Chapter 10

Reliable Messages and Connection Establishment

Butler W. Lampson

10.1 Introduction

Given an unreliable network, we would like to reliably deliver messages from a sender to a receiver. This is the function of the transport layer of the ISO seven-layer cake. It uses the network layer, which provides unreliable message delivery, as a channel for communication between the sender and the receiver.

Ideally we would like to ensure that

- messages are delivered in the order they are sent,
- every message sent is delivered exactly once, and
- an acknowledgement is returned for each delivered message.

Unfortunately, it's expensive to achieve the second and third goals in spite of crashes and an unreliable network. In particular, it's not possible to achieve them without making some change to stable state (state that survives a crash) every time a message is received. Why? When we receive a message after a crash, we have to be able to tell whether it has already been delivered. But if delivering the message doesn't change any state that survives the crash, then we can't tell.

So if we want a cheap deliver operation which doesn't require writing stable state, we have to choose between delivering some messages more than once and losing some messages entirely when the receiver crashes. If the effect of a message is idempotent, of course, then duplications are harmless and we will choose

---

The material in this chapter is the result of joint work with Nancy Lynch and Jørgen Søgaard-Andersen.

the first alternative. But this is rare, and the latter choice is usually the lesser of two evils. It is called 'at-most-once' message delivery. Usually the sender also wants an acknowledgement that the message has been delivered, or in case the receiver crashes, an indication that it might have been lost. At-most-once messages with acknowledgements are called 'reliable' messages.

There are various ways to implement reliable messages. An implementation is called a 'protocol', and we will look at several of them. 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 keeps a stock of ers 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 good set. Otherwise, the receiver just discards the message, perhaps after acknowledging it. In order for the sender to be sure that its message will be delivered rather than discarded, it must tag the message with a good identifer.

What makes the implementations tricky is that we expect to lose some state when there is a crash. In particular, the receiver will be keeping track of at least some of its good identifiers in volatile variables, so these identifiers will become bad at the crash. But the sender doesn't know about the crash, so it will go on using the bad identifiers and thus send messages that the receiver will reject. Different protocols use different methods to keep the sender and the receiver more or less in sync about what identifiers to use.

In practice reliable messages are most often implemented in the form of 'connections'. The idea is that a connection is 'established', any amount of information is sent on the connection, and then the connection is 'closed'. You can think of this as the sending of a single large message, or as sending the first message using one of the protocols we discuss, and then sending later messages with increasing sequence numbers. Usually connections are full-duplex, so that either end can send independently, and it is often cheaper to establish both directions at the same time. We ignore all these complications in order to concentrate on the essential logic of the protocols.

What we mean by a crash is not simply a failure and restart of a node. In practice, protocols for reliable messages have limits, called 'timeouts', on the length of time for which they will wait to deliver a message or get an ack. We model the expiration of a timeout as a crash: the protocol abandons its normal operation and reports failure, even though in general it's possible that the message in fact has been or will be delivered.

We begin by writing a careful specification S for reliable messages. Then we present a 'lower-level' spec D in which the non-determinism associated with losing messages when there is a crash is moved to a place that is more convenient for implementations. We explain why D implements S but don't give a proof, since that requires techniques beyond the scope of this chapter. With this groundwork, we present a generic protocol G and a proof that it implements D. Then we describe two protocols that are used in practice, the handshake protocol H and the clock-based protocol C, and show how both implement G. Finally, we explain how to modify our protocols to work with finite sets of message identifiers, and summarize our results.

The goals of this chapter are to:

- Give a simple, clear, and precise specification of reliable message delivery in the presence of crashes.
- Explain the standard handshake protocol for reliable messages that is used in TCP, ISO TP4, and many other widespread communication systems, as well as a newer clock-based protocol.
- Show that both protocols can be best understood as special cases of a simpler, more general protocol for using identifiers to tag messages and acknowledgements for reliable delivery.
- Use the method of abstraction functions and invariants to help in understanding these three subtle concurrent and fault-tolerant algorithms, and in the process present all the hard parts of correctness proofs for all of them.
- Take advantage of the generic protocol to simplify the analysis and the arguments.

10.1.1 Methods

We use the definition of 'implements' and the abstraction function proof method explained in Chapter 3. Here is a brief summary of this material.

Suppose that X and Y are state machines with named transitions called *actions*; think of X as a specification and Y as an implementation. We partition the actions of X and Y into *external* and *internal* actions. A *behavior* of a machine M is a sequence of actions that M can take starting in an initial state, and an *external behavior* of M is the subsequence of a behavior that contains only the external actions. We say Y *implements* X iff every external behavior of Y is an external behavior of X.[1] This expresses the idea that what it means for Y to implement X is that from the outside you don't see Y doing anything that X couldn't do.

The set of all external behaviors is a rather complicated object and difficult to reason about. Fortunately, there is a general method for proving that Y implements X without reasoning explicitly about behaviors in each case. It works as follows. First, define an *abstraction function* f from the state of Y to the state of X. Then show that Y *simulates* X:

1. f maps an initial state of Y to an initial state of X.
2. For each Y-action and each reachable state y there is a sequence of X-actions (perhaps empty) that is the same externally, such that the following diagram commutes.

---

[1] Actually this definition only deals with the implementation of *safety* properties. Roughly speaking, a safety property is an assertion that nothing bad happens; it is a generalization of the notion of partial correctness for sequential programs. A system that does nothing implements any safety property. Specifications may also include *liveness* properties, which roughly assert that something good eventually happens; these generalize the notion of termination for sequential programs. A full treatment of liveness is beyond the scope of this chapter, but we do explain informally why the protocols make progress.

$$\begin{tikzcd}
f(y) \arrow[r, "\textbf{X-actions}"] & f(y') \\
y \arrow[u, "f"] \arrow[r, "\textbf{action}"] & y' \arrow[u, "f"]
\end{tikzcd}$$

A sequence of X-actions is the same externally as a Y-action if they are the same after all internal actions are discarded. So if the Y-action is internal, all the X-actions must be internal (perhaps none at all). If the Y-action is external, all the X-actions must be internal except one, which must be the same as the Y-action.

A straightforward induction shows that Y implements X: For any Y-behavior we can construct an X-behavior that is the same externally, by using (2) to map each Y-action into a sequence of X-actions that is the same externally. Then the sequence of X-actions will be the same externally as the original sequence of Y-actions.

In order to prove that Y simulates X we usually need to know what the reachable states of Y are, because it won't be true that every action of Y from an arbitrary state of Y simulates a sequence of X-actions; in fact, the abstraction function might not even be defined on an arbitrary state of Y. The most convenient way to characterize the reachable states of Y is by an *invariant*, a predicate that is true of every reachable state. Often it's helpful to write the invariant as a conjunction, and to call each conjunct an invariant. It's common to need a stronger invariant than the simulation requires; the extra strength is a stronger induction hypothesis that makes it possible to establish what the simulation does require.

So the structure of a proof goes like this:

- Establish invariants to characterize the reachable states, by showing that each action maintains the invariants.
- Define an abstraction function.
- Establish the simulation, by showing that each Y-action simulates a sequence of X-actions that is the same externally.

This method works only with actions and does not require any reasoning about behaviors. Furthermore, it deals with each action independently. Only the invariants connect the actions. So if we change (or add) an action of Y, we only need to verify that the new action maintains the invariants and simulates a sequence of X-actions that is the same externally. We exploit this remarkable fact in Section 10.9 to extend our protocols so that they use finite, rather than infinite, sets of identifiers.

In what follows we give abstraction functions and invariants for each protocol. The actual proofs that the invariants hold and that each Y-action simulates a suitable sequence of X-actions are routine, so we give proofs only for a few sample actions.

10.1.2 Types and Notation

We use a type M for the messages being delivered. We assume nothing about M.

All the protocols except S and D use a type I of identifiers for messages. In general we assume only that Is can be compared for equality; C assumes a total ordering. If x is a multiset whose elements have a first I component, we write ids(x) for the multiset of Is that appear first in the elements of x.

We write ···⟩ for a sequence with the indicated elements and + for concatenation of sequences. We view a sequence as a multiset in the obvious way. We write $x = (y, *)$ to mean that x is a pair whose first component is y and whose second component can be anything, and similarly for $x = (*, y)$.

We define an action by giving its name, a *guard* that must be true for the action to occur, and an *effect* described by a set of assignments to state variables. We encode parameters by defining a whole family of actions with related names; for instance, *get*(m) is a different action for each possible m. Actions are atomic; each action completes before the next one is started.

To express concurrency we introduce more actions. Some of these actions may be internal, that is, they may not involve any interaction with the client of the protocol. Internal actions usually make the state machine non-deterministic, since they can happen whenever their guards are satisfied, not just when there is an interaction with the environment. We mark external actions with *s, two for an input action and one for an output action. Actions without *s are internal.

It's convenient to present the sender actions on the left and the receiver actions on the right. Some actions are not so easy to categorize, and we usually put them on the left.

10.2 The Specification S

The specification S for reliable messages is a slight extension of the spec for a FIFO queue. Figure 10.1 shows the external actions and some examples of its transitions. The basic state of S is the FIFO queue q of messages, with *put*(m) and *get*(m) actions. In addition, the *status* variable records whether the most recently sent message has been delivered. The sender can use *getAck*(a) to get this information; after that it may be forgotten by setting *status* to *lost*, so that the sender doesn't have to remember it forever. Both sender and receiver can crash and recover. In the absence of crashes, every message put is delivered by *get* in the same order and is positively acknowledged. If there is a crash, any message still in the queue may be lost at any time between the crash and the recovery, and its ack may be lost as well.

The *getAck*(a) action reports on the message most recently put, as follows. If there has been no crash since it was put there are two possibilities:

- the message is still in q and *getAck* cannot occur;
- the message was delivered by *get*(m) and *getAck*(*OK*) occurs.

If there have been crashes, there are two additional possibilities:

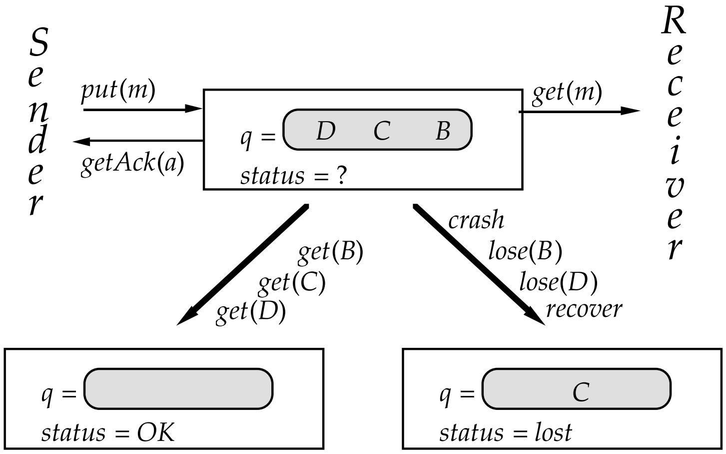

Figure 10.1. Some states and transitions for S

- the message was lost and *getAck*(*lost*) occurs;
- the message was delivered or is still in *q* but *getAck*(*lost*) occurs anyway.

The ack makes the most sense when the sender alternates *put*(*m*) and *getAck*(*a*) actions. Note that what is being acknowledged is delivery of the message to the client, not its receipt by some part of the implementation, so this is an end-to-end ack. In other words, the *get* should be thought of as including client processing of the message, and the ack might include some result returned by the client such as the result of a remote procedure call. This could be expressed precisely by adding an *ack* action for the client. We won't do that because it would clutter up the presentation without improving our understanding of how reliable messages work.

To define S we introduce the types *A* (for acknowledgement) with values in {*OK*, *lost*} and *Status* with values in {*OK*, *lost*, ?}. Table 10.1 gives the state and actions of S. Note that it says nothing about channels; they are part of the implementation and have nothing to do with the spec.

Why do we have both *crash* and *recover* actions, as opposed to just a *crash* action? A spec which only allows messages to be lost at the time of a *crash* is not implemented by a protocol like C in which the sender accepts a message with *put* and sends it without verifying that the receiver is running normally. In this case the message is lost even though it wasn't in the system at the time of the crash. This is why we have a separate $recover_r$ action which allows the receiver to declare the point after a crash when messages are again guaranteed not to be lost. There seems to be no need for a $recover_s$ action, but we have one for symmetry.

A spec which only allows messages to be lost at the time of a *recover* is not implemented by any protocol that can have two messages in the network at the same time, because after a $crash_s$ and before the following $recover_s$ it's possible for the second message in the network to be delivered, which means that the first one must be lost to preserve the FIFO property.

Sender Name	Guard	Effect	Receiver Name	Guard	Effect
**$put(m)$	$rec_s = false$	append m to q, $status := ?$	*$get(m)$	$rec_r = false$, m is first on q	remove head of q, if $q = empty$ and $status = ?$ then $status := OK$
*$getAck(a)$	$rec_s = false$, $status = a$	optionally $status := lost$			
**$crash_s$		$rec_s := true$	**$crash_r$		$rec_r := true$
*$recover_s$	rec_s	$rec_s := false$	*$recover_r$	rec_r	$rec_r := false$
$lose$	rec_s or rec_r	delete some element from q; if it's the last then $status := lost$, or $status := lost$			

q : sequence$[M]$ $:= \langle\, \rangle$
$status$: $Status$ $:= lost$
rec_s : Boolean $:= false$ (rec is short for 'recovering')
rec_r : Boolean $:= false$

Table 10.1. State and actions of S

The simplest spec which covers both these cases can lose a message at any time between a *crash* and its following *recover*, and we have adopted this alternative.

10.3 The Delayed-Decision Specification D

Next we introduce an implementation of S, called the delayed-decision specification D, that is more non-deterministic about when messages are lost. The reason for D is to simplify the proofs of the protocols: with more freedom in D, it's easier to prove that a protocol simulates D than to prove that it simulates S. A typical protocol transmits messages from the sender to the receiver over some kind of channel which can lose messages; to compensate for these losses, the sender retransmits. If the sender crashes with a message in the channel it stops retransmitting, but whether the receiver gets the message depends on whether the channel loses it. This may not be decided until after the sender has recovered. So the protocol doesn't decide whether the message is lost until after the sender has recovered. D has this freedom, but S does not.

D is the same as S except that the decisions about which messages to lose at recovery, and whether to lose the ack, are made by asynchronous *drop* actions that can occur after recovery. Each message in q, as well as the *status* variable, is augmented by an extra component of type *Mark* which is normally + but may become # between crash and recovery because of a *mark* action. At any time an *unmark* action can change a mark from # back to +, a message marked # can be lost by *drop*, or a *status* marked # can be set to *lost* by *drop*. Figure 10.2 gives an example of the transitions of D; the + marks are omitted.

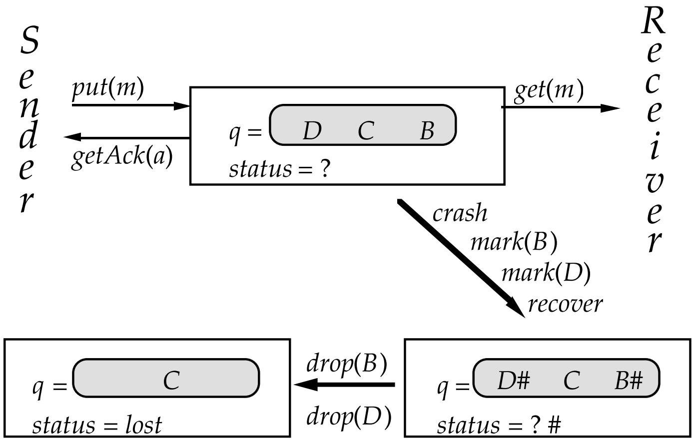

Figure 10.2. Some states and transitions of D

	Sender			Receiver	
Name	**Guard**	**Effect**	**Name**	**Guard**	**Effect**
**put(m)	$rec_s = false$	append $(m, +)$ to q, $status := (?, +)$	*get(m)	$rec_r = false$, $(m, *)$ first on q	remove head of q, if $q = empty$ and $status = (?, x)$ then $status := (OK, x)$
*getAck(a)	$rec_s = false$, $status = (a, *)$	$status := (a, +)$ or $status := (lost, +)$			
**$crash_s$		$rec_s := true$	**$crash_r$		$rec_r := true$
*$recover_s$	rec_s	$rec_s := false$	*$recover_r$	rec_r	$rec_r := false$
mark	rec_s or rec_r	for some element of q or for *status*, $mark := \#$	*unmark*		for some element of q or for *status*, $mark := +$
drop		delete an element of q with $mark = \#$; if it was the last element, $status := (lost, +)$ or if $status = (*, \#)$, $status := (lost, +)$			

q : sequence[$(M, Mark)$] $:= \langle\,\rangle$
$status$: $(Status, Mark)$ $:= (lost, +)$
rec_s : Boolean $:= false$
rec_r : Boolean $:= false$

Table 10.2. State and actions of D

Name	**Guard**	**Effect**	**Name**	**Guard**	**Effect**
**$send_{sr}(p)$		add some number of copies of p to sr	**$send_{rs}(p)$		add some number of copies of p to rs
*$rcv_{sr}(p)$	$p \in sr$	remove one p from sr	$rcv_{rs}(p)$	$p \in rs$	remove one p from rs
$lose_{sr}(p)$	$p \in sr$	remove one p from sr	$lose_{rs}(p)$	$p \in rs$	remove one p from rs

Table 10.3. Actions of the channels

To define D we introduce the type *Mark* which has values in the set {+, #}. Table 10.2 gives the state and actions of D.

10.3.1 Proof that D Implements S

We do not give this proof, since to do it using abstraction functions we would have to introduce 'prophecy variables', also known as 'multi-valued mappings' or 'backward simulations' (Abadi and Lamport [1991], Lynch and Vaandrager [1993]). If you work out some examples, however, you will probably see why the two specs S and D have the same external behavior.

10.4 Channels

All our protocols use the same *channel* abstraction to transfer information between the sender and the receiver. We use the name 'packet' for the messages sent over a channel, to distinguish them from reliable messages. A channel can freely drop and reorder packets, and it can duplicate a packet any finite number of times when it's sent;[2] the only thing it isn't allowed to do is deliver a packet that wasn't sent. The reason for using such a weak specification is to ensure that the reliable message protocol will work over any bit-moving mechanism that happens to be available. With a stronger channel spec, for instance one that doesn't reorder packets, it's possible to have somewhat simpler or more efficient implementations.

There are two channels *sr* and *rs*, one from sender to receiver and one from receiver to sender, each a multiset of packets initially empty. The nature of a packet varies from one protocol to another. Table 10.3 gives the channel actions.

Protocols interact with the channels through the external actions *send*(...) and *rcv*(...) which have the same names in the channel and in the protocol. One of these actions occurs if both its pre-conditions are true, and the effect is both the effects. This always makes sense because the states are disjoint.

---

[2] You might think it would be more natural and closer to the actual implementation of a channel to allow a packet already in the channel to be duplicated. Unfortunately, if a packet can be duplicated any number of times it's possible that a protocol like H (see section 10.8) will not make any progress.

10.5 The Generic Protocol G

The generic protocol G generalizes two practical protocols described later, H and C; in other words, both of them implement G. This protocol can't be implemented directly because it has some 'magic' actions that use state from both sender and receiver. But both real protocols implement these actions, each in its own way.

The basic idea is derived from the simplest possible distributed implementation of S, which we call the stable protocol SB. In SB all the state is stable (that is, nothing is lost when there is a crash), and each end keeps a set g_s or g_r of good identifiers, that is, identifiers that have not yet been used. Initially $g_s \subseteq g_r$, and the protocol maintains this as an invariant. To send a message the sender chooses a good identifier i from g_s, attaches i to the message, moves i from g_s to a $last_s$ variable, and repeatedly sends the message. When the receiver gets a message with a good identifier it accepts the message, moves the identifier from g_r to a $last_r$ variable, and returns an ack packet for the identifier after the message has been delivered by *get*. When the receiver gets a message with an identifier that isn't good, it returns a positive ack if the identifier equals $last_r$ and the message has been delivered. The sender waits to receive an ack for $last_s$ before doing *getAck*(*OK*). There are never any negative acks, since nothing is ever lost.

This protocol satisfies the requirements of S; indeed, it does better since it never loses anything.

1. It provides at-most-once delivery because the sender never uses the same identifier for more than one message, and the receiver accepts an identifier and its message only once.
2. It provides FIFO ordering because at most one message is in transit at a time.
3. It delivers all the messages because the sender's good set is a subset of the receiver's.
4. It acks every message because the sender keeps retransmitting until it gets the ack.

The SB protocol is widely used in practice, under names that resemble 'queuing system'. It isn't used to establish connections because the cost of a stable storage write for each message is too great.

In G we have the same structure of good sets and *last* variables. However, they are not stable in G because we have to update them for every message, and we don't want to do a stable write for every message. Instead, there are operations to grow and shrink the good sets; these operations maintain the invariant $g_s \subseteq g_r$ as long as there is no receiver crash. When there is a crash, messages and acks can be lost, but S and D allow this. Figure 10.3 shows the state and some possible transitions of G in simplified form. The names in outline font are state variables of D, and the corresponding values are the values of the abstraction function in that state.

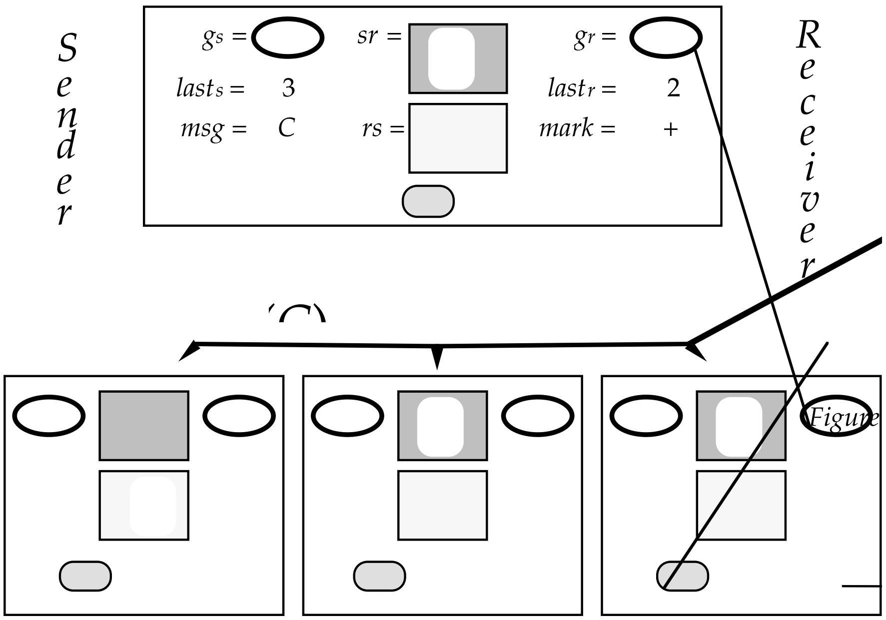

Figure 10.3. Some states and tra

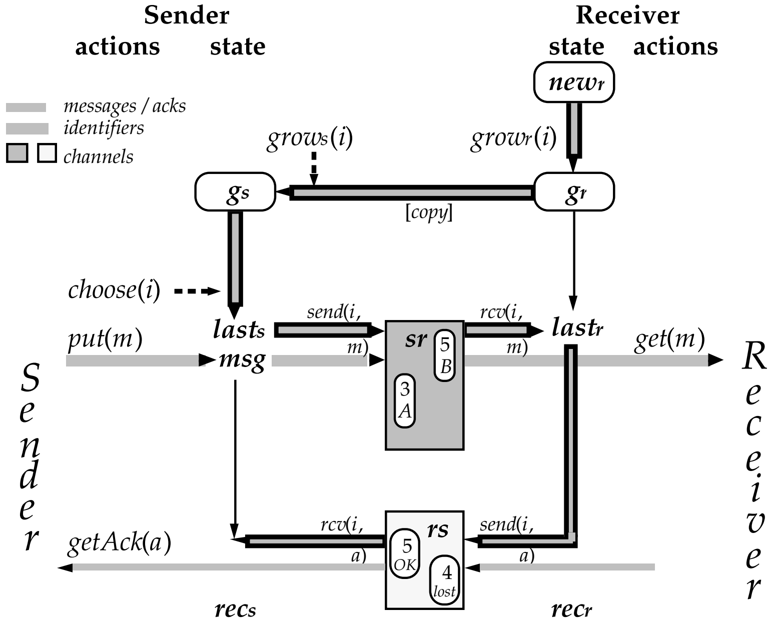

Figure 10.4. State, main actions, and information flow of G

> $last_r$ it optionally returns a negative ack. There is no point in doing this for a message with $i < last_r$ because the sender only cares about the ack for $last_s$, and the protocol maintains the invariant $last_r \le last_s$. If $i > last_r$, however, the receiver must sometimes send a negative ack in response so that the sender can find out that the message may have been lost.

G is organized into a set of implementable actions that also appear, with very minor variations, in both H and C, plus the magic *grow*, *shrink*, and *cleanup* actions that are simulated quite differently in H and in C.

When there are no crashes, the sender and receiver each go through a cycle of modes, the sender perhaps one mode ahead. In one cycle one message is sent and acknowledged. For the sender, the modes are *idle*, [*needI*], *send*; for the receiver, they are *idle* and *ack*. An agent that is not idle is busy. The bracketed mode is 'internal': it's possible to advance to the next mode without receiving another message. The modes are not explicit state variables, but instead are derived from the values of the *msg* and *last* variables, as follows:

$mode_s = idle$ iff $msg = nil$	$mode_r = idle$ iff $last_r = nil$
$mode_s = needI$ iff $msg \neq nil$ and $last_s = nil$	
$mode_s = send$ iff $msg \neq nil$ and $last_s \neq nil$	$mode_r = ack$ iff $last_r \neq nil$

To define G we introduce the types:

I, an infinite set of identifiers.

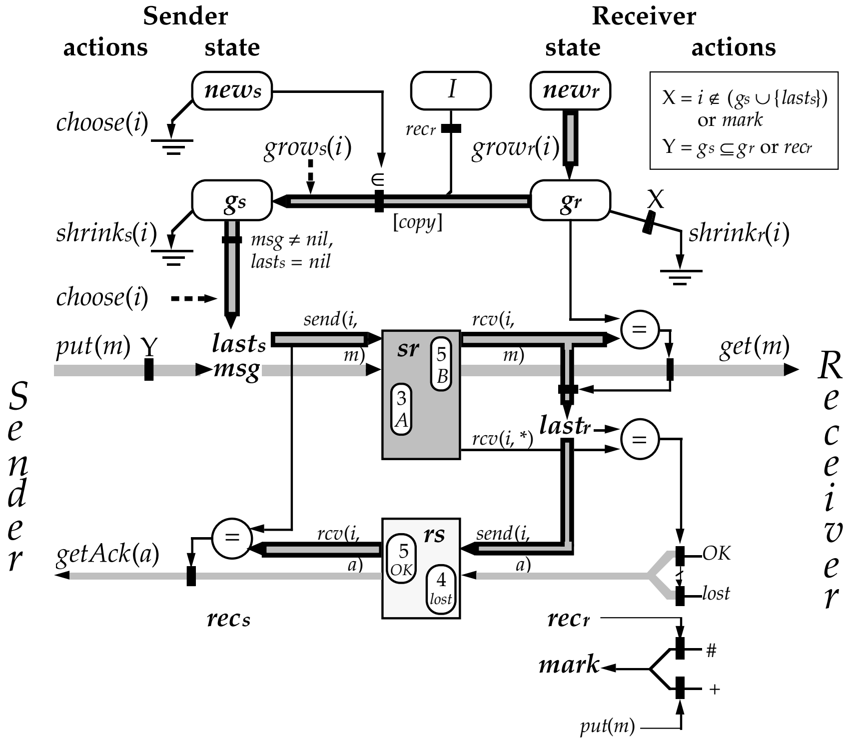

Figure 10.5. Details of actions and information flow in G

P (packet), a pair (I, M or A).

The sender sends (I, M) packets to the receiver, which sends (I, A) packets back. The I is there to identify the packet for the destination. We define a partial order on I by the rule that $i < i'$ iff i precedes i' in the sequence $used_S$.

The G we give is a somewhat simplified version, because the actions are not as atomic as they should be. In particular, some actions have two external interactions, sometimes one with a channel and one with the client, sometimes two with channels. However, the simplified version differs from one with the proper atomicity only in unimportant details. The appendix gives a version of G with all the fussy details in place. We don't give these details for the C and H protocols that follow, but content ourselves with the simplified versions in order to emphasize the important features of the protocols.

Figure 10.5 is a more detailed version of Figure 10.4, which shows all the actions and the flow of information between the sender and the receiver. State variables are given in bold, and the black guards on the transitions give the preconditions. The *mark* variable can be # when the receiver has recovered since a message was put; it reflects the fact that the message may be dropped.

Table 10.4 gives the state and actions of G. The magic parts, that is, those that touch non-local state, are boxed. The conjunct $\neg\, rec_S$ has been omitted from the

Name	**Guard**	**Effect**	**Name**	**Guard**	**Effect**
$^{**}\mathit{put}(m)$	$\mathit{msg} = \mathit{nil}$, $\boxed{g_s \subseteq g_r \text{ or } \mathit{rec}_r}$	$\mathit{msg} := m$, $\boxed{\mathit{mark} := +}$			
$\mathit{choose}(i)$	$\mathit{msg} \neq \mathit{nil}$, $\mathit{last}_s = \mathit{nil}$, $i \in g_s$	$g_s \mathrel{-:=} \{j \mid j \leq i\}$, $\mathit{last}_s := i$, $\mathit{used}_s \mathrel{+:=} \langle i \rangle$	$^{*}\mathit{get}(m)$	exists i such that $\mathit{rcv}_{sr}(i,m)$, $i \in g_r$	$g_r \mathrel{-:=} \{j \mid j \leq i\}$, $\mathit{last}_r := i$, $\mathit{send}_{rs}(i, \mathit{OK})$
send	$\mathit{last}_s \neq \mathit{nil}$	$\mathit{send}_{sr}(\mathit{last}_s, \mathit{msg})$			
$^{*}\mathit{getAck}(a)$	$\mathit{rcv}_{rs}(\mathit{last}_s, a)$	$\mathit{last}_s := \mathit{nil}$, $\mathit{msg} := \mathit{nil}$	$\mathit{sendAck}$	exists i such that $\mathit{rcv}_{sr}(i, {*})$, $i \notin g_r$	optionally send_{rs} (i, if $i = \mathit{last}_r$ then OK else lost)
$^{**}\mathit{crash}_s$		$\mathit{rec}_s := \mathit{true}$	$^{**}\mathit{crash}_r$		$\mathit{rec}_r := \mathit{true}$
$^{*}\mathit{recover}_s$	rec_s	$\mathit{last}_s := \mathit{nil}$, $\mathit{msg} := \mathit{nil}$, $\mathit{rec}_s := \mathit{false}$	$^{*}\mathit{recover}_r$	rec_r, $\boxed{\mathit{used}_r \supseteq g_s \cup \mathit{used}_s}$	$\mathit{last}_r := \mathit{nil}$, $\mathit{mark} := \#$, $\mathit{rec}_r := \mathit{false}$
$\mathit{shrink}_s(i)$		$g_s \mathrel{-:=} \{i\}$	$\mathit{shrink}_r(i)$	$\boxed{i \notin g_s,\ i \neq \mathit{last}_s}$ or $\mathit{mark} = \#$	$g_r \mathrel{-:=} \{i\}$
$\mathit{grow}_s(i)$	$i \notin \mathit{used}_s$, $\boxed{i \in g_r \text{ or } \mathit{rec}_r}$	$g_s \mathrel{+:=} \{i\}$	$\mathit{grow}_r(i)$	$i \notin \mathit{used}_r$	$g_r \mathrel{+:=} \{i\}$, $\mathit{used}_r \mathrel{+:=} \{i\}$
$\mathit{grow\text{-}used}_s(i)$	$i \notin \mathit{used}_s \cup g_s$, $\boxed{i \in \mathit{used}_r \text{ or } \mathit{rec}_r}$	$\mathit{used}_s \mathrel{+:=} \{i\}$	$\mathit{cleanup}$	$\boxed{\mathit{last}_r \neq \mathit{last}_s}$	$\mathit{last}_r := \mathit{nil}$
			unmark	$\boxed{g_s \subseteq g_r,\ \mathit{last}_s \in g_r \cup \{\mathit{last}_r, \mathit{nil}\}}$	$\mathit{mark} := +$

used_s	: sequence[I]	$:= \langle\ \rangle$ (stable)	used_r	: set[I]	$:= \{\ \}$ (stable)
g_s	: set[I]	$:= \{\ \}$	g_r	: set[I]	$:= \{\ \}$
last_s	: I or nil	$:= \mathit{nil}$	last_r	: I or nil	$:= \mathit{nil}$
msg	: M or nil	$:= \mathit{nil}$	mark	: Mark	$:= \#$
rec_s	: Boolean	$:= \mathit{false}$	rec_r	: Boolean	$:= \mathit{false}$

Table 10.4. State and actions of G

guards of all the sender actions except $\mathit{recover}_s$, and likewise for $\neg\, \mathit{rec}_r$ and the receiver actions.

In addition to meeting the spec S, this protocol has some other important properties:

- It makes progress: regardless of prior crashes, provided both ends stay up and the channels don't always lose messages, then if there's a message to send it is eventually sent, and otherwise both parties eventually become idle, the sender because it gets an ack, the receiver because eventually *cleanup* makes *mode* = *idle*. Progress depends on doing enough *grow* actions, and in particular on completing the sequence $\mathit{grow}_r(i)$, $\mathit{grow}_s(i)$, $\mathit{choose}(i)$.

- It's not necessary to do a stable storage operation for each message. Instead, the cost of a stable storage operation can be amortized over as many messages as you like. G has only two stable variables: $used_s$ and $used_r$. Different implementations of G handle $used_s$ differently. To reduce the number of stable updates to $used_r$, refine G to divide $used_r$ into the union of a stable $used_r$-*s* and a volatile $used_r$-*v*. Move a set of *I*s from $used_r$-*s* to $used_r$-*v* with a single stable update. The $used_r$-*v* becomes empty in $recover_r$; simulate this with $grow_r(i)$ followed immediately by $shrink_r(i)$ for every *i* in $used_r$-*v*.
- The only state required for an idle agent is the stable variable *used*. All the other (volatile) state is the same at the end of a message transmission as at the beginning. The sender forgets its state in *getAck*, the receiver in *cleanup*, and both in *recover*. The *shrink* actions make it possible for both parties to forget the good sets. This is important because agents may need to communicate with many other agents between crashes, and it isn't practical to require that an agent maintain some state for everyone with whom it has ever communicated.
- An idle sender doesn't send any packets. An idle receiver doesn't send any packets unless it receives one, because it sends an acknowledgement only in response to a packet. This is important because the channel resources shouldn't be wasted.

We have constructed G with as much non-determinism as possible in order to make it easy to prove that different practical protocols implement G. We could have simplified it, for instance by eliminating *unmark*, but then it would be more difficult to construct an abstraction function from some other protocol to G, since the abstraction function would have to account for the fact that after a $recover_r$ the *mark* variable is # until the next *put*. With *unmark*, an implementation of G is free to set *mark* back to + whenever the guard is true.

10.5.1 Abstraction Function to D

The abstraction function is an essential tool for proving that the protocol implements the spec. But it is also an important aid to understanding what is going on. By studying what happens to the value of the abstraction function during each action of G, we can learn what the actions are doing and why they work.

Definitions

cur-q = $\{(msg, mark)\}$ if $msg \neq nil$ and ($last_s = nil$ or $last_s \in g_r$)
$\{\ \}$ otherwise
$inflight_{sr}$ = $\{(i, m) \in \text{ids}(sr) \mid i \in g_r \text{ and } i \neq last_s\}$,
sorted by *i* to make a sequence
old-q = the sequence of $(M, Mark)$'s gotten by turning
each (i, m) in $inflight_{sr}$ into $(m, \#)$
$inflight_{rs}$ = $\{last_s\}$ if $(last_s, OK) \in rs$ and $last_s \neq last_r$
$\{\ \}$ otherwise.

Note that the *inflight*s exclude elements that might still be retransmitted as well as elements that are not of interest to the destination. This is so the abstraction function can pair them with the # mark.

Abstraction function

q	$\mathit{old\text{-}q} + \mathit{cur\text{-}q}$		
status	$(?, \mathit{mark})$	if $\mathit{cur\text{-}q} \neq \{\ \}$	(a)
	$(\mathit{OK}, +)$	if $\mathit{mode}_s = \mathit{send}$ and $\mathit{last}_s = \mathit{last}_r$	(b)
	$(\mathit{OK}, \#)$	if $\mathit{mode}_s = \mathit{send}$ and $\mathit{last}_s \in \mathit{inflight}_{rs}$	(c)
	$(\mathit{lost}, +)$	if $\mathit{mode}_s = \mathit{send}$ and $\mathit{last}_s \notin (g_r \cup \{\mathit{last}_r\} \cup \mathit{inflight}_{rs})$	(d)
	$(\mathit{lost}, +)$	if $\mathit{mode}_s = \mathit{idle}$	(e)
$\mathit{rec}_{s/r}$	$\mathit{rec}_{s/r}$		

The cases of *status* are exhaustive. Note that we do *not* want $(\mathit{msg}, +)$ in q if $\mathit{mode}_s = \mathit{send}$ and $\mathit{lasts}_s \notin g_r$, because in this case *msg* has been delivered or lost.

We see that G simulates the q of D using $\mathit{old\text{-}q} + \mathit{cur\text{-}q}$, and that *old-q* is the leftover messages in the channel that are still good but haven't been delivered, while *cur-q* is the message the sender is currently working on, as long as its identifier is not yet assigned or still good. Similarly, *status* has a different value for each step in the delivery process: still sending the message (a), normal ack (b), ack after a receiver crash (c), lost ack (d), or delivered ack (e).

10.5.2 Invariants

Like the abstraction function, the invariants are both essential to the proof and an important aid to understanding. They express a great deal of information about how the protocol is supposed to work. It's especially instructive to see how the parts of the state that have to do with crashes ($\mathit{rec}_{s/r}$ and *mark*) affect them.

The first few invariants establish some simple facts about the *used* sets and their relation to other variables. (G2) reflects that fact that identifers move from g_s to used_s one by one, (G3) the fact that unless the receiver is recovering, identifiers must enter used_r before they can appear anywhere else (G4) the fact that they must enter used_s before they can appear in *last* variables or channels.

If $\mathit{msg} = \mathit{nil}$ then $\mathit{last}_s = \mathit{nil}$ (G1)

$g_s \cap \mathit{used}_s = \{\ \}$ (G2a)

All elements of used_s are distinct. (G2b)

$\mathit{used}_r \supseteq g_r$ (G3a)

If $\neg\, \mathit{rec}_r$ then $\mathit{used}_r \supseteq g_s \cup \mathit{used}_s$ (G3b)

$\mathit{used}_s \supseteq \{\mathit{last}_s, \mathit{last}_r\} - \{\mathit{nil}\} \cup \mathrm{ids}(\mathit{sr}) \cup \mathrm{ids}(\mathit{rs})$ (G4)

The next invariants deal with the flow of identifiers during delivery. (G5) says that each identifier tags at most one message. (G6) says that if all is well, g_s and last_s are such that a message will be delivered and acknowledged properly. (G7) says that an identifier for a message being acknowledged can't be good.

$\{m \mid (i = last_s \text{ and } m = msg) \text{ or } (i, m) \in sr\}$ has 0 or 1 elements (G5)

If $mark = +$ and $\neg\, rec_s$ and $\neg\, rec_r$ then $g_s \subseteq g_r$ and $last_s \in g_r \cup \{last_r, nil\}$ (G6)

$g_r \cap (\{last_r\} \cup \text{ids}(rs)) = \{\ \}$ (G7)

Finally, some facts about the identifier $last_s$ for the message the sender is trying to deliver. It comes later in the identifier ordering than any other identifier in *sr* (G8a). If it's been delivered and is getting a positive ack, then neither it nor any other identifier in *sr* is in g_r, but they are all in $used_r$ (G8b). If it's getting a negative ack then it won't get a later positive one (G8c).

If $last_s \neq nil$ then

$\text{ids}(sr) \leq last_s$ (G8a)

and if $last_s = last_r$ or $(last_s, OK) \in rs$ then $(\{last_s\} \cup \text{ids}(sr)) \cap g_r = \{\ \}$ (G8b)
and $(\{last_s\} \cup \text{ids}(sr)) \subseteq used_r$

and if $(last_s, lost) \in \text{ids}(rs)$ then $last_s \neq last_r$ (G8c)

10.5.3 Proof that G Implements D

This requires showing that every action of G simulates some sequence of actions of D which is the same externally. Since G has quite a few actions, the proof is somewhat tedious. A few examples give the flavor.

—$recover_s$: Mark *msg* and drop it unless it moves to *old-q*; mark and drop *status*.

—*get*(*m*): For the change to *q*, first drop everything in *old-q* less than *i*. Then *m* is first on *q* since either *i* is the smallest *I* in *old-q*, or $i = last_s$ and *old-q* is empty by (G8a). So D's *get*(*m*) does the rest of what G's does. Everything in *old-q* + *cur-q* that was $\leq i$ is gone, so the corresponding *M*'s are gone from *q* as required.

We do *status* by the abstraction function's cases on its old value. D says it should change to (*OK*, *x*) iff *q* becomes empty and it was (?, *x*). In cases (c-e) *status* isn't (?, *x*) and it doesn't change. In case (b) the guard $i \in g_r$ of *get* is false by (G8b). In case (a) either $i = last_s$ or not. If not, then *cur-q* remains unchanged by (G8a), so *status* does also and *q* remains non-empty. If so, then *cur-q* and *q* both become empty and *status* changes to case (b). Simulate this by umarking *status* if necessary; then D's *get*(*m*) does the rest.

—*getAck*(*a*): The *q* is unchanged because $last_s = i \in \text{ids}(rs)$, so $last_s \notin g_r$ by (G7) and hence *cur-q* is empty, so changing *msg* to *nil* keeps it empty. Because *old-q* doesn't change, *q* doesn't either. We end up with *status* = (*lost*, +) according to case (e), as required by D. Finally, we must show that *a* agrees with the old value of *status*. We do this by the cases of *status* as we did for *get*:

(a) Impossible, because it requires $last_s \in g_r$, but we know $last_s \in \text{ids}(rs)$, which excludes $last_s \in g_r$ by (G7).

(b) In this case $last_s = last_r$, so (G8c) ensures $a \neq lost$, so $a = OK$.

(c) If $a = OK$ we are fine. If $a = lost$ drop *status* first.

(d) Since $last_s \notin inflight_{rs}$, only $(last_s, lost) \in rs$ is possible, so $a = lost$.

(e) Impossible because $last_s \neq nil$.

—*shrink*$_r$: If *rec*$_r$ then *msg* may be lost from q; simulate this by marking and dropping it, and likewise for *status*. If *mark* = # then *msg* may be lost from q, but it is marked, so simulate this by dropping it, and likewise for *status*. Otherwise the precondition ensures that $last_s \in g_r$ doesn't change, so *cur-q* and *status* don't. *Inflight*$_{sr}$, and hence *old-q*, can lose an element; simulate this by dropping the corresponding element of q, which is possible since it is marked #.

10.6 How C and H Implement G

We now proceed to give two practical protocols, the clock-based protocol C and the handshake protocol H. Each implements G, but they handle the good sets quite differently.

In C the good sets are maintained using time; to make this possible the sender and receiver clocks must be roughly synchronized, and there must be an upper bound on the time required to transmit a packet. The sender's current time $time_s$ is the only member of g_s; if the sender has already used $time_s$ then g_s is empty. The receiver accepts any message with an identifier in the range $(time_r - 2\varepsilon - \delta, time_r + 2\varepsilon)$, where ε is the maximum clock skew from real time and δ the maximum packet transmission time, as long as it hasn't already accepted a message with a later identifier.

In H the sender asks the receiver for a good identifier; the receiver's obligation is to keep the identifier good until it crashes or receives the message, or learns from the sender that the identifier will never be equal to $last_s$.

We begin by giving the abstraction functions from C and H to G, and a sketch of how each implements the magic actions of G, to help the reader in comparing the protocols. Careful study of these should make it clear exactly how each protocol implements G's magic actions in a properly distributed fashion.

Then for each protocol we give a figure that shows the flow of packets, followed by a formal description of the state and the actions. The portion of the figures that shows messages being sent and acks returned is exactly the same as the bottom half of Figure 10.4 for G; all three protocols handle messages and acks identically. They differ in how the sender obtains good identifiers, shown in the top of the figures, and in how the receiver cleans up its state. In the figures for C and H we show the abstraction function to G in outline font.

Note that G allows either good set to grow or shrink by any number of *I*s through repeated *grow* or *shrink* actions as long as the invariants $g_s \subseteq g_r$ and $last_s \in g_r \cup \{last_r\}$ are maintained in the absence of crashes. For C the *increase* actions simulate occurrences of several *grow*$_r$ and *shrink*$_r$ actions, one for each i in the set defined in the table. Likewise $rcv_{rs}(j_s, i)$ in H may simulate several *shrink*$_s$ actions.

Abstraction functions to G

G	**C**	**H**
$used_s$	$\{i \mid 0 \le i < time_s\} \cup \{sent\} - \{nil\}$	$used_s$ (history)
$used_r$	$\{i \mid 0 \le i < low\}$	$used_r$
g_s	$\{time_s\} - \{sent\}$	$\{i \mid (j_s, i) \in rs\}$
g_r	$\{i \mid low < i \text{ and } i < high\}$	$\{i_r\} - \{nil\}$
mark	# if $last_s \in g_r$ and $deadline = nil$ + otherwise	# if $mode_s = needI$ and $g_s \not\subseteq g_r$ + otherwise

msg, $last_{s/r}$, and $rec_{s/r}$ are the same in G, C, and H

sr	*sr*	the (I, M) messages in *sr*
rs	*rs*	the (I, A) messages in *rs*

Sketch of implementations

G	**C**	**H**
$grow_s(i)$	$tick(i)$	$send_{rs}(j_s, i)$
$shrink_s(i)$	$tick(i'), i \in \{time_s\} - \{sent\}$	$lose_{rs}(j_s, i)$ if the last copy is lost or $rcv_{rs}(j_s, i')$, for each $i \in g_s - \{i'\}$
$grow_r(i)$	$increase\text{-}high(i')$, for each $i \in \{i \mid high < i < i'\}$	$mode = idle$ and rcv_{sr}(needI, *)
$shrink_r(i)$	$increase\text{-}low(i')$, for each $i \in \{i \mid low < i \le i'\}$	$rcv_{sr}(i_r,$ done)
cleanup	*cleanup*	$rcv_{sr}(last_r,$ done)

10.7 The Clock-Based Protocol C

This protocol is due to Liskov, Shrira, and Wroclawski [1991]. Figure 10.6 shows the state and the flow of information. Compare it with Figure 10.4 for G, and note that there is no flow of new identifiers from receiver to sender. In C the passage of time supplies the sender with new identifiers, and is also allows the receiver to clean up its state.

The idea behind C is to use loosely synchronized clocks to provide the identifiers for messages. The sender uses its current *time* for the next identifier. The receiver keeps track of *low*, the biggest clock value for which it has accepted a message: bigger values than this are good. The receiver also keeps a stable bound *high* on the biggest value it will accept, chosen to be larger than the receiver's clock plus the maximum clock skew. After a crash the receiver sets $low := high$; this ensures that no messages are accepted twice.

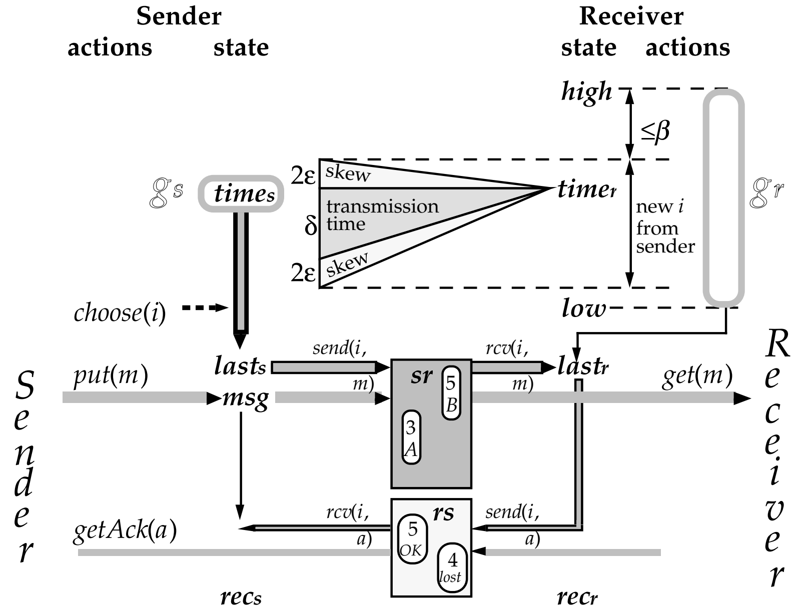

Figure 10.6. The flow of information in C

The sender's clock advances, which ensures that it will get new identifiers and also ensures that it will eventually get past *low* and start sending messages that will be accepted after a receiver crash.

It's also possible for the receiver to advance *low* spontaneously (by *increase-low*) if it hasn't received a message for a long time, as long as *low* stays smaller than the current time – 2ε – δ, where ε is the maximum clock skew from real time and δ is the maximum packet transmission time. This is good because it gives the receiver a chance to run several copies of the protocol (one for each of several senders), and make the values of *low* the same for all the idle senders. Then the receiver only needs to keep track of a single *low* for all the idle senders, plus one for each active sender. Together with C's *cleanup* action this ensures that the receiver needs no storage for idle senders.

If the assumptions about clock skew and maximum packet transmission time are violated, C still provides at-most-once delivery, but it may lose messages (because *low* is advanced too soon or the sender's clock is later than *high*) or acknowledgements (because *cleanup* happens too soon).

Modes, types, packets, and the pattern of messages are the same as in G, except that the *I* set has a total ordering. The *deadline* variable expresses the assumption about maximum packet delivery time: real time doesn't advance (by *progress*) past the deadline for delivering a packet. In a real implementation, of course, there will be some other properties of the channel from which the constraint imposed by *deadline* can be deduced. These are usually probabilistic; we deal with this by declaring a *crash* whenever the channel fails to meet its deadline.

Name	Guard	Effect	Name	Guard	Effect
***put*(*m*)	$msg = nil$	$msg := m$			
choose(*i*)	$msg \neq nil$, $last_s = nil$, $i{=}time_s$, $i{\neq}sent$	$sent := i$, $last_s := i$, $deadline := now{+}\delta$	**get*(*m*)	exists i such that $rcv_{sr}(i, m)$, $i \in (low..high)$	$low := i$, $last_r := i$, $deadline := nil$, $send_{rs}(i, OK)$
send	$last_s \neq nil$	$send_{sr}(last_s, msg)$			
**getAck*(*a*)	$rcv_{rs}(last_s, a)$	$last_s := nil$, $msg := nil$	*sendAck*	exists i such that $rcv_{sr}(i, *)$, $i \notin (low..high)$	$low := \max(low, i)$, $send_{rs}(i$, if $i = last_r$ then OK else $lost$) if $i = last_s$ then $deadline := nil$
***crash*$_s$		$rec_s := true$, $deadline := nil$	***crash*$_r$		$rec_r := true$, $deadline := nil$
**recover*$_s$	rec_s	$last_s := nil$, $msg := nil$, $rec_s := false$	**recover*$_r$	rec_r, $high < time_r - 2\varepsilon$	$last_r := nil$, $low := high$, $high := time_r + 2\varepsilon + \beta$, $rec_r := false$
			increase-low(*i*)	$low < i \leq time_r - 2\varepsilon - \delta$	$low := i$
			increase-high(*i*)	$high < i \leq time_r + 2\varepsilon + \beta$	$high := i$
cleanup	$sent \neq time_s$	$sent := nil$	*cleanup*	$last_r < time_r - 2\varepsilon - 2\delta$	$last_r := nil$
tick(*i*)	$time_s < i$, $\lvert now - i \rvert < \varepsilon$	$time_s := i$	*tick*(*i*)	$time_r < i$, $\lvert now - i \rvert < \varepsilon$, $i + 2\varepsilon < high$ or rec_r	$time_r := i$
progress(*i*)	$now < i$, $\lvert i - time_{s/r} \rvert < \varepsilon$, $i < deadline$ or $deadline{=}nil$	$now := i$			

$time_s$	: I	:= 0 (stable)	$time_r$	: I	:= 0 (stable)
$sent$	: I or nil	:= nil	low	: I	:= 0
			$high$	: I	:= β (stable)
$last_s$	: I or nil	:= nil	$last_r$	: I or nil	:= nil
msg	: M or nil	:= nil			
rec_s	: Boolean	:= $false$	rec_r	: Boolean	:= $false$

$deadline$: I or nil := nil

now : I := 0

Table 10.5. State and actions of C. Actions below the thick line handle the passage of time.

Table 10.5 gives the state and actions of C. The conjunct $\neg$ *rec_s* has been omitted from the guards of all the sender actions except *$recover_s$*, and likewise for $\neg$ *rec_r* and the receiver actions.

Note that like G, this version of C sends an ack only in response to a message. This is unlike H, which has continuous transmission of the ack and pays the price of a *done* message to stop it. Another possibility is to make timing assumptions about *rs* and time out the ack; some assumptions are needed anyway to make *cleanup* possible. This would be less practical but more like H.

Note that $time_s$ and $time_r$ differ from real time (*now*) by at most ε, and hence $time_s$ and $time_r$ can differ from each other by as much as 2ε. Note also that the deadline is enforced by the *progress* action, which doesn't allow real time to advance past the deadline unless someone is recovering. Both $crash_s$ and $crash_r$ cancel the deadline.

About the parameters of C

The protocol is parameterized by three constants:

- δ = maximum time to deliver a packet
- β = amount beyond $time_r + 2\varepsilon$ to increase *high*
- ε = maximum of $|now - time_{r/s}|$

These parameters must satisfy two constraints:

- $\delta > \varepsilon$ so that $mode_s = send$ implies $last_s < deadline$.
- $\beta > 0$ so *increase-high* can be enabled. Aside from this constraint the choice of β is just a tradeoff between the frequency of stable storage writes (at least one every β, so a bigger β means fewer writes) and the delay imposed on $recover_r$ to ensure that messages put after $recover_r$ don't get dropped (as much as $4\varepsilon + \beta$, because *high* can be as big as $time_r + 2\varepsilon + \beta$ at the time of the crash because of (e), and $time_r - 2\varepsilon$ has to get past this via $tick_r$ before $recover_r$ can happen, so a bigger β means a longer delay).

10.7.1 Invariants

Mostly these are facts about the ordering of various time variables; a lot of $x \neq nil$ conjuncts have been omitted. Nothing being sent is later than $time_s$ (C1). Nothing being acknowledged is later than *low*, which is no later than *high*, which in turn is big enough (C2). Nothing being sent or acknowledged is later than $last_s$ (C3). The sender's time is later than *low*, hence good unless equal to *sent* (C4).

$last_s \leq time_s$ (C1)

$last_r \leq low \leq high$ (C2a)

$\mathrm{ids}(rs) \leq low$ (C2b)

If $\neg rec_r$ then $time_r + 2\varepsilon \leq high$ (C2c)

$\mathrm{ids}(sr) \leq last_s$ (C3a)

$last_r \leq last_s$ (C3b)

$\{i \mid (i, OK) \in rs\} \leq last_s$ (C3c)

$low \leq time_s$ (C4)

$low < time_s$ if $last_s \neq time_s$

If a message is being sent but hasn't been delivered, and there hasn't been a crash, then *deadline* gives the deadline for delivering the packet containing the message (based on the maximum time for a packet that is being retransmitted to get through *sr*), and it isn't too late for it to be accepted (C5).

If $deadline \neq nil$ then

$now < last_s + \varepsilon + \delta$ (C5a)

$low < last_s$ (C5b)

An identifier getting a positive ack is no later than *low*, hence no longer good (C6). If it's getting a negative ack, it must be later than the last one accepted (C7).

If $(last_s, OK) \in rs$ then $last_s \leq low$ (C6)

If $(last_s, lost) \in rs$ then $last_r < last_s$ (C7)

10.8 The Handshake Protocol H

This is the standard protocol for setting up network connections, used in TCP, ISO TP-4, and many other transport protocols. It is usually called three-way handshake, because only three packets are needed to get the data delivered, but five packets are required to get it acknowledged and all the state cleaned up (Belsnes [1976]).

As in the generic protocol, when there are no crashes the sender and receiver each go through a cycle of modes, the sender perhaps one ahead. For the sender, the modes are *idle*, *needI*, *send*; for the receiver, they are *idle*, *accept*, and *ack*. In one cycle one message is sent and acknowledged by sending three packets from sender to receiver and two from receiver to sender, for a total of five packets. Table 10.6 summarizes the modes and the packets that are sent.

The modes are derived from the values of the state variables *j* and *last*:

$mode_s = idle$ iff $j_s = last_s = nil$ $\quad$ $mode_r = idle$ iff $j_r = last_r = nil$

$mode_s = needI$ iff $j_s \neq nil$ $\quad$ $mode_r = accept$ iff $j_r \neq nil$

$mode_s = send$ iff $last_s \neq nil$ $\quad$ $mode_r = ack$ iff $last_r \neq nil$

Figure 10.7 shows the state, the flow of identifiers from the receiver to the sender at the top, and the flow of *done* information back to the receiver at the bottom so that it can clean up. These are sandwiched between the standard exchange of message and ack, which is the same as in G (see Figure 10.4).

Intuitively, the reason there are five packets is that:

- One round-trip (two packets) is needed for the sender to get from the receiver an *I* (namely i_r) that both know has not been used.
- One round-trip (two packets) is then needed to send and ack the message.

	Sender				**Receiver**	
mode	**send**	**advance on**	**packet**	**advance on**	**send**	**mode**
idle	see *idle* below	*put*, to *needI*			(i, *lost*) when (i, m) arrives[3]	*idle*
needI	(needI, j_s) repeatedly		(needI, j) →	(needI, j) arrives, to *accept*		
		(j_s, i) arrives, to *send*	(j, i) ←		(j_r, i_r) repeatedly	*accept*
send	($last_s$, m) repeatedly		(i, m) →	(i_r, m) arrives, to *ack* (i_r, done) arrives, to *idle*		
		($last_s$, a) arrives, to *idle*	(i, a) ←		($last_r$, *OK*) repeatedly[4]	*ack*
idle	(i, done) when (i, a) arrives		(i, done) →	($last_r$, done) arrives, to *idle*		
needI or *send*	(i, done) when ($j \neq j_s$, i) or (i, *OK)* arrives, to force receiver to *idle*					

Table 10.6. Exchange of messages in H

- A final *done* packet from the sender informs the receiver that the sender has gotten the ack. The receiver needs this information in order to stop retransmitting the ack and discard its state. If the receiver discards its "I got the message" state before it knows that the sender got the ack, then if the channel loses the ack the sender won't be able to find out that the message was actually received, even though there was no crash. This is contrary to the spec S. The *done* packet itself needs no ack, because the sender will also send it when idle and hence can become *idle* as soon as it sees the *ack*.

We introduce a new type:

J, an infinite set of identifiers that can be compared for equality.

The sender and receiver send packets to each other. An *I* or *J* in the first component is there to identify the packet for the destination. Some packets also have an *I* or *J* as the second component, but it does not identify anything; rather it is being communicated to the destination for later use. The (i, a) and (i, done) packets are both often called 'close' packets in the literature.

---

[3] (i, *lost*) is a negative acknowledgement; it means that one of two things has happened:

— The receiver has forgotten about i because it has learned that the sender has gotten a positive ack for i, but then the receiver has gotten a duplicate (i, m), to which it responds with the negative ack, which the sender will ignore.

— The receiver has crashed since it assigned i, and i's message may have been delivered to *get* or may have been lost.

[4] (i, *OK*) is a positive acknowledgement; it means i's message was delivered to *get*.

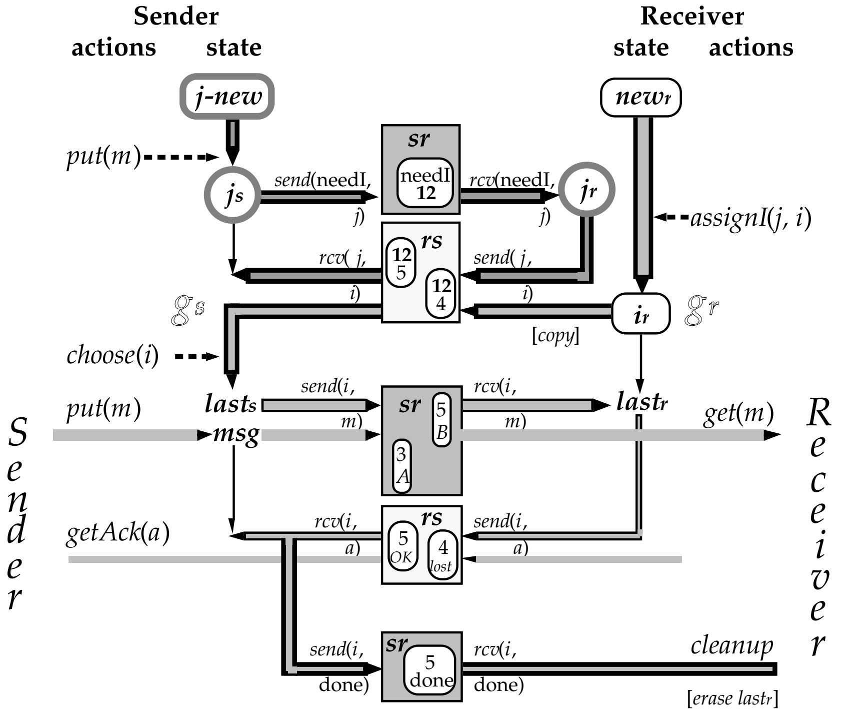

Figure 10.7. The flow of information in H

The H protocol has the same progress and efficiency properties as G, and in addition, although the protocol as given does assume an infinite supply of *I*s, it does not assume anything about clocks.

It's necessary for a busy agent to send something repeatedly, because the other end might be idle and therefore not sending anything that would get the busy agent back to idle. An agent also has a set of expected packets, and it wants to receive one of these in order to advance normally to the next mode. To ensure that the protocol is self-stabilizing after a crash, both ends respond to an unexpected packet containing the identifier *i* by sending an acknowledgement: (*i, lost*) or (*i,* done). Whenever the receiver gets *done* for its current *I*, it becomes idle. Once the receiver is idle, the sender advances normally until it too becomes idle.

Table 10.7 gives the state and actions of H. The conjunct $\neg\, rec_s$ has been omitted from the guards of all the sender actions except $recover_s$, and likewise for $\neg\, rec_r$ and the receiver actions.

10.8.1 Invariants

Recall that ids(c) is $\{i \mid (i, *) \in c\}$. We also define jds(c) = $\{j \mid (j, *) \in c$ or $(*, j) \in c\}$.

Most of H's invariants are boring facts about the progress of *I*'s and *J*'s from *used* sets through $i/j_{s/r}$ to $last_{s/r}$. We need the history variables $used_s$ and *seen* to

Name	Guard	Effect	Name	Guard	Effect
$^{**}\mathit{put}(m)$	$\mathit{msg} = \mathit{nil},$	$\mathit{msg} := m,$			
	exists j such that $j \notin \mathit{j\text{-}used}$	$j_s := j,$ $\mathit{j\text{-}used} \mathrel{+}:= \{j\}$			
requestI	$j_s \neq \mathit{nil},$ $\mathit{last}_s = \mathit{nil}$	send_{sr}(needI, j_s)	$\mathit{assignI}(j, i)$	rcv_{sr}(needI, j), $i_r = \mathit{last}_r = \mathit{nil},$ $i \notin \mathit{used}_r$	$j_r := j,\ i_r := i,$ $\mathit{used}_r \mathrel{+}:= i,$ $\mathit{seen} \mathrel{+}:= \{(j, i)\}$
$\mathit{choose}(i)$	$\mathit{last}_s = \mathit{nil},$ $\mathit{rcv}_{rs}(j_s, i)$	$j_s := \mathit{nil},\ \mathit{last}_s := i,$ $\mathit{used}_s \mathrel{+}:= \langle i \rangle$	*sendI*	$j_r \neq \mathit{nil}$	$\mathit{send}_{rs}(j_r, i_r)$
send	$\mathit{last}_s \neq \mathit{nil}$	$\mathit{send}_{sr}(\mathit{last}_s, \mathit{msg})$	$^{*}\mathit{get}(m)$	exists i such that $\mathit{rcv}_{sr}(i, m)$, $i = i_r$	$j_r := i_r := \mathit{nil},$ $\mathit{last}_r := i,$ $\mathit{send}_{rs}(i, \mathit{OK})$
			sendAck	$\mathit{last}_r \neq \mathit{nil}$	$\mathit{send}_{rs}(\mathit{last}_r, \mathit{OK})$
$^{*}\mathit{getAck}(a)$	$\mathit{rcv}_{rs}(\mathit{last}_s, a)$	if $a = \mathit{OK}$ then $\mathit{send}_{sr}(\mathit{last}_s, \mathit{done})$ $\mathit{msg} := \mathit{last}_s := \mathit{nil}$	*bounce*	exists i such that $\mathit{rcv}_{sr}(i, *)$, $i \neq i_r,\ i \neq \mathit{last}_r$	$\mathit{send}_{rs}(i, \mathit{lost})$
bounce (j, i)	$\mathit{rcv}_{rs}(j, i)$, $j \neq j_s,\ i \neq \mathit{last}_s$ or $\mathit{rcv}_{rs}(i, \mathit{OK})$	$\mathit{send}_{sr}(i$, done)	$\mathit{cleanup}(i)$	$\mathit{rcv}_{sr}(i$, done), $i = i_r$ or $i = \mathit{last}_r$	$j_r := i_r := \mathit{nil},$ $\mathit{last}_r := \mathit{nil}$
$^{**}\mathit{crash}_s$		$\mathit{rec}_s := \mathit{true}$	$^{**}\mathit{crash}_r$		$\mathit{rec}_r := \mathit{true}$
$^{*}\mathit{recover}_s$	rec_s	$\mathit{msg} := \mathit{nil},$ $j_s := \mathit{last}_s := \mathit{nil},$ $\mathit{rec}_s := \mathit{false}$	$^{*}\mathit{recover}_r$	rec_r	$j_r := i_r := \mathit{nil},$ $\mathit{last}_r := \mathit{nil},$ $\mathit{rec}_r := \mathit{false}$
grow-j-used(j)		$\mathit{j\text{-}used} \mathrel{+}:= \{j\}$	*grow-used*(i)		$\mathit{used}_r \mathrel{+}:= \{i\}$

used_s	: sequence[I]	:= ⟨ ⟩ (history)	used_r	: set[I]	:= { } (stable)
$\mathit{j\text{-}used}$	: set[J]	:= { } (stable)	seen	: set[(J, I)]	:= { } (history)
j_s	: J or nil	:= nil	j_r	: J or nil	:= nil
msg	: M or nil	:= nil	i_r	: I or nil	:= nil
last_s	: I or nil	:= nil	last_r	: I or nil	:= nil
rec_s	: Boolean	:= false	rec_r	: Boolean	:= false

Table 10.7. State and actions of H. Heavy black lines outline additions to G

express some of them. (H6) says that there's at most one J (from a needI packet) that gets assigned a given I. (H8) says that as long as the sender is still in mode *needI*, nothing involving i_r has made it into the channels.

$$\mathit{j\text{-}used} \supseteq \{j_s, j_r\} - \{\mathit{nil}\} \cup \mathrm{jds}(sr) \cup \mathrm{jds}(rs) \tag{H1}$$

$$\mathit{used}_r \supseteq \{i_r, \mathit{last}_r\} - \{\mathit{nil}\} \cup \mathit{used}_s \cup \{i \mid (*, i) \in rs\} \cup \mathrm{ids}(sr) \cup \mathrm{ids}(rs) \tag{H2}$$

$$\mathit{used}_s \supseteq \{\mathit{last}_s, \mathit{last}_r\} - \{\mathit{nil}\} \cup \mathrm{ids}(sr) \cup \mathrm{ids}(rs) \tag{H3}$$

$$\text{If } (i, \mathit{done}) \in sr \text{ then } i \neq \mathit{last}_s \tag{H4}$$

$$\text{If } i_r \neq \mathit{nil} \text{ then } (j_r, i_r) \in \mathit{seen} \tag{H5}$$

If $(j, i) \in seen$ and $(j', i) \in seen$ then $j = j'$ (H6)

If $(j, i) \in rs$ then $(j, i) \in seen$ (H7)

If $j_s = j_r \neq nil$ then $(i_r, *) \notin sr$ and $(i_r, done) \notin rs$ (H8)

10.8.2 Progress

We consider first what happens without failures, and then how the protocol recovers from failures.

If neither partner fails, then both advance in sync through the cycle of modes. The only thing that derails progress is for some party to change *mode* without advancing through the full cycle of modes that transmits a message. This can only happen when the receiver is in *accept* mode and gets (i_r, done), as you can see from Table 10.6. This can only happen if the sender got a packet containing i_r. But if the receiver is in *accept*, the sender must be in *needI* or *send*, and the only thing that's been sent with i_r is (j_s, i_r). The sender goes to or stays in *send* and doesn't make *done* when it gets (j_s, i_r) in either of these modes, so the cycling through the modes is never disrupted as long as there's no crash.

If either partner fails and then recovers, the other becomes idle rather than getting stuck; in other words, the protocol is self-stabilizing. Why? When the receiver isn't idle it always sends something, and if that isn't what the sender wants, the sender responds *done*, which forces the receiver to become idle. When the sender isn't idle it's either in *needI*, in which case it will eventually get what it wants, or it's in *send* and will get a negative ack and become idle. In more detail:

The receiver bails out when the sender crashes because

- the sender forgets i_s and j_s when it crashes,
- if the receiver isn't idle, it keeps sending (j_r, i_r) or $(last_r, OK)$,
- the sender responds with ($i_r/last_r$, done) whenever it sees either of these, and
- the receiver ends up in *idle* whenever it receives this.

The sender bails out or makes progress when the receiver crashes because

- If the sender is in *needI*, either
  —it gets $(j_s, i \neq i_r)$ from the pre-crash receiver, advances to *send*, and bails out as below, or
  —it gets (j_s, i_r) from the post-crash receiver and proceeds normally.
- If the sender is in *send* it keeps sending $(last_s, msg)$,
  —the receiver has $last_r = nil \neq last_s$, so it responds $(last_s, lost)$, and
  —when the sender gets this it becomes *idle*.

An idle receiver might see an old (needI, j) with $j \neq j_s$ and go into *accept* with $j_r \neq j_s$, but the sender will respond to the resulting (j_r, i_r) packets with (i_r, done), which will force the receiver back to *idle*. Eventually all the old *needI* packets will drain out. This is the reason that it's necessary to prevent a channel from delivering an unbounded number of copies of a packet.

Name	**Guard**	**Effect**
recycle(i) for G	$\boxed{\begin{array}{l} i \notin g_s \cup g_r \cup \{last_s, last_r\} \\ \quad \cup \text{ids}(sr) \cup \text{ids}(rs) \end{array}}$	$used_s \mathrel{-:=} \{i\}$, $used_r \mathrel{-:=} \{i\}$
recycle(i) for H	$\boxed{\begin{array}{l} i \notin \{last_s, i_r, last_r\} \\ \quad \cup \{i \mid (*, i) \in rs\} \cup \text{ids}(sr) \cup \text{ids}(rs) \end{array}}$	$used_s \mathrel{-:=} \{i\}$, $used_r \mathrel{-:=} \{i\}$, $seen \mathrel{-:=} \{j \mid (j, i) \in seen \mid (j, i)\}$
recycle-j(j) for H	$\boxed{j \notin \{j_s, j_r\} \cup \text{jds}(sr) \cup \text{jds}(rs)}$	$used\text{-}j \mathrel{-:=} \{j\}$, $seen \mathrel{-:=} \{i \mid (j, i) \in seen \mid (j, i)\}$

Table 10.8. Actions to recycle identifiers

10.9 Finite Identifiers

So far we have assumed that the identifier sets I and J are infinite. Practical protocols use sets that are finite and often quite small. We can easily extend G to use finite sets by adding a new action *recycle*(i) that removes an identifier from $used_s$ and $used_r$ so that it can be added to g_r again. As we saw in Section 10.1, when we add a new action the only change we need in the proof is to show that it maintains the invariants and simulates something in the spec. The latter is simple: *recycle* simulates no change in the spec. The former is also simple: we put a strong enough guard on *recycle* to ensure that all the invariants still hold. To find out what this guard is we need only find all the invariants that mention $used_s$ or $used_r$, since those are the only variables that *recycle* changes. Intuitively, the result is that an identifier can be recycled if it doesn't appear anywhere else in the variables or channels.

Similar observations apply to H, with some minor complications to keep the history variable *seen* up to date, and a similar *recycle-j* action. Table 10.8 gives the *recycle* actions for G and H.

How can we implement the guards on the *recycle* actions? The tricky part is ensuring that i is not still in a channel, since standard methods can ensure that it isn't in a variable at the other end. There are three schemes that are used in practice:

- Use a FIFO channel. Then a simple convention ensures that if you don't send any i_1's after you send i_2, then when you get back the ack for i_2 there aren't any i_1's left in either channel.
- Assume that packets in the channel have a maximum lifetime once they have been sent, and wait longer than that time after you stop sending packets containing i.
- Encrypt packets on the channel, and change the encryption key. Once the receiver acknowledges the change it will no longer accept packets encrypted with the old key, so these packets are in effect no longer in the channel.

For C we can recycle identifiers by using time modulo some period as the identifier, rather than unadorned time. Similar ideas apply; we omit the details.

10.10 Conclusions

We have given a precise specification S of reliable at-most-once message delivery with acknowledgements. We have also presented precise descriptions of two practical protocols (C and H) that implement S, and the essential elements of proofs that they do so; the handshake protocol H is used for connection establishment in most computer networking. Our proofs are organized into three levels: we refine S first into another specification D that delays some of the decisions of S and then into a generic implementation G, and finally we show that C and H both implement G. Most of the work is in the proof that G implements D.

In addition to complete expositions of the protocols and their correctness, we have also given an extended example of how to use abstraction functions and invariants to understand and verify subtle distributed algorithms of some practical importance. The example shows that the proofs are not too difficult and that the invariants, and especially the abstraction functions, give a great deal of insight into how the implementations work and why they satisfy the specifications. It also illustrates how to divide a complicated problem into parts that make sense individually and can be attacked one at a time.

References

Abadi, M. and Lamport, L. (1991), The Existence of Refinement Mappings, *Theoretical Computer Science* **82** (2), 253-284.

Belsnes, D. (1976), Single Message Communication, *IEEE Trans. Communications* **COM-24**, 2.

Lampson, B., Lynch, N., and Søgaard-Andersen, J. (1993), Reliable At-Most-Once Message Delivery Protocols, Technical Report, MIT Laboratory for Computer Science, to appear.

Liskov, B., Shrira, L., and Wroclawski, J. (1991), Efficient At-Most-Once Messages Based on Synchronized Clocks, *ACM Trans. Computer Systems* **9** (2), 125-142.

Lynch, N. and Vaandrager, F. (1993), Forward and Backward Simulations, Part I: Untimed Systems, Technical Report, MIT Laboratory for Computer Science, to appear.

Appendix

For reference we give the complete protocol for G, with every action as atomic as it should be. This requires separating the getting and putting of messages, the sending and receiving of packets, the sending and receiving of acks, and the getting of acks. As a result, we have to add buffer queues $buf_{s/r}$ for messages at both ends, a buffer variable *ack* for the ack at the sender, and a *send-ack* flag for positive acks and a buffer *nack-buf* for negative acks at the receiver.

The state of the full G is:

$used_s$	: sequence[I]	:= $\langle\,\rangle$ (stable)	$used_r$	: set[I]	:= { } (stable)
g_s	: set[I]	:= { }	g_r	: set[I]	:= { }
$last_s$	: I or *nil*	:= *nil*	$last_r$	: I or *nil*	:= *nil*
buf_s	: sequence[M]	:= $\langle\,\rangle$	buf_r	: sequence[M]	:= $\langle\,\rangle$
msg	: M or *nil*	:= *nil*	*mark*	: + or #	:= +
ack	: A	:= *lost*	*send-ack*	: Boolean	:= *false*
			nack-buf	: sequence[I]	:= $\langle\,\rangle$
rec_s	: Boolean	:= *false*	rec_r	: Boolean	:= *false*

The abstraction function to D is:

q		the elements of buf_r paired with +		
		+ *old-q* + *cur-q*		
		+ the elements of buf_s paired with +		
status		(?, +)	if $buf_s \neq empty$	
	else	(?, *mark*)	if $cur\text{-}q \neq \{\,\}$	(a)
		(?, +)	if $mode_s = send$, $last_s = last_r$, $buf_r \neq \{\,\}$	(b)
		(*OK*, +)	if $mode_s = send$, $last_s = last_r$, $buf_r = \{\,\}$	(c)
		(*OK*, #)	if $mode_s = send$ and $last_s \in inflight_{rs}$	(d)
		(*lost*, +)	if $mode_s = send$ and $last_s \notin (g_r \cup \{last_r\} \cup inflight_{rs})$	(e)
		(*ack*, +)	if $mode_s = idle$	(f)
$rec_{s/r}$	$rec_{s/r}$			

Name	**Guard**	**Effect**	**Name**	**Guard**	**Effect**
$^{**}\mathit{put}(m)$		append m to buf_s			
$\mathit{prepare}(m)$	$\mathit{msg} = \mathit{nil}$, m first on buf_s, $\boxed{g_s \subseteq g_r \text{ or } \mathit{rec}_r}$	buf_s:=tail (buf_s), $\mathit{msg} := m$, $\boxed{\mathit{mark} := +}$			
$\mathit{choose}(i)$	$\mathit{msg} \neq \mathit{nil}$, $\mathit{last}_s = \mathit{nil}$, $i \in g_s$	$g_s -:= \{j \mid j \leq i\}$, $\mathit{last}_s := i$, $\mathit{used}_s +:= \langle i \rangle$			
$\mathit{send}_{sr}(i, m)$	$i = \mathit{last}_s \neq \mathit{nil}$ $m = \mathit{msg}$		$\mathit{rcv}_{sr}(i, m)$	if $i \in g_r$ then append m to buf_r, $\mathit{sendAck} := \mathit{false}$, $g_r -:= \{j \mid j \leq i\}$, $\mathit{last}_r := i$, else if $i \notin g_r \cup \{\mathit{last}_r\}$ then optionally $\mathit{nack\text{-}buf} +:= \langle i \rangle$ else if $i = \mathit{last}_r$ then $\mathit{sendAck} := \mathit{true}$	
			$^{*}\mathit{get}(m)$	m first on buf_r	if $\mathit{buf}_r = \langle m \rangle$ then $\mathit{sendAck} := \mathit{true}$, buf_r :=tail (buf_r)
$\mathit{rcv}_{rs}(i, a)$	if $i = \mathit{last}_s$ then $\mathit{ack} := a$, $\mathit{msg} := \mathit{nil}$, $\mathit{last}_s := \mathit{nil}$		send_{rs} (i, OK)	$i = \mathit{last}_r$, $\mathit{sendAck}$	optionally $\mathit{sendAck} := \mathit{false}$
			send_{rs} (i, lost)	i first on $\mathit{nack\text{-}buf}$	$\mathit{nack\text{-}buf} :=$ tail ($\mathit{nack\text{-}buf}$)
$^{*}\mathit{getAck}(a)$	$\mathit{msg} = \mathit{nil}$, $\mathit{buf}_s = \mathit{empty}$, $\mathit{ack} = a$	$\mathit{ack} := \mathit{lost}$			
$^{**}\mathit{crash}_s$		$\mathit{rec}_s := \mathit{true}$	$^{**}\mathit{crash}_r$		$\mathit{rec}_r := \mathit{true}$
$^{*}\mathit{recover}_s$	rec_s	$\mathit{last}_s := \mathit{nil}$, $\mathit{msg} := \mathit{nil}$, $\mathit{buf}_s := \langle\,\rangle$, $\mathit{ack} := \mathit{lost}$, $\mathit{rec}_s := \mathit{false}$	$^{*}\mathit{recover}_r$	rec_r, $\boxed{\mathit{used}_r \supseteq g_s \cup \mathit{used}_s}$	$\mathit{last}_r := \mathit{nil}$, $\mathit{mark} := \#$, $\mathit{buf}_r := \langle\,\rangle$, $\mathit{nack\text{-}buf} := \langle\,\rangle$, $\mathit{rec}_r := \mathit{false}$
$\mathit{shrink}_s(i)$		$g_s -:= \{i\}$	$\mathit{shrink}_r(i)$	$\boxed{i \notin g_s,\ i \neq \mathit{last}_s}$ or $\mathit{mark} = \#$	$g_r -:= \{i\}$
$\mathit{grow}_s(i)$	$i \notin \mathit{used}$, $\boxed{i \in g_r \text{ or } \mathit{rec}_r}$	$g_s +:= \{i\}$	$\mathit{grow}_r(i)$	$i \notin \mathit{used}_r$	$g_r +:= \{i\}$, $\mathit{used} +:= \{i\}$
$\mathit{grow\text{-}used}_s(i)$	$i \notin \mathit{used} \cup g_s$, $\boxed{i \in \mathit{used}_r \text{ or } \mathit{rec}_r}$	$\mathit{used} +:= \{i\}$	$\mathit{cleanup}$	$\mathit{last}_r \neq \mathit{last}_s$	$\mathit{last}_r := \mathit{nil}$
			unmark	$\boxed{g_s \subseteq g_r,\ \mathit{last}_s \in g_r \cup \{\mathit{last}_r, \mathit{nil}\}}$	$\mathit{mark} := +$

Table 10.9. G with honest atomic actions