Correctness of At-Most-Once Message Delivery Protocols

Butler W. Lampson[a], Nancy A. Lynch[b] and Jørgen F. Søgaard-Andersen[c*]

[a]Cambridge Research Laboratory, DEC, Cambridge, MA 02139, USA.

[b]Lab. for Computer Science, MIT, 545 Tech. Square, Cambridge, MA 02139, USA.

[c]Department of Computer Science, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark.

This paper addresses the issues of formal description and verification for communication protocols. Specifically, we present the results of a project concerned with proving correctness of two different solutions to the at-most-once message delivery problem. The two implementations are the well-known five-packet handshake protocol and a timing-based protocol developed for networks with bounded message delays.

We use an operational automaton-based approach to formal specification of the problem statement and the implementations, plus intermediate levels of abstraction in a step-wise development from specification to implementations. We use simulation techniques for proving correctness.

In the project we deal with safety, timing, and liveness properties. In this paper, however, we concentrate on safety and timing properties.

Keyword Codes: C.2.2; D.2.4; F.1.1
Keywords: Computer Systems Organization, Network Protocols; Software, Program Verification; Theory of Computation, Models of Computation

1. INTRODUCTION

During the past few years, the technology for formal verification of communication protocols has matured to the point where we believe that it now provides practical assistance for protocol design and validation. Recent advances include the development of formal models that allow reasoning about timed systems as well as untimed systems, e.g., [2, 5, 13], and the development of simulation techniques (including refinement mappings and forward and backward simulations) for proving that one protocol implements another, e.g., [1, 5–7, 13]. In this paper, we show how these techniques can be used to verify an important class of communication protocols – those for *at-most-once message delivery*. The goals of our project are twofold: to provide better understanding, documentation and proof for these protocols, and to test the adequacy of the models and proof techniques.

The *at-most-once message delivery* problem is that of delivering a sequence of messages submitted by a user at one location to a user at another location. Ideally, we would like to insist that all messages be delivered in the order in which they are sent, each exactly once, and that an acknowledgement be returned for each delivered message.[1]

*Research supported in part by the Danish Research Academy.

[1]Our definition of at-most-once message delivery is different from what some people call at-most-once message delivery in that we include acknowledgements and require messages to be delivered in order.

Unfortunately, it is expensive to achieve these goals in the presence of failures (e.g., node crashes and timing anomalies). In fact, it is impossible to achieve them at all unless some change is made to the stable state (i.e., the state that survives a crash) each time a message is delivered. To permit less expensive solutions, we weaken the statement of the problem slightly. We allow some messages to be lost when a node crash occurs; however, no messages should otherwise be lost, and those messages that are delivered should not be reordered or duplicated. (The specification is weakened in this way because message loss is generally considered to be less damaging than duplicate delivery.) Now it is required that the user receive either an acknowledgement that the message has been delivered, or in the case of crashes, an indication that the message might have been lost.

There are various ways to solve the at-most-once message delivery problem. All are based on the idea of tagging a message with an identifier and transmitting it repeatedly to overcome the unreliability of the channel. The receiver[2] keeps a stock of "good" identifiers that it has never accepted before; when it sees a message tagged with a good identifier, it accepts it, delivers it, and removes that identifier from the set. Otherwise, the receiver just discards the message, perhaps after acknowledging it. Different protocols use different methods to keep the sender and the receiver more or less in agreement about what identifiers to use.

A desirable property, which is not directly related to correctness, is that the implementations offer a way of cleaning up "old" information when this cannot affect the future behavior.

In this work, we consider two protocols that are important in practice: the clock-based protocol of Liskov, Shrira and Wroclawski [10] and the five-packet handshake protocol of Belsnes [3]. The latter is the standard protocol for setting up network connections, used in TCP, ISO TP-4, and many other transport protocols. It is sometimes called the three-way handshake, because only three packets are needed for message delivery; the additional packets are required for acknowledgement and cleaning up the state. The former protocol was developed as an example to show the usefulness of clocks in network protocols [9] and has been implemented at M.I.T. Both protocols are sufficiently complicated that formal specification and proof seems useful.

The basic model we use is based on the (*timed*) *automaton* model of Lynch and Vaandrager [13] with an extra added component to express liveness [5]. We express both protocols, as well as the formal specification of the at-most-once message delivery problem, in terms of this model. In the project we carry out complete correctness proofs for both protocols. Some highlights of our proofs are as follows:

Although the two protocols appear to be quite different, we have found that both can be expressed formally as implementations of a common *generic protocol* G, which, in turn, is an implementation of the problem specification. To prove that G implements the specification, for proof-technical reasons we introduce an additional level of abstraction, the *delayed-decision specification* D. This is depicted in Figure 1. Introducing intermediate levels of abstraction, like G and D, is a general proof strategy that allows large,

[2]We denote by "receiver" the protocol entity that is situated on the receiver node, and use phrases like "the user at the receiver end" to denote the user that communicates with the receiver. Correspondingly for "sender".

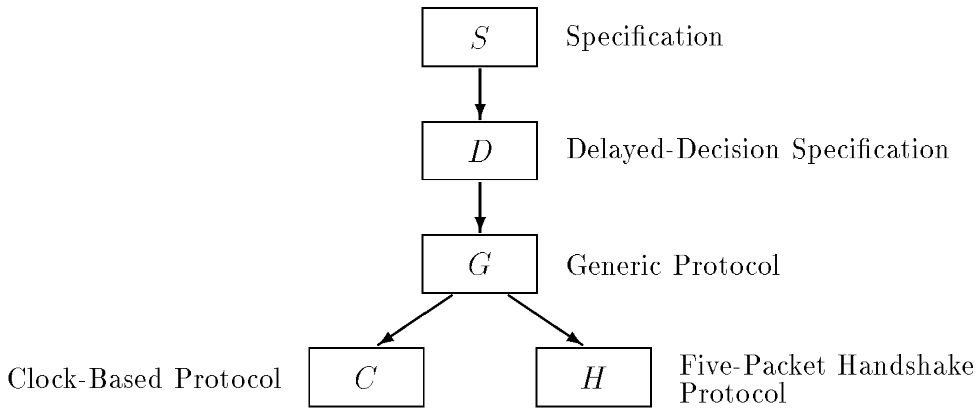

Figure 1. Overview of the levels of abstraction in our work.

complicated proofs to be split into smaller and more managerable subproofs.

The proof that an implementation correctly implements a specification is divided into two parts. First, a simulation technique is applied to show that the implementation *safely* implements the specification, i.e., that all safety (and timing) properties of the implementation are allowed by the specification. Then, heavily based on the simulation result, we prove liveness properties, and thus, correctness.

As proof techniques we use a backward simulation to show that D safely implements the specification, and forward simulations to show that each of the two protocols safely implements G and that G safely implements D. Because of space limitations, we only treat safe implementation in this paper. Our intention with this paper is to give an overview of our work and to convey our experiences with specifying and verifying practical communication protocols. For this reason we have left out many formal details. We refer to our full report [8] for such details. The full report also contains an exhaustive treatment of liveness properties of the protocols.

The rest of the paper is organized as follows. We start in Section 2 by giving an introduction to our model. Then in Sections 3 and 4 we present the problem specification and the low-level protocol C. In this paper we will only briefly deal with the H protocol. In Section 5 we show how aspects of both low-level protocols can be captured in the generic protocol G, and in Section 6 we outline the correctness proofs. In doing so we present the delayed-decision specification D. Finally, we give concluding remarks in Section 7.

2. THE UNDERLYING THEORY

The general model we use to specify safety and timing properties at all levels of abstraction is based on the (*timed*) *automaton* model of [5, 13] and the I/O automaton model of [11, 12]. An *automaton* is a state machine with named *actions* associated with its transitions. Thus, an automaton consists of

- a (possibly infinite) set of *states*. In timed systems a *now* component indicates the time,

- a (nonempty) set of *start states*,
- a set of *actions* partitioned into *visible actions* (which are furthermore partitioned into *input* actions and *output* actions), *internal actions* (which are invisible from the environment of the system), and, for timed systems, a special *time-passage action*, and
- a transition relation of ((pre-)state, action, (post-)state) triples. Each triple is called a *step*.

For timed systems, the steps involving the time-passage action have to satisfy certain axioms that give natural properties of real time, e.g., that time cannot go backwards.

Correctness of an automaton is specified in terms of its *external behavior* given by the set of *traces*, each of which consists of a sequence of visible actions that the automaton can perform. In timed systems these actions are furthermore paired with their time of occurrence to form *timed traces*. One automaton, A, *safely implements* another automaton, B, if the set of (timed) traces of A is included in that of B.

The papers [6, 13, 5] present a collection of *simulation* proof techniques (refinement mappings, forward and backward simulations, etc.) for showing that one (timed) automaton safely implements another. Each of these techniques involves describing a relation between the states of an implementation automaton and those of a specification automaton. Each technique requires a particular type of correspondence between initial states of the two automata, as well as a particular type of correspondence between their steps. Figure 2 illustrates how each step of the implementation automaton must correspond to a (possibly empty) sequence of steps of the specification automaton *containing exactly the same visible actions*.

The main distinction between forward and backward simulations, which are abstract representations of *history* [16] and *prophecy* [1] variables, respectively, lies in the way these corresponding sequences of steps of the specification are found given a step of the implementation: in a *forward* simulation it must be shown that *from each* state of the specification which is related to the pre-state of the step of the implementation, there exists a sequence of steps (with the right actions) ending in *some* state of the specification which is related to the post-state of the step of the specification. Thus, one must, for all states related to the pre-state, trace *forward* to find some state related to the post-state. Conversely, in a *backward* simulation one must, for all states related to the post-state, trace *backward* to find some state related to the pre-state. A *refinement* mapping is a forward simulation where the relation is a function.

Since we only have to consider the steps of the implementation automaton which start in a reachable state, we will usually prove some *invariants*, i.e., properties that are true of all the reachable states, to restrict the states that we need to consider.

More formally, let A and B be automata and let R be a relation over the states of A and the states of B. Then R is a forward simulation from A to B iff

- For each start state of A, there is a related (by R) start state of B.
- For each step (s, a, s') of A, where s and s' satisfy the invariants of A, and each state u related to s that satisfies the invariants of B, there exists a sequence of steps

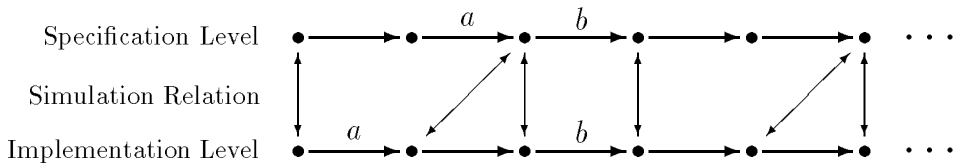

Figure 2. Example of a simulation. The actions a and b are visible actions. The rest of the transitions are thought of as labelled by internal actions.

of B, starting in u and ending in some state related to s', such that the sequence of steps contains the same visible actions as (s, a, s').

For timed systems the *now* components and time-passage actions require additional treatment. A backward simulation can be defined similarly.

Inductive proofs show the *soundness* of the simulation proof techniques, i.e., that they imply safe implementation. However, not all techniques are *complete*, meaning that for some of the simulation techniques, e.g., forward and backward simulations, safe implementation does not imply the existence of such a simulation. Also, different simulation techniques apply to different situations. Thus, although a forward simulation is the most intuitive technique, there are some situations that require other techniques, like backward simulations. For instance, a backward, but not a forward, simulation can be used in situations where the implementation makes some decisions later than the corresponding decisions are made in the specification. We will see an example of this in Section 6.

3. SPECIFICATION S

This section is devoted to giving the specification, called S, of the at-most-once message delivery problem more formally. Since this is an *untimed* specification, we give the specification in terms of an untimed automaton.

Figure 3 shows the user-interface of the protocols to be developed by depicting the specification as a "black box" with visible input and output actions. A user can send a message m by performing a *send_msg*(m) action and the protocol can deliver the next message m by performing a *receive_msg*(m) action. A Boolean acknowledgement b is passed to the user at the sender side by an $ack(b)$ action. At both the sender and receiver sides, a *crash* action causes the protocol to enter a *recovery* phase where messages might be lost (modelled by an internal *lose* action). A *recover* action, at the side where the crash occurred, then signals the end of the recovery phase, after which no messages can be lost unless new crashes occur.

The following code describes the specification S in a simple precondition-effect style commonly used for I/O-automata protocols [11, 12]. Note that input actions have no preconditions since our system should be able to respond to input from the environment at any time.

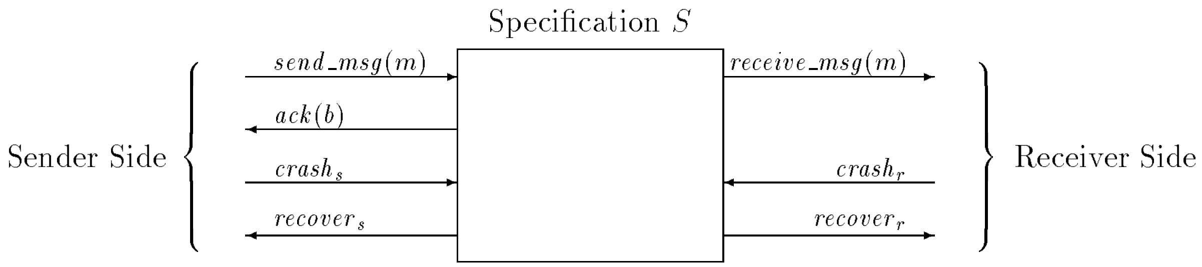

Figure 3. The specification S as a "black box"

The state of the automaton contains a *queue* of pending messages, plus two flags rec_s and rec_r, where the subscripts refer to the sender and receiver sides, to indicate that we are in a recovery phase, and a *status* component giving the status of the last message sent. The *status* can be either "?" denoting that the last message sent is still in *queue*, *true* denoting that the last message sent has been successfully delivered to the user, or *false*. A *status* value of *false*, and therefore a negative acknowledgement, only allows the user to conclude that there has been a crash, but even in this situation the last message sent may have been successfully delivered.[3]

$send\_msg(m)$
  Eff: $queue := queue \frown m$
    $status := ?$

$ack(b)$
  Pre: $status = b$ ($\in$ Bool)
  Eff: none

$crash_s$
  Eff: $rec_s := true$

$lose$
  Pre: $rec_s = true \vee rec_r = true$
  Eff: delete arbitrary elements of *queue*
    if the element at the end of *queue* was deleted then
      $status := false$
    else
      optionally $status := false$

$recover_s$
  Pre: $rec_s = true$
  Eff: $rec_s := false$

$receive\_msg(m)$
  Pre: $queue \neq \langle\rangle \wedge \underline{\mathbf{hd}}\, queue = m$
  Eff: $queue := \underline{\mathbf{tl}}\, queue$
    if $queue = \langle\rangle \wedge status = ?$ then
      $status := true$

$crash_r$
  Eff: $rec_r := true$

$recover_r$
  Pre: $rec_r = true$
  Eff: $rec_r := false$

[3]Throughout this paper we use the following operations on lists: let l be a list $\langle e_0, \ldots, e_n\rangle$ with elements e_0 through e_n. Then $l \frown m$ and $\underline{\mathbf{tl}}\, l$ denote the lists $\langle e_0, \ldots, e_n, m\rangle$ and $\langle e_1, \ldots, e_n\rangle$, respectively, and $\underline{\mathbf{hd}}\, l$ and $\underline{\mathbf{last}}\, l$ denote the elements e_0 and e_n, respectively.

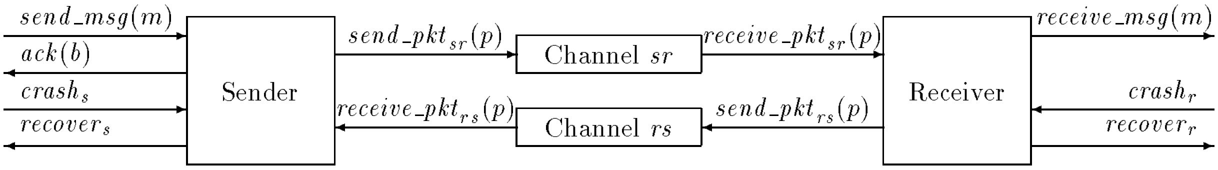

Figure 4. The Clock-Based Protocol C.

4. CLOCK-BASED PROTOCOL C

Figure 4 shows the structure of the clock-based protocol of [10], which we call C. An additional component, "the clock subsystem", is needed to keep the local clocks of the sender and the receiver "almost" synchronized.

Informally, the clock-based protocol works as follows. The sender associates a timestamp with each message it wishes to transmit. The timestamps are obtained from the sender's local clock $time_s$. The receiver uses the associated timestamp to decide whether or not to accept a received message—roughly, it will accept a message provided that the associated timestamp is greater than the timestamp of the last message that was accepted. However, the receiver does not always remember the timestamp of the last accepted message: it might forget this information because of a crash, or simply because a long time has elapsed since the last message was accepted and it is no longer efficient to remember it. Therefore, the receiver uses safe time estimates determined from its own local clock ($time_r$) to decide when to accept a message. The estimates are kept in the variables $lower_r$ and $upper_r$; the receiver accepts if the message's timestamp is in the interval $(lower_r, upper_r]$.

The $lower_r$ bound is designed to be at least as big as the time of the last message accepted. It can be bigger, however, as long as it is sufficiently less than the receiver's local time (at least approximately a one-way message delay less). This is because the receiver should not accidentally fail to accept a valid message that takes the maximum time to arrive. We note that the reason that we do not want to remember just the last timestamp is that we envision using this protocol in parallel for many users, and a single $lower_r$ bound could be used for all users that have not sent messages for a long while.

The $upper_r$ bound is chosen to be big enough so that the receiver still accepts the most recent messages, even if they arrive very fast. That is, it should be somewhat larger than the current time. But this bound will be kept in stable storage, and therefore should not be updated very often. Thus, it will generally be set to be a good deal larger than the current local time. In particular, it will be larger than the timestamp of any message so far accepted.

All that needs to be kept in stable storage is just the local clocks, plus the one variable $upper_r$ of the receiver. When the receiver side crashes and recovers again, it resets its $lower_r$ bound to the old $upper_r$ bound, to be sure that it will not accept, and thus deliver, any message twice. This explains why we cannot just set $upper_r$ to infinity.

There is also a simple acknowledgement protocol, which we do not discuss in detail here.

We now consider how to model this protocol formally. Since we are in a timed setting, the architecture of the protocol consists of the parallel composition of a timed automaton for each of the sender and receiver, plus timed automata to represent the two communication channels, plus an additional timed automaton to represent an almost-synchronized clock subsystem.

Each communication channel acts as follows. A $send\_pkt(p)$ action places the indicated packet[4] p in the channel. The channel is allowed to lose, duplicate, and reorder any packets, but we will assume that for every k $send\_pkt(p)$ actions for a particular packet, at least one packet p is not lost. For each such packet p, a $receive\_pkt(p)$ action will occur within time d, the maximum channel delay time. We note that it is possible to give a more abstract description of the channels; our results do not depend on this particular description.

The clock subsystem updates the local clocks of the sender and receiver by issuing *tick* actions in arbitrary ways so as to keep those clocks within ϵ of real time. We do not describe the channels and the clock subsystem formally in this paper.

For the sender and receiver, we first mention the state variables not described above and then define the transition relations.

For the sender, $mode_s$, ranging over `idle`, `send`, and `rec`, indicates whether the sender is idle, sending the current message to the channel, or in recovery phase, respectively. The variable $current\text{-}msg_s$ holds the current message, while $last_s$ holds its timestamp. The list of messages buf_s contains the remaining messages waiting to be sent. Finally, $current\text{-}ack_s$ of type Bool holds the acknowledgement received from the receiver.

For the receiver, $mode_r$, ranges over `idle`, `rcvd`, `ack`, and `rec`. `rcvd` indicates that packets have been received on the channel but the associated messages not yet passed to the user, `ack` indicates that all messages have been passed to the user and that the receiver is issuing positive acknowledgements, and finally `rec` indicates the recovery phase. The list of messages buf_r holds the messages received from the channel but not yet passed to the user. For the simple acknowledgement protocol, we have $last_r$ to hold the timestamp of the last accepted message, and $nack\text{-}buf_r$ to hold the timestamps for which negative acknowledgements must be issued. Finally, the variable $rm\text{-}time_r$ is used by the $cleanup_r$ action.

The definition of the steps is listed in the left column below for the sender and in the right for the receiver. We have aligned the *send_pkt* and corresponding *receive_pkt* actions to increase readability.

We treat timing requirements implicitly by giving upper time bounds on certain classes of actions. This corresponds to the way timing requirements are specified in [15]. The code contains three unspecified timing constants α, β, and ρ, to be explained after the code.

[4] Here and elsewhere, we use the term "packet" to denote objects sent over the channels in an implementation; we reserve the term "message" for the "higher-level", user-meaningful messages that appear, e.g., in the specification.

$send\_msg(m)$
Eff: if $mode_s \neq \texttt{rec}$ then $buf_s := buf_s \,\hat{}\, m$

$choose\_id(t)$
Pre: $mode_s = \texttt{idle} \wedge buf_s \neq \langle\rangle \wedge time_s = t \wedge t > last_s$
Eff: $mode_s := \texttt{send}$
$last_s := t$
$current\text{-}msg_s := \underline{\mathbf{hd}}\, buf_s$
$buf_s := \underline{\mathbf{tl}}\, buf_s$

$send\_pkt_{sr}(m,t)$
Pre: $mode_s = \texttt{send} \wedge$
$current\text{-}msg_s = m \wedge last_s = t$
Eff: none

$receive\_pkt_{sr}(m,t)$
Eff: if $mode_r \neq \texttt{rec}$ then
if $lower_r < t \leq upper_r$ then
$mode_r := \texttt{rcvd}$
$buf_r := buf_r \,\hat{}\, m$
$last_r, lower_r := t$
else if $last_r < t \leq lower_r$ then
$nack\text{-}buf_r := nack\text{-}buf_r \,\hat{}\, t$
else if $mode_r = \texttt{idle} \wedge last_r = t$ then
$mode_r := \texttt{ack}$

$receive\_msg(m)$
Pre: $mode_r = \texttt{rcvd} \wedge buf_r \neq \langle\rangle \wedge \underline{\mathbf{hd}}\, buf_r = m$
Eff: $buf_r := \underline{\mathbf{tl}}\, buf_r$
if $buf_r = \langle\rangle$ then
$mode_r := \texttt{ack}$
$rm\text{-}time_r := time_r$

$receive\_pkt_{rs}(t,b)$
Eff: if $mode_s = \texttt{send} \wedge last_s = t$ then
$mode_s := \texttt{idle}$
$current\text{-}ack_s := b$
$current\text{-}msg_s := \texttt{nil}$

$ack(b)$
Pre: $mode_s = \texttt{idle} \wedge$
$buf_s = \langle\rangle \wedge current\text{-}ack_s = b$
Eff: none

$send\_pkt_{rs}(t, true)$
Pre: $mode_r = \texttt{ack} \wedge last_r = t$
Eff: $mode_r := \texttt{idle}$

$send\_pkt_{rs}(t, false)$
Pre: $mode_r \neq \texttt{rec} \wedge$
$nack\text{-}buf_r \neq \langle\rangle \wedge \underline{\mathbf{hd}}\, nack\text{-}buf_r = t$
Eff: $nack\text{-}buf_r := \underline{\mathbf{tl}}\, nack\text{-}buf_r$

$crash_s$
Eff: $mode_s := \texttt{rec}$

$crash_r$
Eff: $mode_r := \texttt{rec}$

$recover_s$
Pre: $mode_s = \texttt{rec}$
Eff: $mode_s := \texttt{idle}$
$last_s := time_s$
$rm\text{-}time_r := \infty$
$buf_s := \langle\rangle$
$current\text{-}msg_s := \texttt{nil}$
$current\text{-}ack_s := false$

$recover_r$
Pre: $mode_r = \texttt{rec} \wedge upper_r + 2\epsilon < time_r$
Eff: $mode_r := \texttt{idle}$
$last_r := 0$
$rm\text{-}time_r := \infty$
$buf_r, nack\text{-}buf_r := \langle\rangle$
$lower_r := upper_r$
$upper_r := time_r + \beta$

increase-lower$_r(t)$
  Pre: $mode_r \neq \texttt{rec} \wedge lower_r \leq t < time_r - \rho$
  Eff: $lower_r := t$

increase-upper$_r(t)$
  Pre: $mode_r \neq \texttt{rec} \wedge upper_r \leq t = time_r + \beta$
  Eff: $upper_r := t$

cleanup$_r$
  Pre: $mode_r \in \{\texttt{idle}, \texttt{ack}\} \wedge$
  $time_r > rm\text{-}time_r + \alpha$
  Eff: $mode_r := \texttt{idle}$
  $last_r := 0$
  $rm\text{-}time_r := \infty$

$tick_s(t)$
  Eff: $time_s := t$

$tick_r(t)$
  Eff: $time_r := t$

The correctness of the clock-based protocol requires that we put upper time bounds on one set of actions of the sender. Informally,

- every time a $send\_pkt_{sr}(m,t)$ action becomes possible (or stays possible after being executed), it must occur within time l_s unless it gets disabled in the meantime.

Similarly, for the receiver we need to put upper bound on two classes of actions:

- $send\_pkt_{rs}(id, true)$ has an upper bound of l_r, and
- *increase-upper*$_r(t)$ has an upper bound of l'_r.

The correctness of the protocol depends on the timing constants in the code being related properly to these time bounds, and to channel and local clock characteristics. The requirements are: $\beta \geq 2\epsilon + l'_r$, $\rho \geq kl_s + d + 2\epsilon$, and $\alpha \geq k(l_r + d) + (k-1)kl_s + 2\epsilon$. Note, how all three constants depend on the maximum difference of 2ϵ between the local clocks.

In this paper we do not give a formal specification of H. Unlike C, which uses timing assumptions, H uses handshakes to first make the sender and receiver agree on a message identifier and then perform the actual message transmission. An additional packet type is used as cleanup information. We proceed by describing the generic protocol G.

5. GENERIC PROTOCOL G

The two protocols C and H both go through three major phases during normal operation:

Choosing a message identifier The sender picks an identifier *id* that is within the set of identifiers that the receiver is willing to accept. In C time bounds are used to choose a good identifier; in H an initial handshake between the sender and the receiver is used.

Sending the message and getting acknowledgement This phase is similar in both C and H. The sender (re)transmits the current message with the chosen id, until it receives an acknowledgement packet for that id.

Cleaning up Here again, C uses time bounds (in particular timeouts) whereas H uses a handshake to determine when some "old" information may be discarded.

Our generic protocol G is designed to capture these three phases in an abstract way that both C and H implement. The key abstractions incorporated into the protocol G are two variables, $good_s$ and $good_r$. The variable $good_s$ represents the identifiers that the sender might shortly assign to messages, and $good_r$ represents the identifiers that the receiver is willing to accept.

Four actions of G deal with "growing" and "shrinking" $good_s$ and $good_r$, respectively. As an example, remember that in C the identifier for the current message is taken from the sender's local clock $time_s$. Every time the local clock $time_s$ is advanced, the identifier that the sender might assign to the current message is changed from the old value of $time_s$ to the new value. In G this corresponds to first "shrinking" $good_s$ with the old value and then "growing" it with the new value.

The preconditions of the grow and shrink actions are designed to preserve certain key invariants, one of which we will present in Section 6. We actually allow more freedom in these actions than is actually needed by C and H. This leaves open the possibility that other low-level protocols, other than C and H, can be proved to be implementations of G. We show the parts of G that deal with $good_s$ and $good_r$.

$send\_msg(m)$ ...

$prepare$
Pre: $mode_s = \texttt{idle} \wedge buf_s \neq \langle\rangle$
Eff: $mode_s := \texttt{needid}$
$good_s := \{\,\}$
$current\text{-}msg_s := \underline{\mathbf{hd}}\ buf_s$
$buf_s := \underline{\mathbf{tl}}\ buf_s$
if $mode_r \neq \texttt{rec}$ then $current\text{-}ok := true$

$choose\_id(id)$
Pre: $mode_s = \texttt{needid} \wedge id \in good_s$
Eff: $mode_s := \texttt{send}$
$last_s := id$
$used_s := used_s \,\hat{}\, id$

$send\_pkt_{sr}(m, id)$
Pre: $mode_s = \texttt{send} \wedge$
$last_s = id \wedge current\text{-}msg_s = m$
Eff: none

$receive\_pkt_{sr}(m, id)$
Eff: if $mode_r \neq \texttt{rec}$ then
if $id \in good_r$ then
$mode_r := \texttt{rcvd}$
$buf_r := buf_r \,\hat{}\, m$
$last_r := id$
$good_r := good_r \setminus \{id' \mid id' \leq id\}$
if $id = last_s \wedge mode_s = \texttt{send}$ then
$current\text{-}ok := false$
else if $id \neq last_r$ then
optionally $nack\text{-}buf_r := nack\text{-}buf_r \,\hat{}\, id$
else if $mode_r = \texttt{idle}$ then
$mode_r := \texttt{ack}$

$receive\_msg(m)$ ...

$receive\_pkt_{rs}(id, b)$ ...

$send\_pkt_{rs}(id, true)$ ...

$ack(b)$ ...

$send\_pkt_{rs}(id, false)$ ...

$crash_s$ ...

$crash_r$ ...

$recover_s$ ...

$recover_r$ ...

$shrink\_good_s(ids)$
Pre: none
Eff: $good_s := good_s \setminus ids$

$shrink\_good_r(ids)$
Pre: $current\text{-}ok = false \vee$
$((mode_s = \mathtt{needid} \Longrightarrow ids \cap good_s = \{\,\}) \wedge$
$(mode_s = \mathtt{send} \Longrightarrow last_s \notin ids))$
Eff: $good_r := good_r \setminus ids$

$grow\_good_s(ids)$
Pre: $mode_s \neq \mathtt{needid} \vee$
$((mode_r \neq \mathtt{rec} \Longrightarrow ids \subseteq issued_r) \wedge$
$(current\text{-}ok = true \Longrightarrow ids \subseteq good_r) \wedge$
$(ids \cap used_s = \{\,\}))$
Eff: $good_s := good_s \cup ids$

$grow\_good_r(ids)$
Pre: $ids \cap issued_r = \{\,\}$
Eff: $good_r := good_r \cup ids$
$issued_r := issued_r \cup ids$

$cleanup_r$
Pre: $mode_r \in \{\mathtt{idle}, \mathtt{ack}\} \wedge$
$(mode_s = \mathtt{send} \Longrightarrow last_s \neq last_r)$
Eff: $mode_r := \mathtt{idle}$
$last_r := \mathtt{nil}$

6. CORRECTNESS PROOFS

In this section we sketch the proofs of safe implementation for the different levels of abstraction in our work. We will not be strictly formal. For such formal treatment and full proofs we refer to our full report [8].

6.1. Correctness of G

To prove that G is a safe implementation of S, we need a *backward simulation*. Informally this is because G (and the lower-level protocols) may *postpone* the decisions about which messages to lose because of a crash till after recovery, whereas in S message loss occurs between crash and recovery. It is due to certain *race conditions* on the channels that the decisions are delayed in G, C, and H.

Since a backward simulation directly from G to S is still more complicated than we would like, we split the task one more time. Our strategy is to try to localize the backward simulation reasoning, because reasoning in this way seems to be inherently difficult compared to the more intuitive forward simulation and refinement mapping techniques. Thus, we define a new, "delayed-decision" version D of the specification (see Figure 1). D is just like S except that it delays the point at which the decision about loss of messages is made. Now, when a crash occurs, messages in the system and *status* may get *marked*.

Later, even after recovery, any marked message is allowed to be lost. Also, a marked *status* is allowed to be lost, i.e., changed to *false*.

Due to space constraints we will not define D formally, nor will we define or prove the backward simulation from D to S. We only note that such a backward simulation exists, which allows us to conclude that D safely implements S.

The proof that G safely implements D uses certain invariants. For instance, the following key invariant of G states that when no crashes have occurred since the sender executed a *prepare* action (i.e., *current-ok* $=$ *true*) and the sender still has not chosen an identifier, then the sender can only choose identifiers that are considered good by the receiver. The *shrink_good* and *grow_good* actions are explicitly designed to preserve this invariant (among others).

If $\textit{current-ok} = \textit{true} \wedge \textit{mode}_s = \texttt{needid}$ then $\textit{good}_s \subseteq \textit{good}_r$.

The proof that G safely implements D is now discharged by exhibiting a refinement mapping from G to D. Again, we do not give details, but refer to [8] for the complete proofs.

Together, the existence of the backward simulation from D to S and the refinement mapping from G to D allow us to conclude that G safely implements S.

6.2. Correctness of C and H

To prove that C and H safely implement S we just have to prove some simulations from C and H to G, since we have already shown that G safely implements S. We only consider C.

The first step, a technical one, in the correctness proof involves adding a history variable [16] *deadline* to C to get an equivalent version of the protocol (which we still call C). The variable *deadline* is set to the current real time plus a maximum one-way delay when the current message gets a timestamp, and gets reset (to ∞) when either a crash occurs or the current message gets accepted by the receiver. Below we show an invariant that states that this deadline is always met in any execution of C. Another history variable $\textit{used}_s$ is a list containing the timestamps used so far.

Next, note that G is formalized as an untimed automaton whereas C is formalized as a timed automaton. We resolve this model inconsistency by converting G into the timed model to get G_t. That is, an untimed automaton can be considered to be a timed automaton, where time can pass arbitrarily. (Additional liveness restrictions on G will make sure that time-passage is not the only activity of G_t). Below we prove that C safely implements G_t in the timed setting. Certain *embedding* results then allow us to conclude that since G safely implements S, G_t safely implements the converted specification S_t, which furthermore implies that C safely implements S_t. Thus, the embedding results allow us to work mostly within the simpler untimed model.

Definition 1 (Refinement from C to G_t) If s is a state of C then $R_{CG}(s)$ is the state u of G_t such that

$$
\begin{array}{lcl}
u.good_s & = & \{s.time_s\} - \{s.last_s\} \\
u.good_r & = & (s.lower_r, s.upper_r] \\
u.issued_r & = & (0, s.upper_r] \\
u.current\text{-}ok & = & (s.deadline \neq \infty)
\end{array}
$$

All remaining variables (including the channels) have the same contents in C and G_t. ∎

The timestamp that the sender in C might associate with a message, corresponding to $good_s$ in G, is taken from the local clock $time_s$, but only if the local clock has advanced since the last timestamp ($last_s$). This explains the line for $good_s$. The lines for $good_r$ and $issued_r$ can be explained similarly.

A key invariant we need in the refinement proof is that the history variable *deadline* of C is always at least as big as real time, i.e., $now \leq deadline$. This can then be used to prove the following invariant: for all timestamps t on the channel from sender to receiver, $deadline \neq \infty \implies upper_r \geq t$. This invariant states that $upper_r$ is always sufficiently large when the current message is being transmitted on the channel and no crashes have occurred.

Lemma 2 *R_{CG} is a refinement mapping from C to G_t.*

Proof The proof of a refinement mapping in the timed setting has three points, which we sketch here:

- For any state s of C, $R_{CG}(s)$ has the same real time (now) as s. This is satisfied immediately by the definition of R_{CG}.
- For any start state s of C, $R_{CG}(s)$ is a start state of G_t. This is easy to check.
- For each step (s, a, s') of C, where s and s' satisfy the invariants of C, we must show the existence of a sequence of steps $(R_{CG}(s), a_1, \ldots, a_n, R_{CG}(s'))$ of G_t with the same trace (see Figure 2). We conduct such a proof by doing a *case analysis* based on the different actions a of C. For instance, if $a = increase\text{-}lower_r(t)$, we show that $(R_{CG}(s), shrink\_good_r((0,t]), R_{CG}(s'))$ is a step of G_t. ∎

The correctness of H can also be proved using the refinement mapping technique.

7. CONCLUSION

In this paper, we have used a simple automaton model to present the at-most-once message delivery problem and two interesting solutions, and have used simulation techniques to prove that the two algorithms meet the specification. We have only given arguments for safety and timing properties here, and have left liveness for a longer report. We believe that this work yields important insights into the protocols, and also serves to show the adequacy of the model and proof techniques. Similar protocols have been verified formally, using different techniques. LOTOS has, for instance, been used in [14].

There is a considerable amount of further work remaining. First, if the timing assumptions on C are weakened or removed, the resulting algorithm still will not deliver any message more than once; however, it may lose messages even in the absence of a crash.

It remains to formulate the weaker specification and prove that the weaker version of C satisfies it.

Second, there are other algorithms that solve the at-most-once message delivery problem, for example, using bounded identifier spaces or cryptographic assumptions. We would like also to verify these, again reusing as much of our proofs as possible. In [17] bounded identifier spaces are dealt with for similar protocols.

Third, we would like to automate our simulation proofs using a mechanical theorem prover. We have already begun this work, by proving the equivalence of versions of S and D using the Larch Prover [18, 4]. We have been pleased with the preliminary results: the prover has not only been able to check our hand proofs, but in fact has been able to fill in many of the details. We can draw several conclusions:

- Automata, invariants, and simulations are all excellent tools for verifying timed and untimed communication protocols. The methods scale well, yield insight, and are not too difficult to use.

- A general model such as timed automata is needed in order to model and verify most communication protocols in a single coherent framework. However, for reasoning about particular protocols, it is often better to work in a simplified special case of the general model. (For instance, for untimed protocols such as H, it is best to avoid details of timing.) What is needed is a collection of special models, each of which can be easily "embedded" in the general model.

- Safety proofs are challenging. They require insight to obtain the right invariants and simulations, and a lot of detailed work to verify these choices. Computer assistance can help with the details; however, the insight will always be required.

- Backward simulations are much harder to do than refinements and forward simulations but are necessary in certain situations.

- Many algorithms can be treated as implementations of a common abstract algorithm.

- Verifying a coordinated collection of protocols, rather than just a single isolated protocol, is extremely valuable. It leads to the discovery of useful abstractions, and tends to make the proofs more elegant and reusable.

- Doing proofs for realistic communication protocols is feasible now. We predict that it will become more so, and will be of considerable practical importance.

REFERENCES

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.
2. M. Abadi and L. Lamport. An old-fashioned recipe for real time. In J. W. de Dakker, C. Huizing, and G. Rozenberg, editors, *Proceedings of REX Workshop "Real-Time: Theory in Practice", Mook, The Netherlands, June 1991*, number 600 in Lecture Notes in Computer Science, pages 1–27. Springer-Verlag, 1992.

3. D. Belsnes. Single message communication. *IEEE Transactions on Communications*, Com-24(2), February 1976.
4. Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, DEC, Systems Research Center, December 1991.
5. R. Gawlick, N. Lynch, R. Segala, and J. Søgaard-Andersen. Liveness in timed and untimed systems. Technical report, MIT, Laboratory for Computer Science, 1993.
6. B. Jonsson. Simulations between specifications of distributed systems. In J. C. M. Baeten and J. F. Groote, editors, *Proceedings of CONCUR '91. 2nd International Conference on Concurrency Theory, Amsterdam, The Netherlands, August 1991*, number 527 in Lecture Notes in Computer Science, pages 346–360. Springer-Verlag, 1991.
7. S. S. Lam and A. U. Shankar. Protocol verification via projections. *IEEE Transactions on Software Engeneering*, 10(4), 1984.
8. B. Lampson, N. Lynch, and J. Søgaard-Andersen. Reliable at-most-once message delivery protocols. Technical report, MIT, Laboratory for Computer Science, 1993. Full report.
9. B. Liskov. Practical uses of synchronized clocks in distributed systems. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 1–10, 1991.
10. B. Liskov, L. Shrira, and J. Wroclawski. Efficient at-most-once messages based on synchronized clocks. *ACM Transactions on Computer Systems*, 9(2):125–142, May 1991.
11. N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, MIT, Laboratory for Computer Science, Cambridge, MA, 02139, April 1987.
12. N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.
13. N. Lynch and F. Vaandrager. Forward and backward simulations for timing-based systems. In J. W. de Dakker, C. Huizing, and G. Rozenberg, editors, *Proceedings of REX Workshop "Real-Time: Theory in Practice", Mook, The Netherlands, June 1991*, number 600 in Lecture Notes in Computer Science, pages 397–446. Springer-Verlag, 1992.
14. Eric Madelaine and Didier Vergamini. Specification and verification of a sliding window protocol in LOTOS. In K. R. Parker and G. A. Rose, editors, *Formal Description Techniques, IV*, pages 495–510. North-Holland, 1991.
15. M. Merritt, F. Modugno, and M. Tuttle. Time-constrained automata. In *Proceedings of CONCUR'91. 2nd International Conference on Concurrency Theory, Amsterdam, The Netherlands, August 1991*, number 527 in Lecture Notes in Computer Science, pages 408–423. Springer-Verlag, 1991.
16. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.
17. A. U. Shankar. Verified data transfer protocols with variable flow control. *ACM Transactions on Computer Systems*, 7(3):281–316, August 1989.
18. Jørgen F. Søgaard-Andersen, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. Computer-assisted simulation proofs. In Costas Courcoubetis, editor, *Computer Aided Verification. 5th International Conference, CAV '93. Elounda, Greece, June/July 1993*, volume 697 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 1993.