and C^{mi} that deal with any pair of error types by doing encoding just as in the previous section.

6 The Alternating Bit and Stenning's protocol

The communication model for the Alternating Bit protocol assumes that the data domain consists of k bit strings, for some fixed k, and that we can send $(k+1)$-bit messages along the channel. The channel is is asynchronous; messages can take an arbitrary number of rounds to arrive. Messages may also be lost or detectably corrupted.

Recall that protocol B^{fs} works perfectly well even without the assumption of complete synchrony, as long as messages are not reordered. In fact, the Alternating Bit protocol turns out to be simply an implementation of protocol B. It follows much the same lines as B^{fs}, except that now $var(m,i)$ is $m0$ if i is even and $m1$ if i is odd (where mn denotes the result of concatenating m and n). The proof of correctness is now precisely the same as that of protocol B^{fs}.

As we remarked before, in an event-driven system with no rounds, we must modify the protocol so that messages are sent periodically until an appropriate acknowledgement is received. This can be done using a timer. We omit the straightforward details here.

We now consider Stenning's protocol, which handles a wider variety of faulty behavior than the Alternating Bit protocol. Messages can be lost, duplicated, reordered, or corrupted detectably. However, we now assume we can send messages of unbounded length on the channel. Stenning's protocol is a completely straightforward implementation of protocol B. Essentially all that is done is that the test 'if $K_S K_R(x_i)$ ...' on line S1 is replaced by 'if $z =$ "$K_S K_R(x_i)$" ...' and the test "if $K_R(x_{i'})$ ..." on line R1 is replaced by 'if $z' =$ "$x_{i'} = y$" ....' (Recall that the messages such as "$K_S K_R(x_i)$" are arbitrary strings of symbols.)

Correctness of the Alternating Bit protocol and Stenning's protocol now follow immediately from the correctness of protocol B.

7 Necessary and sufficient conditions on knowledge

Part of our motivation in constructing protocol B was to investigate whether S needs depth 4 knowledge of the form $K_S K_R K_S K_R(x_i)$ before sending x_i to R. Although S does not wait until it attains this knowledge before sending x_{i+1} in protocol B, S does eventually attain it. In particular, S knows that when R receives x_{i+1}, R will know that S knows that R knows x_i. More formally, the following formula is valid in all reasonable interpreted systems consistent with protocol B (recall an interpreted system is reasonable if the formulas $x_i = 0$ and $x_i = 1$ have the appropriate interpretation):

$$K_S \Box (R \text{ received } x_{i+1} \supset K_R K_S K_R(x_i)), \tag{1}$$

where $\Box$ be the standard temporal logic symbol for *always*. (In our formal model, we have $(I,r,m) \models \Box\varphi$ iff $(I,r,m') \models \varphi$ for all $m' \geq m$.)

The question arises whether the state of knowledge defined by Equation 1 is really necessary. It is easy to see that the answer is no. For example, S might assume that the channel is relatively reliable and send five messages at a time before waiting for an acknowledgement. In this case,

S would be sending x_1 without even knowing that R knew x_0 (although it would not send x_5 before it knew that R knew $x_0, \ldots, x_4$; moreover, S would know that when R received x_5, R would know that S knew that R knew $x_0, \ldots, x_4$). However, provided we require that S and R alternate reading and writing (i.e., S reads x_{i+1} only after R writes x_i), then we can show that there is a precise sense in which the level of knowledge described in Equation 1 is necessary and sufficient. Observe that in protocols A^{fs}, B^{fs}, and C^{fs} (as well as the Alternating Bit protocol and Stenning's protocol), reads and writes do indeed alternate, so this is not an unreasonable assumption.

There is a small problem in making this statement completely precise: We must make clear exactly what it means for R to receive x_{i+1}. In a protocol such as A or B, we can take this to mean that R gets a message of the form "$x_{i+1} = y$". However, in a protocol such as A^{fs}, R only receives a message of the form 0, 1, ack^2, or λ. While we could say that R receives x_{i+1} when it receives the first 0 or 1 after S reads x_{i+1}, it is not immediately clear how to generalize this notion to receiving x_{i+1} for an arbitrary protocol. The situation becomes even worse when we consider a protocol such as A^{di} or A^{dm}, where none of the messages received corresponds to x_{i+1}; rather, a whole block of messages does.

We avoid this problem by replacing the phrase "R received x_{i+1}" by "R wrote x_{i+1}". We restrict our attention in what follows to interpreted systems I where for any point (r, m) in I, the environment component has the form $(X, c, Y, \ldots)$, where X is the input sequence, c is a counter that describes which element of X is currently being read by S, and Y is the output sequence of elements thus far written by R. (We remark that in the formal semantics for protocols A, A^{fs}, A^{di}, A^{dm}, and A^{mi} given in the appendix, the environment component always has this form.) The formula "R wrote x_i" is true at the point (r, m) if output sequence Y in $r(m)$ has length at least $i+1$. (Since we also restrict attention to runs that have the safety property, it will actually be x_{i+1} that is written at this point.) Note that it is the case that in all the protocols we considered, x_{i+1} is written at the same round that it is first received by R, so this change does capture the intuition we had all along. Similarly, we take "S read x_i" to be true at (r, m) if the value of c in $r(m)$ is at least i.

We do need to make one further assumption on the set of runs in order to prove the result. We must assume that processes remember the sequence of values they have read or written (i.e., this sequence is encoded in their local states). Note that this is weaker than the assumption that processes record their complete message history, since from the message history it can be deduced what values were read and written.

Theorem 7.1: *Let I be a reasonable interpreted system such that for any point in I:*

1. *the safety property holds (so that Y is always a prefix of X),*
2. *reading and writing alternate (so that the number of data elements read by S is always $|Y|$ or $|Y|+1$),*
3. *S's state records all the elements it has read, and R's state records all the elements it has written.*

Then for all points (r, m) in I and all $i \geq 0$, we have

$$(I, r, m) \models K_S \Box (R \text{ wrote } x_{i+1} \supset K_R K_S K_R(x_i)). \tag{2}$$

Proof To prove Equation 2, it suffices to show that for all points (r,m) in I we have

$$(I,r,m) \models (R \text{ wrote } x_{i+1} \supset K_R K_S K_R(x_i)). \tag{3}$$

Equation 2 follows from Equation 3 and the general observation that if $(I,r,m) \models \varphi$ for all r and m, then $(I,r,m) \models K_S \Box \varphi$ for all r and m.

To prove Equation 3, assume, by way of contradiction, that for some point (r,m), we have

$$(I,r,m) \models (R \text{ wrote } x_{i+1} \wedge \neg K_R K_S K_R(x_i)).$$

It follows that we can find points (r',m') and (r'',m'') such that $(r,m) \sim_R (r',m')$, $(r',m') \sim_S (r'',m'')$, $(I,r',m') \models \neg K_S K_R(x_i)$, and $(I,r'',m'') \models \neg K_R(x_i)$.

Since $(r,m) \sim_R (r',m')$ and R's state records the elements that R has written, it must be the case that $(I,r',m') \models R$ wrote x_{i+1}. Since reading and writing alternate, it must be the case that $(I,r',m') \models S$ read x_{i+1}. But $(r',m') \sim_S (r'',m'')$ and S's state records the elements it has read, so we must have $(I,r'',m'') \models S$ read x_{i+1}. Again, since reading and writing alternate, we must have $(I,r'',m'') \models R$ wrote x_i. Thus, in $r''(m'')$, the output sequence has length at least $i+1$. Suppose, without loss of generality, that the $(i+1)^{\text{st}}$ element written is 0. Since $(I,r'',m'') \models \neg K_R(x_i)$, there must be some point (r''',m''') such that $(r'',m'') \sim_R (r''',m''')$ and $(I,r''',m''') \models (x_i = 1)$. Since R's state records all the elements it has written, it must be the case that the $(i+1)^{\text{st}}$ element written by R at the point (r''',m''') is 0. But this contradicts the safety property. Hence Equation 3 must hold. ∎

8 Conclusions

We have described high-level knowledge-based protocols for the sequence transmission problem, and have shown that several well-known protocols are simply implementations of one of our knowledge-based protocols. These observations allowed us to provide well-motivated and easy correctness proofs for all these protocols. We feel that such an approach—starting with a knowledge-based protocol that is almost model-independent, and then implementing the knowledge acquisition using particular properties of the model—leads to a better understanding of the protocol and far easier proofs of correctness than other approaches that have been used. Moses and Tuttle [MT86] also start with a knowledge-based protocol (for Byzantine agreement) and derive efficient implementations of it. Gafni has also advocated a similar "layered" approach to protocol design [Gaf86], although he did not specifically suggest using knowledge-based protocols at the highest level.

We also considered the state of knowledge required to perform the sequence transmission problem, and essentially characterized it in the case of alternating reads and writes via Equation 2. Theorem 5 demonstrates that this state of knowledge is a necessary precondition; protocols A and B demonstrate that it is sufficient. It would be interesting to try to characterize the state of knowledge required to solve the sequence transmission problem without the additional assumption of alternating reads and writes.

The knowledge-based protocols we designed assumed processes with infinitely many distinct states and required that infinitely many distinct messages could be sent. In retrospect, this is

perhaps not surprising. It is often the case that a high-level solution to a problem is inefficient. Protocols such as the AUY protocols and the Alternating Bit protocol show that under some circumstances (i.e., under appropriate restrictions on the actions of the environment) we can find finite-state solutions to the sequence transmission problem. However, as we observed at the end of Section 3, if we allow messages to be reordered and lost (or reordered and duplicated), and there is no upper bound on message delivery time (for messages that are delivered), then there is no finite-state solution. (We remark that if there is an upper bound on message delivery time, then the modified Stenning protocol [Ste76] gives a finite-state solution.) And if we allow undetectable corruption of messages (so that any message can be converted to any other), then there is no solution at all, either finite state or infinite state (since any message sequence can then be converted to any other). It would be interesting to characterize the assumptions required to guarantee that a solution to the sequence transmission problem exists, and the assumptions required to guarantee a finite-state solution, and then to generalize these results to the case where we have a whole network rather than just two processes (see [GA87] for preliminary work along these lines).

A Appendix: Semantics and correctness of the protocols

A.1 Semantics for A and A^{fs}

We give the semantics of A and A^{fs} as a set of runs as described in Section 2. We start with A. Our first step is to specify $\mathcal{G}$—the set of possible global states of protocol A.

Let M_S (resp. M_R) be the set of messages that S (resp. R) can send according to A_S (resp. A_R), i.e., all the messages of the type "$x_i = y$" and "$K_S K_R(x_i)$" (resp. "$K_R(x_i)$" and "$K_R K_S K_R(x_i)$"). Note that both M_S and M_R are infinite. Since messages can be lost or corrupted (detectably), the set of messages S (resp. R) can receive is actually $M_R \cup \{\lambda, *\}$ (resp. $M_S \cup \{\lambda, *\}$), where λ denotes that nothing is received and $*$ denotes a special corrupted message.

We have some latitude when it comes to representing the runs of A. We have chosen to make explicit the fact that a "step" of the protocol consists of a sending phase, receiving phase, and a local computation phase. In order to bring this out in the set of runs, we take L_S to consist of states of the form (Sj, q, y, z, h, i), where $j \in \{1, 2\}$, $q \in \{s, r, l\}$, $y \in \{0, 1\}$, $z \in M_R \cup \{\lambda, *\}$, $i \geq 0$, and h, S's history, is a sequence over $\{\text{"sent } m\text{"} \mid m \in M_S\} \cup \{\text{"received } m\text{"} \mid m \in M_S \cup \{\lambda, *\}\}$. The first component, Sj, is the step in the protocol S is about to perform or in the midst of performing, the second component, q, describes whether S is in the *s*end, *r*eceive, or *l*ocal computation phase, the third component, y, is the data element last read, the fourth component is the value last received, h is a sequence, initially empty, that is updated whenever S sends/receives a message (such that every time a message m is sent by S, "sent m" is appended to h, and every time a message m' is received by S, "received m'" is appended to h), and i records the value of the counter i used in A_S. Similarly, we take L_R to consist of states of the form (Rk, q', z', h', i'). L_e, the environment states, consist of states of the form $(X, c, Y, q'', b_S, b_R, go_S, go_R)$, where X is the input sequence, c is a counter describing which element of X is currently being read, Y is the sequence of elements written, q'' is the phase, b_S

(resp. b_R) is a sequence over M_S (resp. M_R) containing all the messages sent by S (resp. R). We take go_S (resp. go_R) to be a Boolean variable whose valuen is 0 or 1 depending on whether S (resp. R) is scheduled to move on that round.

$\mathcal{G}_0$, the set of initial global states of A, consists of all global states of the form

$$((X, 0, \langle\rangle, s, \langle\rangle, \langle\rangle, go_S, go_R), (S1, s, x_0, \lambda, \langle\rangle, 0)), (R1, s, \lambda, \langle\rangle, 0)),$$

where X is an infinite sequence of 0s and 1s.

We define $\mathcal{R}_A^p$, the set of *potential runs of* A, to be the set of all runs r over $\mathcal{G}$ such that $r(0) \in \mathcal{G}_0$. When we consider interpreted systems that are consistent with A, we will only consider systems $\mathcal{R}$ that are consistent with respect to $\mathcal{G}$, $\mathcal{G}_0$, and τ, where τ is a transition function we are about to define. In particular, we will have $\mathcal{R} \subseteq \mathcal{R}_A^p$.

It is now easy to view S's protocol A_S as a function from its local state and an intepreted system I to actions. In a sending phase, S performs an action of the form "send m", $m \in M_S$; in a receiving phase it performs an action of the form "receive z"; in the local phase it performs actions of the form "go to Sj" and perhaps "read y" or "$i := i + 1$". The only case where the interpreted system I plays a role is when there is a test for knowledge (in a local phase). Typical clauses for A_S include:

- $A_S((\mathrm{S1}, s, y, z, h, i), I) =$ send "$x_i = y$";
- $A_S((\mathrm{S2}, s, y, z, h, i), I) =$ send "$K_S K_R(x_i)$";
- $A((\mathrm{S}j, r, y, z, h, i), I) =$ receive z
- $A_S((\mathrm{S1}, l, y, z, h, i), I) = \begin{cases} \text{go to S2} & \text{if } (I, r, m) \models K_R(x_i) \text{ for all global states } r(m) \\ & \text{where } S\text{'s state is } (\mathrm{S1}, l, y, z, h, i) \\ \text{go to S1} & \text{otherwise.} \end{cases}$

We leave it to the reader to fill in the remaining clauses for A_S and the similar clauses for A_R.

The environment runs the nondeterministic protocol that performs the empty action Λ during the *send* phase, nondeterministically sends messages during the *receive* phase (in a manner described below), and perform actions of the form $go_j := i$, for $j \in \{\mathrm{S,R}\}$ and $i \in \{0, 1\}$. (Note that the values of go_S and go_R are only changed after the local computation phase, so they are constant during a round.)

At the receive phase, the environment is allowed to lose, reorder, or duplicate messages, as well as to detectably corrupt them. We capture this by having the environment nondeterministically choose to perform a "send$_S$ m" action, for some $m \in b_R \cup \{*, \lambda\}$ whenever it is in the receive phase and $go_S = 1$. The result of this action is that S will receive the message m. Similarly, the environment nondeterministically chooses to perform a "send$_R$ m" action for some $m \in b_S$ in the receive phase if $go_R = 1$. The environment does not send any messages to S (resp. R) if S (resp. R) is not scheduled. (We could of course assume that messages are delivered even when processes are not scheduled, or that the environment can deliver more than one message at a time. The resulting model would be similar to the one we use.) Duplication of messages is possible, since a message is not deleted from b_R or b_S after being sent. We capture the possibility of message corruption by allowing the environment to send the message $*$ (other

more sophisticated ways of capturing message corruption are clearly possible); message deletion is captured by allowing the environment to send λ.

Finally, we need to define τ. The definition is completely straightforward, although tedious to write down. For example, we have

$$((X,c,Y,s,b_S,b_R,1,0),(S1,s,y,z,h,i),(R3,s,z',h',i')) \xrightarrow{\tau(\Lambda,\text{“send } m\text{”},\text{“send } m'\text{”})} ((X,c,Y,r,b_S;\langle m\rangle,b_R,1,0),(S1,r,y,z,h;\langle\text{“sent } m\text{”}\rangle,i),(R3,s,z',h',i')).$$

This says that if, in the send phase, S sends m and R sends m' when only S is scheduled, then the result is that m is appended to the environment's message buffers b_S, "sent m" is appended to S's message history (h), and S moves into the receive phase. Note that the actions of R is disabled (i.e., has no effect on the system) since R is not scheduled ($go_R = 0$).

The effect of $\tau(act_e, act_S, act_R)$ is similar for other joint actions. For example, in the receive phase, if the environment sends some messages, then h and h' are updated with the values of these messages (by appending "received m"). The result of "read y" is that y is set to x_c (if $X = \langle x_0, x_1, x_2, \ldots\rangle$) and c is increased by 1, while the result "write z'" is that z' is appended to Y. We leave further details to the reader.

Although up to now we have implicitly been associating A with the pair (A_S, A_R), we now formally take it to be the joint protocol (A_e, A_S, A_R), where A_e is the nondeterministic protocol for the environment described above.

We have now given all the details necessary to determine whether a given set of runs is consistent with the knowledge-based protocol A. Note that this construction guarantees that the assumptions of Theorem 1 hold. In particular, both S's and R's local states contain a complete record of the messages they have sent and received.

To give the semantics of A^{fs}, we start by specifying $\mathcal{G}^{fs}$, the set of global states of A^{fs}. The definition of $\mathcal{G}^{fs}$ is similar to the definition of $\mathcal{G}$; the main difference is that we omit the message histories, the counters, and the infinite message buffers in the local states of S and R (after all, A^{fs} is intended to be a finite state protocol!). We therefore take L_S^{fs} to consist of states of the form (Sj, q, y, z), where j, q, and y are just like before, and $z \in \{*, \lambda, ack, ack^3\}$. Similarly, we take L_R^{fs} to consist of states of the form (Rk, q', z') where $z' \in \{*, \lambda, 0, 1, ack^2\}$. L_e^{fs}, the set of possible environment states, consists of states of the form $(X, c, Y, q'', b_S, b_R, go_S, go_R)$, where X, c, Y, q'', go_S, and go_R are just like before, and b_S (resp. b_R) is a sequence over $\{0, 1, ack^2\}$ (resp. $\{ack, ack^3\}$).

We take the set $\mathcal{G}_0^{fs}$ of initial global states of A to consist of all global states of the form

$$((X, 0, \langle\rangle, s, \langle\rangle, \langle\rangle, go_S, go_R), (S1, s, x_0, \lambda)), (R1, s, \lambda))$$

where X is an infinite sequence of 0s and 1s.

Recall that A^{fs} is a standard protocol. It is now easy to view S's protocol A_S^{fs} as a function from L_S^{fs} to actions. For example, we have

- $A_S^{fs}((S1, s, y, z)) =$ send y;

- $A_S^{fs}((S2, s, y, z)) =$ send ack^2;
- $A^{fs}((Sj, r, y, z)) =$ receive z
- $A_S((S1, l, y, z, h, i), \mathcal{R}) = \begin{cases} \text{go to S2} & \text{if } z = ack^2 \\ \text{go to S1} & \text{otherwise.} \end{cases}$

We leave it to the reader to fill in the remaining clauses for A_S^{fs} and the similar clauses for A_R^{fs}.

Unlike A, in A^{fs} the environment cannot reorder messages (although it can delete them, duplicate them, or corrupt them detectably). We capture this by having b_S and b_R (the message buffers which are part of the environment's state) consist only of the messages that the environment can still deliver. Thus, if $b_S = \langle m_1, \ldots, m_k \rangle$ and the environment delivers the message m_j for some $j \leq k$, then all the messages m_i with $i < j$ are deleted from b_S. (We remark that we can show that for A^{fs}, the buffers are always of length at most 2.) In the receive phase, if $go_S = 1$ (resp. $go_R = 1$) the environment nondeterministically chooses an element of $b_R \cup \{*, \lambda\}$ (resp. $b_S \cup \{*, \lambda\}$) and delivers it to S (resp. to R). It also has to update b_S and b_R as explained above. As in A, the environment also performs actions of the form $go_j := i$ for $j \in \{S, R\}$ and $i \in \{0, 1\}$ in the local computation phase, and performs the null action Λ in the send phase.

We define a transition function τ^{fs} in much the same way as before. We leave further details to the reader. We now formally take A^{fs} it to be the joint protocol $(A_e^{fs}, A_S^{fs}, A_R^{fs})$, where A_e^{fs} is the nondeterministic protocol for the environment described above.

We now have given all the details necessary to generate $\mathcal{R}(A^{fs})$, the set of all runs consistent with A^{fs}.

A.2 Correctness of A

In this subsection we prove Theorem 1. Let $I = (\mathcal{R}, \pi)$ be a reasonable interpreted system consistent with A satisfying the semantic construction of the previous subsection. Since I is otherwise an arbitrary interpreted system, it might well be a system where there is no message duplication (since there may be no runs in $\mathcal{R}$ where the environment delivers the same message twice). There may also be some *a priori* knowledge in I about the input sequence X; for example, it might be common knowledge (i.e., true in all runs of $\mathcal{R}$) that the first and second data elements in X are the same.

The proof of safety for A is almost trivial, since R writes $x_{i'}$ only if R knows its value. Formally, we show

Lemma 1 *For all runs $r \in \mathcal{R}$ and all times $m \geq 0$, if $r(m) = (s_e, s_S, s_R)$, where s_e is of the form $(X, c, Y, \ldots)$ and s_R is of the form $(Rk, \ldots, i')$, then $Y \preceq X$ and $i' = |Y|$.*

Proof We proceed by induction on m. The case $m = 0$ follows from our characterization of $\mathcal{G}_0$, the set of initial global states. For the inductive step, note that Y and i' are the same in $r(m)$ and $r(m+1)$ unless in $r(m)$ it is the case that R is scheduled, R's local state is of the form (Rk, l, z, h, i'), where $k = 1$ or $k = 3$, and $(I, r, m) \models K_R(x_{i'})$. But in this case $x_{i'}$ is appended to Y and i' is increased by one. Thus, in $r(m+1)$, we still have $Y \preceq X$ and $i' = |Y|$. ∎

In order to prove liveness, we need the three preliminary lemmas. Following [HM84], we say that a formula φ is *stable with respect to I* if, once φ is true in a run of I, it remains true; i.e., $(I,r,m) \models \varphi$ implies $(I,r,m') \models \varphi$ for all $m' \geq m$.

Lemma 2 *If φ, φ_1 and φ_2 are stable with respect to I, then so are $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, and $K_j\varphi$ for $j \in \{S,R\}$.*

Proof The fact that $\varphi_1 \wedge \varphi_2$ and $\varphi_1 \vee \varphi_2$ are stable is immediate. The stability of $K_j\varphi$ depends on the fact that processes do not forget (i.e., they record their message histories in their local states). Suppose $(I,r,m) \models K_j\varphi$ and $m' \geq m$; we want to show that $(I,r,m') \models K_j\varphi$. Suppose $r' \in \mathcal{R}$ and $(r',\ell') \sim_j (r,m')$. Since j's message history at $r(m)$ must be a prefix of j's message history at $r'(m)$, while j's message history at $r'(\ell')$ is the same as its message history in $r(m')$ (this is because $(r',\ell') \sim_j (r,m')$ and we assume that j's state records its message history), it easily follows that there is some $\ell \leq \ell'$ such that $(r,m) \sim_j (r',\ell)$. Since $(I,r,m) \models K_j\varphi$ and $(r,m) \sim_j (r',\ell)$, it follows that $(I,r',\ell) \models \varphi$. Since φ is stable, we have that $(I,r',\ell') \models \varphi$. We have just shown that for all points (r',ℓ') with $(r',\ell') \sim_j (r,m')$, we have $(I,r',\ell') \models \varphi$. Thus $(I,r,m') \models K_j\varphi$. It follows that $K_j\varphi$ is stable with respect to I. ∎

Lemma 3 *For every $i \geq 0$, $j \in \{R,S\}$, and φ of the form $x_i = y$, $K_R(x_i)$, $K_SK_R(x_i)$, or $K_RK_SK_R(x_i)$, we have:*

1. *φ is stable with respect to I.*
2. *If 'sent "φ" ' is in process j's message history at the point (r,m) in I, then $(I,r,m) \models \varphi$.*
3. *If 'received "φ" ' is in process j's message history at the point (r,m) in I, then $(I,r,m) \models K_j\varphi$.*

Proof The stability of $x_i = y$ follows from the assumption that I is a reasonable interpretation. The stability of all the other formulas follows immediately from Lemma 2.

For part 2, we proceed by induction on m. First suppose 'sent "φ" ' is in S's message history at (r,m). The code for protocol A shows that φ is either of the form $x_i = y$ or $K_SK_R(x_i)$. By part 1, both of these formulas are stable. Thus, if 'sent "φ" ' is also in S's message history at $(r,m-1)$, it follows from the induction hypothesis and the stability of φ that $(I,r,m) \models \varphi$. If 'sent "φ" ' is not in S's message history at $(r,m-1)$, this message must have been sent at the point (r,m). To deal with the case that φ is of the form $x_i = y$, observe that for all points (r',m') in I, if $r'(m') = ((X,\ldots),(Sj,q,y,\ldots,i),\ldots)$ and $X = \langle x_0,x_1,\ldots\rangle$, then $y = x_i$. ¿From this it immediately follows that $(I,r,m) \models \varphi$. If φ is of the form $K_SK_R(x_i)$ and this is the first time the message "φ" is sent, then it follows that S is scheduled at (r,m) and S's local state at (r,m) must be of the form $(S2,\ldots,i)$. If $m' < m$ is the previous time that S was scheduled in r, then S's local state at (r,m') must be of the form $(S1,\ldots,i)$. Moreover, it must be the case that $(I,r,m') \models K_SK_R(x_i)$ (otherwise S would have stayed at location S1). Since $K_SK_R(x_i)$ is stable, the result follows. The proof for the case that 'sent "φ" ' is in R's message history is similar and left to the reader.

For part 3, we again proceed by induction on m. Suppose 'received "φ" ' is in S's message history at (r,m) and suppose $(r',m') \sim_S (r,m)$. It follows that 'received "φ" ' is also in S's

message history at (r', m'). It must be the case that 'sent "φ" ' is in R's message history at (r', m'). From part 2, it follows that $(I, r', m') \models \varphi$. Thus $(I, r, m) \models K_S\varphi$. The proof is similar if 'received "φ" ' is in R's message history. ∎

Given a local state for S of the form (Sj, q, y, z, h, i), let the corresponding *reduced state* be (Sj, i). Similarly, given a local state for R of the form (Rk, q', z', h', i'), let the corresponding reduced state be (Rk, i').

Lemma 4 *Let r be a run of $\mathcal{R}$. Then the sequence of reduced states for S in r is either of the form*

$$(S1,0)^+(S2,0)^+(S1,1)^+(S2,1)^+ \ldots (S1,i)^+(S2,i)^+ \ldots$$

or of the form

$$(S1,0)^+(S2,0)^+(S1,1)^+(S2,1)^+ \ldots (Sj,i)^\omega$$

(where $(Sj,i)^+$ denotes a sequence of one or more global states with reduced state (Sj,i), and $(Sj,i)^\omega$ denotes an infinite sequence of global states with reduced state (Sj,i)). Similarly, the sequence of reduced states for R in r is either of the form

$$(R1,0)^+(R2,1)^+(R3,1)^+(R2,2)^+(R3,2)^+ \ldots (R2,i')^+(R3,i')^+$$

or of the form

$$(R1,0)^+(R2,1)^+(R3,1)^+(R2,2)^+(R3,2)^+ \ldots (Rk,i')^\omega.$$

Proof Immediate from the code of A_S and A_R. ∎

We are finally ready to prove liveness. We show that in r is a fair run of $\mathcal{R}$, then in R's reduced states in r include reduced states of the form (Rk, j) for all $j \geq 0$. From Lemma 1, it follows that R writes arbitrarily long prefixes of X, so we get liveness as desired.

Lemma 5 *If $r \in \mathcal{R}$ is such that S and R are scheduled infinitely often in r, and every message sent by S (resp. R) infinitely often is eventually delivered uncorrupted (when R (resp. S) is scheduled), then the sequence of reduced states in r for R is of the form*

$$(R1,0)^+(R2,1)^+(R3,1)^+(R2,2)^+(R3,2)^+(R2,3)^+ \ldots$$

Proof Suppose $r \in \mathcal{R}$ and the sequence of reduced states in r for R does not have the form above. By Lemma 4, it follows that the sequence of reduced states must either have the form $(R1,0)^\omega$, $(R1,0)^+ \ldots (R2,i')^\omega$, or $(R1,0)^+ \ldots (R3,i')^\omega$. By considering each of these cases in turn, we show that they cannot occur in a fair run.

Case 1: the sequence of reduced states in r for R is of the form $(R1,0)^\omega$.

Since R is scheduled infinitely many times in r it follows that for all $m \geq 0$, $(I, r, m) \models \neg K_r(x_0)$ (otherwise R would go to location $R2$). Hence, the sequence of reduced states in r for S does not include $(S2,0)$ (since $(I, r, m) \models \neg K_R(x_0)$ implies that $(I, r, m) \models \neg K_S K_R(x_0)$ and S enters $(S2,0)$ only if $K_S K_R(x_0)$). By Lemma 4 it follows that the sequence of reduced states in r for S is of the form $(S1,0)^\omega$. Assume that $x_0 = 0$. Since S is scheduled infinitely often, it follows that S sends the message $x_0 = 0$ infinitely often. By assumption, the message $x_0 = 0$ is eventually received by R, so it follows that for some $m' \geq 0$, R's history contains "received $x_0 = 0$". By Lemma 3, it follows that $(I, r, m) \models K_R(x_0)$, contradicting our assumption that for all $m \geq 0$, $(I, r, m) \models \neg K_R(x_0)$.

Case 2: the sequence of reduced states in r for R is of the form $(R1,0)^+ \ldots (R2,i')^\omega$.

It follows that for all $m \geq 0$, $(I,r,m) \models \neg K_R K_S K_R(x_{i'})$. Similar reasoning to the previous case shows that the sequence of reduced states in r for S does not contain $(S1,i'+1)$, and therefore is either of the form $(S1,0)^+ \ldots (Sk,j)^\omega$ for some $k \in \{1,2\}$ and $j \leq i'$.

Thus we have four subcases:

Case 2a: $k = 2$ and $j = i'$:

S sends infinitely many "$K_S K_R(x_{i'})$" messages, (by the fairness assumption) at least one of which is delivered to R, and hence for some $m' \geq 0$, "received $K_S K_R(x_{i'})$" will be in R's message history. But, by Lemma 3 this implies that $(I,r,m) \models K_R K_S K_R(x_{i'})$, contradicting our assumption that for all $m \geq 0$, we have $(I,r,m) \models \neg K_R K_S K_R(x_{i'})$.

Case 2b: $k = 1$ and $j = i'$:

In this case, for all $m \geq 0$, $(I,r,m) \models \neg K_S K_R(x_{i'})$. But R sends infinitely many "$K_R(x_{i'})$" messages, at least one of which is delivered to S, and then $K_S K_R(x_{i'})$ holds. This contradicts the assumption that for all $m \geq 0$, $(I,r,m) \models \neg K_S K_R(x_{i'})$.

Case 2c: $k = 1$ and $j < i'$.

In this case, for all $m \geq 0$, $(I,r,m) \models \neg K_S K_R(x_j)$. However, there must be some $m' < m$ such that R's reduced state at (r,m') is $(R2,j)$ and R's reduced state at $(r',m'+1)$ is $(R3,j)$. Hence $(I,r,m') \models K_R K_S K_R(x_j)$. It follows that $(I,r,m') \models K_S K_R(x_j)$. But by Lemma 2, this formula is stable, so that $(I,r,m) \models K_S K_R(x_j)$. This gives us a contradiction.

Case 2d: $k = 2$ and $j < i'$.

In this case, for all $m \geq 0$, $(I,r,m) \models \neg K_S K_R K_S K_R(x_j)$. But R sends infinitely many "$K_R(x_{i'})$" messages, at least one of which is delivered to S, say at time m'. If $(r',\ell) \sim_S (r,m')$, then 'received "$K_R(x_{i'})$" ' is in S's message history in $r'(\ell)$. It follows that that R must have been in reduced state $(R2,i')$ at some point (r',ℓ'), where $\ell' < \ell$. Since $j < i$, there must be some $\ell'' < \ell'$ such that R is in reduced state $(R2,j)$ in (r',ℓ'') and in reduced state $(R3,j)$ in $(r',\ell''+1)$. It follows that $(I,r',\ell'') \models K_R K_S K_R(x_j)$. Since this formula is stable, we must have $(I,r',\ell) \models K_R K_S K_R(x_j)$. Since (r',ℓ) is an arbitrary point with $(r',\ell) \sim_S (r,m')$, it follows that $(I,r,m') \models K_S K_R K_S K_R(x_j)$, contradicting our original assumption.

Case 3: the sequence of reduced states in r for R is of the form $(R1,0)^+ \ldots (R3,i')^\omega$.

It follows that for all $m \geq 0$, $(I,r,m) \models \neg K_R(x_{i'+1})$. Similar reasoning to the previous case show that the sequence of reduced states in r for S does not contain $(S2,i'+1)$, and therefore is of the form $(S1,0)^+ \ldots (Sk,j)^\omega$ for some $k \in \{1,2\}$ and $j \leq i'$ or $k = 1$ and $j = i'+1$. We can now derive a contradiction as before; we leave details to the reader. ∎

As we remarked in Section 3, if S and R just keep track of the number of data elements they have read (resp. written) thus far, an analogue of protocol A, where we replace $K_R(x_i)$ by R wrote x_i, will still work. Since we no longer keep track of the complete history, it is now no longer the case that Lemma 2 holds; even if φ is stable, $K_i(\varphi)$ may not be stable. However, it is still the case that an analogue to Lemma 3 holds. The point is that a formula such as

$K_S(R \text{ wrote } x_i)$ is stable in any set of runs consistent with the modified version of A. This allows us to prove the correctness of the protocol essentially in the same way as we proved the correctness of A.

A.3 Proving the correctness of A^{fs}

In this subsection we prove the correctness of A^{fs}. We have chosen to do this by proving that, in a precise sense, A^{fs} is an *implementation* of A. We define a mapping h from runs of A^{fs} to runs of A that preserves the reading and writing of data elements, and preserves fairness by mapping fair runs of A^{fs} to fair runs of A. Thus safety for A^{fs} follows from safety for A, and liveness for A^{fs} follows from liveness for A. Although our proof will essentially include all the lemmas required for a direct proof of the correctness of A^{fs} by standard techniques, we have chosen this technique since it demonstrates the intrinsic relationship between A and A^{fs}, as well as highlighting the idea of an implementation.

Formally we proceed as follows: Let ρ be a run. A *finite prefix* of ρ is a function with domain $\{0,\ldots,m-1\}$ that agrees with ρ on their common domain. In this case, we say that the *length* of ρ, written $|\rho|$, is m. If $\mathcal{R}$ is a set of runs, let $\mathcal{P}(\mathcal{R})$ consist of the runs in $\mathcal{R}$ together with all the finite prefixes of runs in $\mathcal{R}$. We can define an ordering $\preceq$ on $\mathcal{P}(\mathcal{R})$ by taking $\rho \preceq \rho'$ whenever ρ' extends ρ (so that ρ' has a larger domain and ρ and ρ' agree on their common domain).

We define a function $h: \mathcal{P}(\mathcal{R}(A^{fs})) \rightarrow \mathcal{P}(\mathcal{R}_A^p)$ so that for every $\rho \in \mathcal{P}(\mathcal{R}(A^{fs}))$ the following properties hold:

1. h is *monotonic*, i.e., if $\rho' \preceq \rho$, then $h(\rho') \preceq h(\rho)$.
2. $|h(\rho)| = |\rho|$.
3. If the last state in ρ is of the form

$$((X, c, Y, q, b_S^{fs}, b_R^{fs}, go_S, go_R), (Sj, q', y, z^{fs}), (Rk, q'', z_1^{fs})),$$

then the last state in $h(\rho)$ is of the form

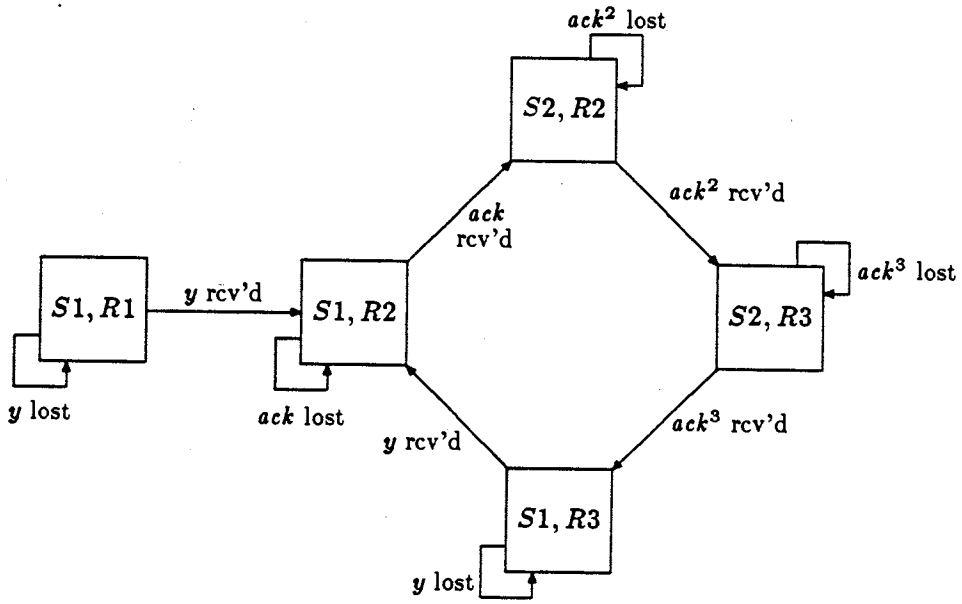
$$((X, c, Y, q, b_S, b_R, go_S, go_R), (Sj, q', y, z, h, c), (Rk, q'', z', h', |Y|)).$$

The definition of h is the obvious one. For example, suppose $\rho \in \mathcal{P}(\mathcal{R}(A^{fs}))$ and S sends the message y in $\rho(m)$. (We can tell this is so if S is in the send phase in $\rho(m)$, $go_S = 1$, and the last message in b_S^{fs} is y.) Then S sends the message "$x_c = y$" in $h(\rho)(m)$, where the value of c is determined by the second component of the environment's state in $\rho(m)$. Since S's state in $h(\rho)$ records the complete message history, we have to append the message "sent $x_c = y$" to h and b_S in $h(\rho)(m)$. Similarly, if S sends ack^2 in ρ, then S sends $K_S K_R(x_c)$ in $h(\rho)$, where again c is determined by the environment's state. If S receives a message ack or ack^3 in $\rho(m)$, then S receives the corresponding message "$K_R(x_c)$" or "$K_R K_S K_R(x_c)$" in $h(\rho)$. If S receives $*$ or λ in $\rho(m)$, then S receives the same message in $h(\rho)(m)$. We omit the straightforward details of the definition of h here.

Let $\mathcal{R} = h(\mathcal{R}(A^{fs}))$, and let $I = (\mathcal{R}, \pi)$ be a reasonable interpreted system. (Recall this means that the formulas $x_c = 1$ and $x_c = 0$ are true exactly where they ought to be.) Our goal

is to show that I is consistent with A. Thus, for example, we must show that when S sends the message "$K_S K_R(x_c)$" at some point in a run $h(\rho)$, then the test $K_S K_R(x_c)$ actually holds.

In order to do that, we characterize the possible runs of A^{fs}. What we intend to show is informally captured in the diagram below, where we focus only on the location counter and ignore the state of the environment. At a high level, what is happening in A^{fs} is that essentially S and R proceed in lockstep, with R at most one step behind S.



Our first step in characterizing the runs of A is to establish some invariants.

Lemma 6 *Assume $r \in \mathcal{R}(A^{fs})$. Then for all $m \geq 0$, if $r(m)$ is of the form*

$$((X, c, Y, q'', b_S, b_R, go_S, go_R), (Sj, q, y, z), (Rk, q', z')),$$

where $X = \langle x_0, x_1, x_2, \ldots \rangle$, then the following all hold:

1. $y = x_c$.
2. *If $go_S = 1$ then $q' = q$; if $go_S = 0$ then $q' = s$. Similarly, if $go_R = 1$ then $q'' = q$; if $go_R = 0$ then $q'' = s$.*
3. *If $j = 1$ then either*
   - *(a) $k = 1$, $c = |Y| = 0$, or*
   - *(b) $k = 2$ and $|Y| = c + 1$, or*
   - *(c) $k = 3$ and $|Y| = c$*
4. *If $j = 2$ then $|Y| = c + 1$ and either $k = 2$ or $k = 3$.*
5. *If $q \in \{r, l\}$ then*
   - *(a) If $j = 1$ and $k = 1$ then $b_S \in \{\langle\rangle, \langle y \rangle\}$ and $b_R = \langle\rangle$. If $go_S = 1$ then $y \in b_S$.*

(b) *If* $j = 1$ *and* $k = 2$ *then* $b_S = \langle y \rangle$. *If* $c = 0$ *then* $b_R \in \{\langle\rangle, \langle ack \rangle\}$, *otherwise* $b_R \in \{\langle ack \rangle, \langle ack^3 \rangle, \langle ack^3, ack \rangle\}$. *If* $go_R = 1$ *then* $ack \in b_R$.

(c) *If* $j = 2$ *and* $k = 2$ *then* $b_S \in \{\langle y \rangle, \langle ack^2 \rangle, \langle y, ack^2 \rangle\}$ *and* $b_R = \langle ack \rangle$. *If* $go_S = 1$ *then* $ack^2 \in b_S$.

(d) *If* $j = 1$ *and* $k = 3$ *then* $b_S \in \{\langle ack^2 \rangle, \langle y \rangle, \langle ack^2, y \rangle\}$ *and* $b_R = \langle ack^3 \rangle$. *If* $go_S = 1$ *then* $y \in b_S$.

(e) *If* $j = 2$ *and* $k = 3$ *then* $b_S = \langle ack^2 \rangle$ *and* $b_R \in \{\langle ack^3 \rangle, \langle ack \rangle, \langle ack, ack^3 \rangle\}$. *If* $go_R = 1$ *then* $ack^3 \in b_R$.

6. *If* $q = l$ *and* $go_S = 1$ *(resp.* $go_R = 1$*) then* $z \in b_R$ *(resp.* $z' \in b_S$*).*

Proof We proceed by induction on m to show that all of the properties above hold for the global state $r(m)$. The base case follows immediately from our assumption about the form of the initial global states. For the inductive step, assume (1)–(6) are true for $r(m')$, $m' \leq m$. We have to show that any joint action that could have been taken from $r(m)$ would lead to an $r(m+1)$ that satisfies (1)–(6). There are lots of possibilities of $r(m)$ to consider: six possible combinations of j and k, three possible values of q, four combinations of go_S and go_R, and several combinations of b_S and b_R. The argument in all cases is similar; we present two of the cases here.

First consider the case where $q = s$, and $j = k = 1$, $go_S = go_R = 1$. By the induction hypothesis, in $r(m)$ we have $c = |Y| = 0$ and $y = x_0$. The joint action taken from $r(m)$ is of the form $a = (a_e, a_S, a_R)$, where a_S = send y, a_R = send λ, and $a_e = \Lambda$. Therefore, $r(m+1) = \tau(a)(r(m))$ is of the form

$$((X, 0, \langle\rangle, r, \langle x_0 \rangle, \langle\rangle, 1, 1), (S1, r, x_0, z), (R1, r, z')),$$

and hence $r(m+1)$ satisfies (1)–(6).

Next suppose $q = l$, $j = 1$, $k = 3$, $go_S = 1$, and $go_R = 0$. By the induction hypothesis, $r(m)$ is of the form

$$((X, c, \langle x_0, \ldots, x_{c-1} \rangle, l, b_S, \langle ack^3 \rangle, 1, 0), (S1, l, x_c, z), (R3, l, z')),$$

where $b_S \in \{\langle x_c \rangle, \langle ack^2, x_c \rangle\}$ and $z \in b_R$ (i.e., $z \neq ack^2$). The joint action taken from $r(m)$ is therefore of the form $a = (a_e, a_S, a_R)$, where a_S is "go to S1", a_R is irrelevant (since $go_R = 0$ and a_R will have no effect) and $a_e = \Lambda$. Thus $r(m+1) = r(m)$, and so satisfies (1)–(6) by the induction hypothesis. ∎

Note that it is easy to add the clause $Y \preceq X$ to the six invariant clauses above, and thus show that A^{fg} has the safety property.

For any global state $g \in \mathcal{G}^{fg}$ of the form $((X, c, Y, q'', b_S, b_R, go_S, go_R), (Sj, q, y, z), (Rk, q', z'))$ we define the *reduced global state of* g to be the tuple (c, Sj, Rk). Parts 3 and 4 of Lemma 6 tell us that the only possible global states that arise are those of the form $(0, S1, R1)$, $(c, S1, R2)$, $(c, S2, R2)$, $(c, S2, R3)$, and $(c, S1, R3)$, for $c \geq 0$. For convenience, we abbreviate these as v_0, v_{4c+1}, v_{4c+2}, v_{4c+3}, and v_{4c+4}, respectively. The following analogue to Lemma 4 says that the diagram above does indeed capture all the possible state transitions.

Lemma 7 *Let $r \in \mathcal{R}(A^{fs})$. Then the sequence of reduced global states of r is either of the form $v_0^+ v_1^+ v_2^+ \ldots$ or of the form $v_0^+ v_1^+ \ldots v_n^\omega$. Moreover, the transition from v_{4c} to v_{4c+1} (resp. v_{4c+1} to v_{4c+2}, v_{4c+2} to v_{4c+3}, v_{4c+3} to v_{4c+4}), for $c \geq 0$, occurs exactly when R receives y (resp. S receives ack, R receives ack^2, S receives ack^3).*

Proof We proceed by induction on m to show that that if ρ is a prefix of r of length m, then the sequence of reduced global states of ρ has the form $v_0^+ \ldots v_n^+$ for some n. By our assumption on the form of initial global states, the base case holds. For the inductive step, assume that ρ' is a prefix of r of length m, and that the sequence of reduced global states in ρ' is $v_0^+ v_1^+ \ldots v_n^+$. Let ρ be the prefix of r of length $m+1$. We want to show that the reduced global state of $\rho(m+1)$ is either v_n or v_{n+1}, and that the latter case occurs only if the appropriate message is received. Consider the case where v_n is of the form $(c, S1, R2)$. By the semantics of A^{fs}, the first component of the reduced global state cannot change (since S does not perform a read action in $S1$), the second component can change only if S is in the local computation phase and receives an ack message from R, in which case it changes to $S2$, and the third component can change only if R is in the local computation phase and receives an ack^2 message from S. By part 5(b) of Lemma 6, R cannot receive an ack^2, since $ack^2 \notin b_S$ at this point. The result follows in this case. The other four cases are similar and left to the reader. This establishes the inductive step. ∎

Note that at this point we could directly prove the liveness property for A by showing that for fair runs the sequence of reduced global states must be of the form $v_0^+ v_1^+ \ldots$, much as we did in Lemma A.2. However, we proceed with our proof that A^{fs} implements A.

Given a run $r \in \mathcal{R}(A^{fs})$, we call the point (r, m) a *transition point* if the reduced global state at (r, m) and $(r, m+1)$ is different. Recall that $\mathcal{R} = \{h(r) | r \in \mathcal{R}(A^{fs})\}$.

Lemma 8 *Suppose $r \in \mathcal{R}(A^{fs})$, (r, m) is a transition point, the reduced global state of $r(m)$ is v_{4c+k} for some $c \geq 0$, $0 \leq k \leq 3$, and $I = (\mathcal{R}, \pi)$ is a reasonable interpreted system. Then*

1. *if $k = 0$ then $(I, h(r), m) \models K_R(x_c) \wedge \neg K_S K_R(x_c)$ and for all $m' < m$, we have $(I, h(r), m') \models \neg K_R(x_c)$*
2. *if $k = 1$ then $(I, h(r), m) \models K_S K_R(x_c) \wedge \neg K_R K_S K_R(x_c)$ and for all $m' < m$, we have $(I, h(r), m') \models \neg K_S K_R(x_c)$*
3. *if $k = 2$ then $(I, h(r), m) \models K_R K_S K_R(x_c) \wedge \neg K_S K_R K_S K_R(x_c)$ and for all $m' < m$, we have $(I, h(r), m') \models \neg K_R K_S K_R(x_c)$*
4. *if $k = 3$ then $(I, h(r), m) \models K_S K_R K_S K_R(x_c) \wedge \neg K_R(x_{c+1})$ and for all $m' < m$, we have $(I, h(r), m') \models \neg K_S K_R K_S K_R(x_c)$.*

Proof We proceed by induction on m. All cases are similar; we consider one representative case here. Suppose $c > 0$ and $k = 0$. Since (r, m) is a transition point, it follows from Lemma 7 that R received the message y at this point in r. For definiteness, suppose that at this point $y = 0$. From Lemma 6, it follows that $x_c = 0$. By definition of h, R receives the message "$x_c = 0$" at this point in $h(r)$ and appends this message to its message history. ¿From Lemma 3, it follows that $(I, h(r), m) \models K_R(x_c = 0)$, so we also have $(I, h(r), m) \models K_R(x_c)$ (since $K_R(x_c)$ is an abbreviation for $K_R(x_c = 0) \vee K_R(x_c = 1)$).

To prove that $(I, h(r), m) \models \neg K_S K_R(x_c)$, consider a run $r' \in \mathcal{R}(A^{fs})$ that agrees with r up to time $m-1$, but R does not receive 0 at $r'(m)$. We clearly have $(h(r), m) \sim_S (h(r'), m)$ and $(I, h(r'), m) \models \neg K_R(x_c = 0)$. We clearly also have $(I, h(r'), m) \models \neg K_R(x_c = 1)$, so that $(I, h(r'), m) \models \neg K_R(x_c)$. Thus $(I, h(r), m) \models \neg K_S K_R(x_c)$.

It remains to show that for every $m' < m$, we have $(I, h(r), m') \not\models K_R(x_c)$. Let $r' \in \mathcal{R}(A^{fs})$ be just like r except that $x_c = 1$ in r'. From Lemma 7 it follows that R does not receive the value of x_c before time m in r, thus we have $(h(r), m') \sim_R (h(r'), m')$ for all $m' < m$. Since $(I, h(r'), m') \models x_c = 1$ for all $m' < m$, we have $(I, h(r), m') \models \neg K_R(x_c = 0)$ for all $m' < m$. And since $(I, h(r), m') \models x_c = 0$, we also have $(I, h(r), m') \models \neg K_R(x_c = 1)$ for all m'. Thus $(I, h(r), m') \models \neg K_R(x_c)$ for all $m' < m$. ∎

We are now ready to establish

Lemma 9 *If $\mathcal{R} = h(\mathcal{R}(A^{fs}))$ and $I = (\mathcal{R}, \pi)$ is a reasonable interpreted sytem, then I is consistent with A.*

Proof Let $h(r) \in \mathcal{R}$. We have to show that for all $m \geq 0$, if $h(r)(m) = (s_e, s_S, s_R)$ then there is a joint action $(a_e, a_S, a_R) \in A_e(s_e, I) \times A_S(s_S, I) \times A_R(s_R, I)$ such that $h(r)(m+1) = \tau(a_e, a_S, a_R)(r(m))$. The proof is by induction on m. If in $h(r)(m)$ the environment is in either the send or the receive phase, then, since $r \in A^{fs}$, the claim is trivial. We therefore consider only the case that in $h(r)(m)$ the environment is in the local phase (i.e., $m = 2 \bmod 3$). But now the claim follows immediately from the semantics of A and Lemma 8. ∎

Since the definition of h guarantees that for every run in $r \in \mathcal{R}(A^{fs})$, the same read and write actions are performed at the same times in r and $h(r)$, and since I is consistent with A, it follows from Theorem 3.1 that every run in $\mathcal{R}(A^{fs})$ has the safety property. It is also easy to see that the mapping h also preserves fairness (if every message in a run $r \in \mathcal{R}(A^{fs})$ that is sent infinitely many times is eventually delivered uncorrupted to the recipient when the latter is scheduled, then the same is true in $h(r)$). Since Theorem 3.1 guarantees that fair runs of $\mathcal{R}$ have the liveness property, the same is true of fair runs in $\mathcal{R}(A^{fs})$.

This completes the proof of Theorem 3.2. ∎

A.4 Semantics and correctness of A^{di}, A^{mi}, and A^{dm}

In this subsection we prove Theorems 3 and 4. We start by constucting the appropriate set of runs for A^{di}. The construction very similar to that for A^{fs}. We take L_S to consist of states of the from (Sj, u, q, y, z_1, z_2) where $j \in \{1,2\}$, $u \in \{a,b\}$, $q \in \{s,r,l\}$, $y \in \{0,1\}$, $z_1 \in \{\lambda, e_1^{di}(ack), e_1^{di}(ack^3)\}$, and $z_2 \in \{\lambda, e_2^{di}(ack), e_2^{di}(ack^3)\}$. Similarly, L_R consists of states of the form (Rk, u', q', z_1', z_2'). The set L_e consists of states of the form (X, c, Y, q'', b_S, b_R), where the components have the same meaning as in A^{fs}, except we now omit the go_S and go_R components since we are dealing with a synchronous system where processes are continuously enabled, and we now take b_S (resp. b_R) to consist of the last message sent by S (resp. R).

The actions that S and R can perform when running A^{di} are analogous to those they perform when running A^{fs}. The environment's actions are somewhat more restricted, since we

only allow deletion and insertion errors. The environment performs the null action Λ in the send phase and the local computation phase. At the receive phase, its actions have the form (a_S, b_R), where a and b are one of $ins0$, $ins1$, del, or Λ. The effect of $ins0_S$ is that R receives 0 if S sent λ; otherwise R receives whatever S sent. The effect of $ins1_S$ is similar. The effect of del_S is that R receives λ (i.e., S's message, if there is one, is deleted), while the effect of Λ_S is that R receives whatever S sent. The effect of actions of the form b_R is analogous. We can now define the transition function τ is defined in the obvious way. We take $\mathcal{G}_0$, the set of initial states, to be the set of all states of the form

$$((X, 0, \langle\rangle, s, \lambda, \lambda), (S1, a, s, x_0, \lambda, \lambda), (R1, a, s, \lambda, \lambda)),$$

and then we can get $\mathcal{R}(A^{di})$, the set of runs that are consistent with A^{di}. We leave details to the reader.

The idea of the correctness proof for A^{di} is straightforward. We show that A^{di} is an implementation of A^{fs}, much as we showed that A^{fs} is an implentation of A (although the details are simpler in this case). We again define a mapping from runs of A^{di} to runs of A^{fs} that preserves the reading and writing of data elements. (Since every two rounds of A^{di} correspond to one round of A^{fs} in an obvious way, this mapping is easy to define.) Moreover, the mapping preserves fairness, in that it maps fair runs of A^{di} to fair runs of A^{fs}. Thus safety for A^{di} follows from safety for A^{fs}, and liveness for A^{di} follows from liveness for A^{fs}.

More formally, we have a function $h^{fs}: \mathcal{P}(\mathcal{R}(A^{di})) \rightarrow \mathcal{P}(\mathcal{R}(A^{fs}))$ such that for every $\rho \in \mathcal{P}(\mathcal{R}(A^{di}))$ the following properties hold:

1. h^{fs} is monotonic, so if $\rho' \preceq \rho$ then $h^{fs}(\rho') \preceq h^{fs}(\rho)$.
2. If $|\rho| = 6m + j$, $j \in \{1, \ldots, 5\}$, then $h^{fs}(\rho) = h^{fs}(\rho')$, where ρ' is the prefix of ρ of length $6m$.
3. If $|\rho| = 6m$ and $\rho(6m)$ is of the form

$$((X, c, Y, s, b_S, b_R, 1, 1), (Sj, a, s, y, z_1, z_2), (Rk, a, s, z_1', z_2')),$$

then $|h^{fs}(\rho)| = 3m$ and $h^{fs}(\rho)(3m)$ is of the form

$$((X, c, Y, s, b_S', b_R', 1, 1), (Sj, s, y, z), (Rk, s, z')),$$

where

$$z = \begin{cases} m & \text{if } e^{di}(m) = z_1 z_2,\ m \in \{ack, ack^3\} \\ \lambda & \text{otherwise} \end{cases}$$

and

$$z' = \begin{cases} m & \text{if } e^{di}(m) = z_1 z_2,\ m \in \{0, 1, ack^2\} \\ \lambda & \text{otherwise.} \end{cases}$$

Property 3 implies that the two parts of the message $e^{di}(m)$ sent at rounds $6m - 6$ and $6m - 3$ of ρ are both received iff the message m sent in round $3m - 3$ of $h^{fs}(\rho)$ is received. In particular, this means that when we pass to infinite runs (using the monotonicity guaranteed by property 1), we get that r is a fair run of A^{di} iff $h^{fs}(r)$ is a fair run of A^{fs}.

We first define h^{fs} for finite prefixes in $\mathcal{P}(\mathcal{R}(A^{di}))$ by induction on the length of the prefix. If $\rho \in \mathcal{P}(\mathcal{R}(A^{di}))$ and $|\rho| = 0$, then $\rho(0)$ must have the form

$$((X, 0, \langle\rangle, s, \langle\rangle, \langle\rangle), (S1, a, s, x_0, \lambda, \lambda), (R1, a, s, \lambda, \lambda)).$$

We take $h^{fs}(\rho)$ to be the prefix of length zero such that $h^{fs}(\rho)(0)$ has the form

$$((X, 0, \langle\rangle, s, \langle\rangle, \langle\rangle, 1, 1), (S1, s, x_0, \lambda, \lambda), (R1, s, \lambda, \lambda)).$$

It is immediate that this is indeed a legitimate initial state of A^{fs}, so $h^{fs}(\rho)$ is well-defined. Clearly it also satisfies property 3 above.

For the inductive step, suppose we have defined h^{fs} on all prefixes of length $m \geq 0$ and $\rho \in \mathcal{P}(\mathcal{R}(A^{di}))$ has length $m+1$. Take m' such that $m+1 = 6m' + j$, for $j \in \{1, \ldots, 6\}$. If $j < 6$, then we define $h^{fs}(\rho) = h^{fs}(\rho')$, where ρ' is the prefix of ρ of length m. Clearly this satisfies all the requirements. If $j = 6$, consider the prefix ρ' of ρ of length $6m'$. It is easy to see that for all runs $r \in \mathcal{R}(A^{di})$ and all n, we have that $r(6n)$ has the form

$$((X, c, Y, s, b_S, b_R), (Sj, a, s, y, z_1, z_2), (Rk, a, s, z_1', z_2')).$$

In particular, $\rho'(6m')$ and $\rho(6m'+6)$ must also have this form. By the induction hypothesis, $|h^{fs}(\rho')| = 3m'$ and $h^{fs}(\rho')(3m')$ is of the form

$$((X, c, Y, s, b_S', b_R', 1, 1), (Sj, s, y, z), (Rk, s, z')).$$

Observe that once we decide what action the environment takes at step $3m'+1$, we get a unique extension of $h^{fs}(\rho)$ to a prefix ρ'' of a run of A^{fs} of length $3m'+3$. (This is because S's and R's actions are a deterministic function of their state, and in step $3m$ and $3m+2$, then environment performs the empty action Λ.) We construct ρ'' so that $\rho''(3m'+3)$ and $\rho(6m'+6)$ satisfy property 3 of h^{fs} above. Thus the environment deletes S's message at step $3m+1$ of ρ'' precisely if it deletes S's message in either step $\rho(6m'+1)$ or $\rho(6m'+4)$, and it deletes R's message precisely if it deletes R's message in either step $\rho(6m'+1)$ or $\rho(6m'+4)$. Take $h^{fs}(\rho) = \rho''$. We must show that this choice indeed satisfies property 3 of the induction hypothesis. In particular, we must show that S and R are in corresponding locations in the global states $\rho(6m'+6)$ and $h^{fs}(\rho)(3m'+3)$. To do this, we must consider all six combinations of pairs of locations for R and S in $\rho(6m')$ and for each of these, consider the sixteen possible actions the environment can perform at step $6m'+1$ and sixteen four possible actions it can perform at step $6m'+4$. Since any insert action performed by the environment has no effect, most of the cases are identical. We omit the straightforward (but tedious) check here.

We have now defined h^{fs} on finite prefixes of runs in $\mathcal{R}(A^{di})$. We extend it to runs in the unique way required to preserve monotonicity. Note that properties 2 and 3 of h^{fs} hold vacuously for infinite runs.

Since we have proved the safety property for A^{fs}, and properties 1, 2, and 3 of h^{fs} together guarantee that for all runs $r \in \mathcal{R}(A^{di})$, the sequence of reads and writes in r is the same as that in $h^{fs}(r)$, the safety property of A^{di} immediately follows.

In order to prove liveness, we again have to define what it means for a run $r \in \mathcal{R}(A^{di})$ to be consistent with an infinite sequence of actions. The definition is analogous to that for runs in

$\mathcal{R}(A^{fs})$, so we omit details here. We say that $r \in \mathcal{R}(A^{di})$ is a run where *infinitely many 2-block messages from S to R (resp. R to S) are delivered* if r is consistent with a sequence of actions $\alpha_0, \alpha_1 \ldots$ such that for infinitely many even i's, neither α_{2i} nor α_{2i+1} has first component del_S (resp. second component del_R).

As we remarked above, h^{fs} preserves fairness: h^{fs} maps fair runs of A^{di} (fair in the sense that infinitely many 2-block messages from S to R and from R to S are delivered) to fair runs of A^{fs} (fair in the sense that infinitely many messages from S to R and from R to S are delivered). Thus, liveness for A^{di} follows immediately from liveness for A^{fs}.

This completes the proof of Theorem 3. ∎

Theorem 4 follows by extending the methods of Theorem 3 to deal with A^{mi} and A^{dm}. Since in both these cases the environment can mutate messages, we must allow the environment additional actions of the form mut_S and mut_R, where mut_S results in R getting a 0 if S sent a 1 and R getting a 1 if S sent a 0, and mut_R is defined similarly. However, in A^{mi} the environment cannot the actions del_S and del_R, while in A^{dm} the environment cannot perform actions of the form $ins0_S$. We leave the straightforward details to the reader. ∎

Acknowledgements: We would particularly like to thank Brent Hailpern, who started us thinking about the AUY protocols. We also thank Ron Fagin, Mike Fischer, Brent Hailpern, Yoni Malachi, Moshe Vardi, and Yoram Moses for their insightful comments on an earlier draft of this paper, Eli Gafni and Laura Haas for pointing out references [GS80] and [SH86], Moshe Vardi for pointing out reference [BG77] (and the quote about the use of knowledge that appears there), and Josh Cohen Benaloh for LaTeX help.

References

[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, 1982, pp. 205–214. This is a later version of [AUY79].

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

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

[BS80] G. V. Bochmann and C. A. Sunshine, Formal methods in communication protocol design, *IEEE Transactions on Communications* **COM-28**, 1980, pp. 624–631.

[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**, 1969, pp. 260–261.

[CES83] E. M . Clarke, E. A. Emerson, and A. P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications: a practical approach, *Proc. 10th ACM Symp. on Principles of Programming Languages*, 1983, pp. 117–126.

[CM86] K. M. Chandy and J. Misra, How processes learn, *Distributed Computing* **1**:1, 1986, pp. 40–52.

[DM86] C. Dwork and Y. Moses, Knowledge and common knowledge in a Byzantine environment I: crash failures (extended abstract), *Theoretical Aspects of Reasoning about Knowledge: Proceedings of the 1986 Conference* (J. Y, ed.), Morgan Kaufmann, 1986, pp. 149–170.

[FHV86] R. Fagin, J. Y. Halpern, and M. Y. Vardi, What can machines know? On the epistemic properties of machines, *Proc. of AAAI-86*, 1986, pp. 428–434.

[FI86] M. J. Fischer and N. Immerman, Foundations of knowledge for distributed systems, *Theoretical Aspects of Reasoning about Knowledge: Proceedings of the 1986 Conference* (J. Y, ed.), Morgan Kaufmann, 1986, pp. 171–186.

[GA87] E. Gafni and Y. Afek, Communication in unreliable networks, unpublished manuscript, 1987.

[Gaf86] E. Gafni, Perspectives on distributed network protocols: a case for building blocks, *MILCOM '86*, 1986.

[Gou85] M. Gouda, On "A simple protocol whose proof isn't", *IEEE Transactions on Communications* **COM-33**:4, 1985, pp. 382–384.

[GS80] V. D. Gligor and S. H. Shattuck, On deadlock detection in distributed systems, *IEEE Transactions on Software Engineering* **SE-6**:5, 1980, pp. 435–440.

[Had87] V. Hadzilacos, A knowledge-theoretic analysis of atomic commitment protocols, *Proc. 6th ACM Symp. on Principles of Database Systems*, 1987.

[Hai82] B. T. Hailpern, *Verifying concurrent processes using temporal logic*, *Lecture Notes in Computer Science, Vol. 129*, Springer-Verlag, 1982.

[Hai85] B. T. Hailpern, A simple protocol whose proof isn't, *IEEE Transactions on Communications* **COM-33**:4, 1985, pp. 330–337.

[Hal87] J. Y. Halpern, Using reasoning about knowledge to analyze distributed systems, *Annual Review of Computer Science, Vol. 2*, Annual Reviews Inc., 1987.

[HF85] J. Y. Halpern and R. Fagin, A formal model of knowledge, action, and communication in distributed systems: preliminary report, *Proc. 4th ACM Symp. on Principles of Distributed Computing*, 1985, pp. 224–236.

[HF87] J. Y. Halpern and R. Fagin, *Modelling knowledge, action, and communication in distributed systems*, Technical Report, IBM, to appear, 1987.

[HM84] J. Y. Halpern and Y. Moses, Knowledge and common knowledge in a distributed environment, *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, 1984, pp. 50–61. A revised version appears as *IBM Research Report RJ 4421*, Aug., 1987.

[HO83] B. T. Hailpern and S. S. Owicki, Modular verification of communication protocols, *IEEE Transactions on Communications* **COM-31**:1, 1983, pp. 56–68.

[HV86] J. Y. Halpern and M. Y. Vardi, The complexity of reasoning about knowledge and time, *Proc. 18th ACM Symp. on Theory of Computing*, 1986, pp. 304–315.

[LR86] R. Ladner and J. Reif, The logic of distributed protocols (preliminary report), *Theoretical Aspects of Reasoning about Knowledge: Proceedings of the 1986 Conference* (J. Y, ed.), Morgan Kaufmann, 1986, pp. 207–222.

[Mer76] P. M. Merlin, A methodology for the design and implementation of communication protocols, *IEEE Transactions on Communications* **COM-24**:4, 1976, pp. 614–621.

[MT86] Y. Moses and M. Tuttle, Programming simultaneous actions using common knowl-

edge, *Proc. 27th IEEE Symp. on Foundations of Computer Science*, 1986, pp. 208–221.

[NT87] G. Neiger and S. Toueg, Substituting for real time and common knowledge in asynchronous distributed systems, *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 1987, pp. 281–293.

[OL82] S. Owicki and L. Lamport, Proving liveness properties of concurrent programs, *ACM Trans. on Programming Languages and Systems* **4**:3, 1982, pp. 455–495.

[Pnu77] A. Pnueli, The temporal logic of programs, *Proc. 18th IEEE Symp. on Foundations of Computer Science*, 1977, pp. 46–57.

[PR85] R. Parikh and R. Ramanujam, Distributed processes and the logic of knowledge, *Proc. of the Workshop on Logics of Programs*, 1985, pp. 256–268.

[SH86] S. R. Soloway and P. A. Humblet, *On distributed network protocols for changing topologies*, Technical Report LIDS-P-1564, MIT, 1986.

[SM82] R. L. Schwartz and P. M. Melliar-Smith, From state machines to Temporal Logic: specification methods for protocol standards, *IEEE Transactions on Communications*, 1982.

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

[Sun79] C. A. Sunshine, Formal techniques for protocol specification and verification, *IEEE Computer* **12**, 1979, pp. 20–27.