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 first alternative. But this is rare, and the latter choice is usually the lesser of

This chapter appeared in Distributed Systems, ed. S. Mullender, Addison-Wesley, 1993, pp 251-281. It is the result of joint work with Nancy Lynch and Jørgen Sagaard-Andersen.
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 mes-
sages 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 over-
come the unreliability of the channel. The receiver keeps a stock of good identifi-
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 identifier.

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. Dif-
ferent 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 ‘con-
nections’. The idea is that a connection is ‘established’, any amount of informa-
tion 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 in-
creasing 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 prac-
tice, 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 los-
ing 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 ground-
work, 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. 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.
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 $I$s can be compared for equality; $C$ assumes a total ordering. If $x$ is a multiset whose elements have a first $I$ component, we write $\text{ids}(x)$ for the multiset of $I$s that appear first in the elements of $x$.

We write $\langle \cdots \rangle$ 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, $\text{get}(m)$ 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 $\text{put}(m)$ and $\text{get}(m)$ actions. In addition, the $status$ variable records whether the most recently sent message has been delivered. The sender can use $\text{getAck}(a)$ 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 $\text{get}$ 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 $\text{getAck}(a)$ 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 $\text{getAck}$ cannot occur;
- the message was delivered by $\text{get}(m)$ and $\text{getAck}(OK)$ occurs.

If there have been crashes, there are two additional possibilities:
• 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 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 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 and before the following recover, 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.
The simplest spec which covers both these cases can lose a message at any time between a crash and its following recovery, 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.
To define D we introduce the type \textit{Mark} which has values in the set \{+\, \#, \#\}.
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.
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 transitions of G

Figure 10.4 shows the state of G, the most important actions, and the S-shaped flow of information. The *new* variables in the figure are the complement of the *used* variables in the code. The heavy lines show the flow of a new identifier from the receiver to the sender, back to the receiver along with the message, and then back again to the sender along with the acknowledgement.

G also satisfies the requirements of S, but not quite in the same way as SB.

1. At-most-once delivery is the same as in SB.
2. The sender may send a message after a crash without checking that a previous outstanding message has actually been received. Thus more than one message can be in transit at a time, so there must be a total ordering on the identifiers in transit to maintain FIFO ordering of the messages. In G this ordering is defined by the order in which the sender chooses identifiers.
3. Complete delivery is the same as in SB as long as there is no receiver crash. When the receiver crashes, it may cease to hold, with the effect that messages that the sender handles during the receiver crash may be assigned identifiers that are not in \( g_r \) and hence may be lost. The protocol ensures that this can’t happen to messages whose *put* happens after the receiver has recovered. When the sender crashes, it stops retransmitting the current message, which may be lost as a result.
4. As in SB, the sender keeps retransmitting until it gets an ack, but since messages can be lost, there must be negative as well as positive acks. When the receiver sees a message with an identifier that is not in \( g_r \) and not equal to \( last_r \) it optionally returns a negative ack. There is no point in doing this for a
message with $i < \text{last}_r$ because the sender only cares about the ack for $\text{last}_s$ and the protocol maintains the invariant $\text{last}_r \leq \text{last}_s$. If $i > \text{last}_r$, 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 $\text{grow}$, $\text{shrink}$, and $\text{cleanup}$ 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 $\text{idle}$, $\text{needI}$, $\text{send}$; for the receiver, they are $\text{idle}$ and $\text{ack}$. 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 $\text{msg}$ and $\text{last}$ variables, as follows:

- $\text{mode}_s = \text{idle}$ iff $\text{msg} = \text{nil}$
- $\text{mode}_s = \text{needI}$ iff $\text{msg} \neq \text{nil}$ and $\text{last}_s = \text{nil}$
- $\text{mode}_s = \text{send}$ iff $\text{msg} \neq \text{nil}$ and $\text{last}_s \neq \text{nil}$
- $\text{mode}_r = \text{idle}$ iff $\text{last}_r = \text{nil}$
- $\text{mode}_r = \text{ack}$ iff $\text{last}_r \neq \text{nil}$

To define $G$ we introduce the types:

- $I$, an infinite set of identifiers.
- $P$ (packet), a pair $(I, M$ or $A)$. 

Figure 10.4. State, main actions, and information flow of $G$
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 \(used\).

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 guards of all the sender actions except \(recover_s\), and likewise for \(\neg rec_r\) and the receiver actions.
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 grow_s(i), grow_s(i), 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 \textit{used}_s differently. To reduce the number of stable updates to \textit{used}_r, refine G to divide \textit{used}_r into the union of a stable \textit{used}_r-s and a volatile \textit{used}_r-v. Move a set of \textit{Is} from \textit{used}_r-s to \textit{used}_r-v with a single stable update. The \textit{used}_r-v becomes empty in \textit{recover}; simulate this with \textit{grow}_r(i) followed immediately by \textit{shrink}_r(i) for every \textit{i} in \textit{used}_r-v.

- The only state required for an idle agent is the stable variable \textit{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 \textit{getAck}, the receiver in \textit{cleanup}, and both in \textit{recover}. The \textit{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 \textit{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 \textit{recover}, the \textit{mark} variable is \# until the next \textit{put}. With \textit{unmark}, an implementation of G is free to set \textit{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

\begin{align*}
\text{cur-q} &= \{(\text{msg, mark})\mid \text{msg \neq nil and (last}_s = \text{nil or last}_s \in \text{gr})\} \quad \text{otherwise} \\
\text{inflight}_{sr} &= \{(i, m) \in \text{id}(sr) \mid i \in \text{gr} \text{ and } i \neq \text{last}_s\}, \\
\text{sorted by } i \text{ to make a sequence} \\
\text{old-q} &= \text{the sequence of } (M, \text{Mark})'s \text{ gotten by turning} \\
\text{each } (i, m) \text{ in } \text{inflight}_{sr} \text{ into } (m, \#) \\
\text{inflight}_{rs} &= \{\text{last}_s\} \quad \text{if } (\text{last}_s, \text{OK}) \in \text{rs and last}_s \neq \text{last}_r \\
\{\} &\quad \text{otherwise.}
\end{align*}

Note that the \textit{inflights} 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

\[ q \rightarrow \text{old-}q + \text{cur-}q \]

\[ \text{status} (?, \text{mark}) \quad \text{if } \text{cur-}q \neq \{ \} \]  
\[ \text{status} (\text{OK}, +) \quad \text{if } \text{mode} = \text{send} \text{ and } \text{last}_s = \text{last}_r \]  
\[ \text{status} (\text{OK}, \#) \quad \text{if } \text{mode} = \text{send} \text{ and } \text{last}_s \in \text{inflight}_r \]  
\[ \text{status} (\text{lost}, +) \quad \text{if } \text{mode} = \text{send} \text{ and } \text{last}_s \neq \{ \} \]  
\[ \text{status} (\text{lost}, +) \quad \text{if } \text{mode} = \text{idle} \]  

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

We see that G simulates the \(q\) of D using \text{old-}q + \text{cur-}q, and that \text{old-}q is the leftover messages in the channel that are still good but haven’t been delivered, while \text{cur-}q is the message the sender is currently working on, as long as its identifier is not yet assigned or still good. Similarly, \text{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 \((\text{rec}_s/r\text{ and } \text{mark})\) affect them.

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

\[ \text{If } \text{msg} = \text{nil} \text{ then } \text{last}_s = \text{nil} \]  
\[ \text{gs} \cap \text{used}_s = \{ \} \]  
\[ \text{used}_r \supseteq \text{gr} \]  
\[ \text{If } \neg \text{rec}_r \text{ then } \text{used}_r \supseteq \text{gs} \cup \text{used}_s \]  
\[ \text{used}_s \supseteq \{ \text{last}_s, \text{last}_r \} - \{ \text{nil} \} \cup \text{ids}(sr) \cup \text{ids}(rs) \]  

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, \text{gs} and \text{last}_r 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.

\[ \{ m \mid (i = \text{last}_s \text{ and } m = \text{msg}) \text{ or } (i, m) \in \text{sr} \} \text{ has 0 or 1 elements} \]  
\[ \text{If } \text{mark} = + \text{ and } \neg \text{rec}_s \text{ and } \neg \text{rec}_r \text{ then } \text{gs} \subseteq \text{gr} \text{ and } \text{last}_s \in \text{gr} \cup \{ \text{last}_r, \text{nil} \} \]  
\[ \text{gr} \cap (\{ \text{last}_r \} \cup \text{ids}(rs)) = \{ \} \]
Finally, some facts about the identifier last 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 gr, but they are all in usedr (G8b). If it’s getting a negative ack then it won’t get a later positive one (G8c).

If last ≠ nil then
\[ \text{ids}(sr) \leq \text{last} \]  \hspace{1cm} (G8a)

and if last = last or (last, OK) ∈ rs then \( \{\text{last} \} \cup \text{ids}(sr) \cap g_r = \{ \} \) \hspace{1cm} (G8b)

and if (last, lost) ∈ ids then last ≠ last  \hspace{1cm} (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: 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 l in old-q, or i = last 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 ≤ 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) if 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 ∈ gr of get is false by (G8b). In case (a) either i = last 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 unmarking status if necessary; then D’s get(m) does the rest.

— getAck(a): The q is unchanged because last = i ∈ ids, so last ∈ gr 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 ∈ g, but we know last ∈ ids, which excludes last ∈ g by (G7).

(b) In this case last = last, so (G8c) ensures a ≠ lost, so a = OK.

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

(d) Since last ∈ inflight, only (last, lost) ∈ rs is possible, so a = lost.

(e) Impossible because last ≠ nil.

— shrink: If rec, 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 $last_s \in gr$ 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 $gs$ if the sender has already used $time_s$ then $gs$ is empty. The receiver accepts any message with an identifier in the range $(time_r - 2\varepsilon - \delta, time_r + 2\varepsilon)$, where $\varepsilon$ is the maximum clock skew from real time and $\delta$ 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 Is through repeated $grow$ or $shrink$ actions as long as the invariants $gs \subseteq gr$ and $last_s \in gr \cup \{last\}$ 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 $rcv_{ps}(i, s)$ in H may simulate several $shrink_s$ actions.
Reliable Messages and Connection Establishment

Abstraction functions to G

<table>
<thead>
<tr>
<th></th>
<th>G</th>
<th>C</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>usedₙ</td>
<td>{i</td>
<td>0 ≤ i &lt; \text{timeₙ}} \cup {\text{sent}} - {\text{nil}}</td>
<td>\text{used}_s (\text{history}) used_r</td>
</tr>
<tr>
<td>used_r</td>
<td>{i</td>
<td>0 ≤ i &lt; \text{low}}</td>
<td></td>
</tr>
<tr>
<td>gs</td>
<td>{\text{timeₙ}} - {\text{sent}}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gr</td>
<td>{i</td>
<td>\text{low} &lt; i \text{ and } i &lt; \text{high}}</td>
<td>{i_r \mid {\text{nil}} }</td>
</tr>
<tr>
<td>mark</td>
<td># if lastₛ ∈ gᵣ and deadline = \text{nil} + otherwise</td>
<td># if modeₛ = \text{needI} and gₛ \subsetneq gᵣ + otherwise</td>
<td></td>
</tr>
</tbody>
</table>

msg, lastₛᵣ, and rcvₛᵣ 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

<table>
<thead>
<tr>
<th></th>
<th>G</th>
<th>C</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>growₛ(i)</td>
<td>tick(i)</td>
<td></td>
<td>sendᵣₛ(jₛ, i)</td>
</tr>
<tr>
<td>shrinkₛ(i)</td>
<td>tick(i'), i ∈ {\text{timeₛ}} - {\text{sent}}</td>
<td></td>
<td>loseᵣₛ(jₛ, i) if the last copy is lost or rcvᵣₛ(jₛ, i'), for each i ∈ gₛ - {i'}</td>
</tr>
<tr>
<td>growᵣ(i)</td>
<td>increase-high(i'), for each i ∈ {i</td>
<td>\text{high} &lt; i &lt; i'}</td>
<td></td>
</tr>
<tr>
<td>shrinkᵣ(i)</td>
<td>increase-low(i'), for each i ∈ {i</td>
<td>\text{low} &lt; i ≤ i'}</td>
<td></td>
</tr>
<tr>
<td>cleanup</td>
<td>cleanup</td>
<td></td>
<td>rcvᵣₛ(lastᵣᵣ, done)</td>
</tr>
</tbody>
</table>

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 \text{time} for the next identifier. The receiver keeps track of \text{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 \text{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 \text{low} := \text{high}; this ensures that no messages are accepted twice.
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 idlesenders. 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.

Figure 10.6. The flow of information in C
### Reliable Messages and Connection Establishment

<table>
<thead>
<tr>
<th>Name</th>
<th>Guard</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>put(m)</strong></td>
<td>msg = nil</td>
<td>msg := m</td>
</tr>
<tr>
<td>choose(i)</td>
<td>msg ≠ nil,</td>
<td>sent := i, last_s := i,</td>
</tr>
<tr>
<td></td>
<td>last_s = nil, i = time_s, i = sent</td>
<td></td>
</tr>
<tr>
<td>send</td>
<td>last_s ≠ nil</td>
<td>send_s(last_s, msg)</td>
</tr>
<tr>
<td><strong>get(m)</strong></td>
<td>exists i such that recv(m, i), deadline := now + (\delta)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i ∈ (low..high)</td>
<td>send_s(i, OK)</td>
</tr>
<tr>
<td><strong>getAck(a)</strong></td>
<td>recv_s(last_s, a) last_s := nil, msg := nil</td>
<td></td>
</tr>
<tr>
<td>sendAck</td>
<td>exists i such that recv(m, i), send_s(i, if i = last_r then OK else lost)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i ∉ (low..high)</td>
<td>if i = last_s then deadline := nil</td>
</tr>
<tr>
<td><strong>crash_s</strong></td>
<td>recv_s := true, deadline := nil</td>
<td></td>
</tr>
<tr>
<td><em>recover_s</em></td>
<td>recv_s := false, last_s := nil, msg := nil</td>
<td></td>
</tr>
<tr>
<td>cleanup</td>
<td>sent ≠ time_s</td>
<td>sent := nil</td>
</tr>
<tr>
<td>tick(i)</td>
<td>time_s &lt; i,</td>
<td>time_s := i, now - i &lt; (\epsilon)</td>
</tr>
<tr>
<td></td>
<td>i &lt; deadline or deadline = nil</td>
<td></td>
</tr>
<tr>
<td>progress(i)</td>
<td>now &lt; i,</td>
<td>i - time_s/r &lt; (\epsilon), now := i</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Guard</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>time_s</td>
<td>:= 0 (stable)</td>
<td></td>
</tr>
<tr>
<td>sent</td>
<td>:= I or nil := nil</td>
<td></td>
</tr>
<tr>
<td>last_s</td>
<td>:= I or nil := nil</td>
<td></td>
</tr>
<tr>
<td>msg</td>
<td>:= M or nil := nil</td>
<td></td>
</tr>
<tr>
<td>rec_s</td>
<td>:= false</td>
<td></td>
</tr>
<tr>
<td>rec_r</td>
<td>:= false</td>
<td></td>
</tr>
<tr>
<td>deadline</td>
<td>:= I or nil := nil</td>
<td></td>
</tr>
<tr>
<td>now</td>
<td>:= I := 0</td>
<td></td>
</tr>
</tbody>
</table>

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 ¬ rec has been omitted from the guards of all the sender actions except recover_s, and likewise for ¬ 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 can 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 + 2ε to increase high
- ε = maximum of \( |now - time_{r/s}| \)

These parameters must satisfy two constraints:

- δ > ε so that mode_s = send implies last_s < deadline.
- β > 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 4ε + β, because high can be as big as time_r + 2ε + β at the time of the crash because of (e), and time_r - 2ε 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 ≠ 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).

\[
\begin{align*}
\text{last}_s & \leq \text{time}_s \\
\text{last}_r & \leq \text{low} \leq \text{high} \\
\text{ids}(rs) & \leq \text{low} \\
\text{If } \neg \text{rec}_r \text{ then } & \text{time}_r + 2\varepsilon \leq \text{high} \\
\text{ids}(sr) & \leq \text{last}_s \\
\text{last}_r & \leq \text{last}_s
\end{align*}
\]
Reliable Messages and Connection Establishment

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

\text{low} \leq \text{time}_s \tag{C4}

\text{low} < \text{time}_s \text{ if } last_s \neq \text{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 \text{deadline} \neq \text{nil} then
\[ \text{now} < last_s + \varepsilon + \delta \tag{C5a} \]
\[ \text{low} < last_s \tag{C5b} \]

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 \text{last}_s, \text{OK} \in rs \text{ then } last_s \leq low \tag{C6}

If \text{last}_s, \text{lost} \in rs \text{ then } last_r < last_s \tag{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:

\begin{align*}
\text{mode}_s &= \text{idle} \quad \text{iff} \quad j_s = \text{last}_s = \text{nil} \\
\text{mode}_s &= \text{needI} \quad \text{iff} \quad j_s \neq \text{nil} \\
\text{mode}_s &= \text{send} \quad \text{iff} \quad \text{last}_s \neq \text{nil} \\
\text{mode}_r &= \text{idle} \quad \text{iff} \quad j_r = \text{last}_r = \text{nil} \\
\text{mode}_r &= \text{accept} \quad \text{iff} \quad j_r \neq \text{nil} \\
\text{mode}_r &= \text{ack} \quad \text{iff} \quad last_r \neq \text{nil}
\end{align*}

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 \(i_I\)) that both know has not been used.
- One round-trip (two packets) is then needed to send and ack the message.
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 \(i, \text{done}\) packets are both often called 'close' packets in the literature.

### 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 \(i, \text{done}\) packets are both often called 'close' packets in the literature.

---

3 \((i, \text{lost})\) 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 \((i, \text{OK})\) is a positive acknowledgement; it means \(i\)’s message was delivered to get.
Reliable Messages and Connection Establishment

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 Is, 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 ¬ recs has been omitted from the guards of all the sender actions except recover, and likewise for ¬ recr and the receiver actions.

10.8.1 Invariants

Recall that ids(c) is {i | (i, *) ∈ c}. We also define jds(c) = {j | (j, *) ∈ c or (*, j) ∈ c}.

Most of H’s invariants are boring facts about the progress of I’s and J’s from used sets through i/j to last/s/r. We need the history variables useds and seen to express some of them. (H6) says that there’s at most one J (from a needI packet)
<table>
<thead>
<tr>
<th>Name</th>
<th>Guard</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(m)</td>
<td>msg := nil, j := j, send(needl, j)</td>
<td>assign(I, i), rcv*(needl, j), j := j, i := i, lastr := nil, usedr := i, seen := i</td>
</tr>
<tr>
<td>request</td>
<td>jk := nil, last := nil, send*(needl, jk)</td>
<td>sendI, jk := jk, last := nil, used := jk</td>
</tr>
<tr>
<td>choose</td>
<td>last := nil, recv(last, i)</td>
<td>send, last := nil, last := i, used := i, recv := last</td>
</tr>
<tr>
<td></td>
<td>msg := last := nil, send*(last, msg)</td>
<td>send, last := nil, last := msg, used := last, recv := msg</td>
</tr>
<tr>
<td></td>
<td>bounce := rcv*(last, a)</td>
<td>bounce, last := nil, last := a, used := last, recv := a</td>
</tr>
<tr>
<td></td>
<td>(j, i) recv := i, done</td>
<td>cleanup, j := i, i := nil, last := nil</td>
</tr>
<tr>
<td></td>
<td>or rcv := ok, OK</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>crash</strong></td>
<td>rec := true</td>
</tr>
<tr>
<td></td>
<td>*recovery</td>
<td>rec := true</td>
</tr>
<tr>
<td></td>
<td>grow-*</td>
<td>rec := false</td>
</tr>
<tr>
<td></td>
<td><strong>crash</strong></td>
<td>rec := true</td>
</tr>
<tr>
<td></td>
<td>*recovery</td>
<td>rec := true</td>
</tr>
<tr>
<td></td>
<td>grow-*</td>
<td>rec := false</td>
</tr>
<tr>
<td></td>
<td>j-used := {i}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>used := sequence[I] := () (history)</td>
<td>used := set[I] := { } (stable)</td>
</tr>
<tr>
<td></td>
<td>j := I or nil := nil</td>
<td>j := I or nil := nil</td>
</tr>
<tr>
<td></td>
<td>msg := M or nil := nil</td>
<td>i := I or nil := nil</td>
</tr>
<tr>
<td></td>
<td>last := I or nil := nil</td>
<td>last := I or nil := nil</td>
</tr>
<tr>
<td></td>
<td>rec := Boolean := false</td>
<td>rec := Boolean := false</td>
</tr>
</tbody>
</table>

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

that gets assigned a given I. (H8) says that as long as the sender is still in mode needl, nothing involving i, has made it into the channels.

\[ j \text{-used} \supseteq \{ j_s, j \} \cap [\text{nil}] \cup \text{ids(sr}) \cup \text{ids(rs)} \]  
\[ \text{used}_r \supseteq \{ i_r, \text{last}_r \} \cap [\text{nil}] \cup \text{used}_s \cup \{ i | (*, i) \in rs \} \cup \text{ids(sr}) \cup \text{ids(rs)} \]  
\[ \text{used}_s \supseteq \{ \text{last}_s, \text{last}_r \} \cap [\text{nil}] \cup \text{ids(sr}) \cup \text{ids(rs)} \]  
\[ \text{If} (i, \text{done}) \in \text{sr} \text{ then } i \neq \text{last}_s \]  
\[ \text{If } i, \neq \text{nil } \text{then } (j_r, i_r) \notin \text{seen} \]  
\[ \text{If } (j, i) \in \text{seen and } (j', i) \in \text{seen then } j = j' \]  

(H1)  
(H2)  
(H3)  
(H4)  
(H5)  
(H6)
If \((j, i) \in rs\) then \((j, i) \in seen\) \hspace{1cm} (H7)

If \(j_s = j_r \neq nil\) then \((i_r, \ast) \notin sr\) and \((i_r, done) \notin rs\) \hspace{1cm} (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 \((i_r/last_r, done)\) when 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.
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 \( \text{recycle}(i) \) that removes an identifier from \( \text{used}_s \) and \( \text{used}_r \) 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: \( \text{recycle} \) simulates no change in the spec. The former is also simple: we put a strong enough guard on \( \text{recycle} \) to ensure that all the invariants still hold. To find out what this guard is we need only find all the invariants that mention \( \text{used}_s \) or \( \text{used}_r \), since those are the only variables that \( \text{recycle} \) 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 \( \text{seen} \) up to date, and a similar \( \text{recycle-j} \) action. Table 10.8 gives the \( \text{recycle} \) actions for \( G \) and \( H \).

10.9 Finite Identifiers

How can we implement the guards on the \( \text{recycle} \) 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Guard</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>recycle(i)</td>
<td>( i \notin g_s \cup g_r \cup {\text{last}_s, \text{last}_r} )</td>
<td>( \text{used}_s := {i} ), ( \text{used}_r := {i} )</td>
</tr>
<tr>
<td>recycle(j)</td>
<td>( j \notin {\text{last}_s, i_r, \text{last}_r} \cup {i \mid (*, i) \in rs} \cup \text{ids}(sr) \cup \text{ids}(rs) )</td>
<td>( \text{used}_j := {j} ), ( \text{seen} := {i \mid (j, i) \in \text{seen} \mid (j, i)} )</td>
</tr>
</tbody>
</table>

Table 10.8. Actions to recycle identifiers
Reliable Messages and Connection Establishment

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


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:

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

The abstraction function to D is:

$$
\begin{align*}
q & \begin{cases} 
+ & \text{the elements of } buf_r \text{ paired with } + \\
+ \text{ old-q + cur-q} & \text{ } \\
+ \text{ the elements of } buf_s \text{ paired with } + 
\end{cases} \\
status & \begin{cases} 
(?, +) & \text{if } buf_s \neq \text{ empty} \\
(?, \text{mark}) & \text{if } cur-q \neq \{ \} \\
(?, +) & \text{if } mode_s = \text{send}, last_s = last_r, buf_r \neq \{ \} \\
(\text{OK}, +) & \text{if } mode_s = \text{send}, last_s = last_r, buf_r = \{ \} \\
(\text{OK}, #) & \text{if } mode_s = \text{send and } last_s \in \text{inflight}_{rs} \\
(\text{lost}, +) & \text{if } mode_s = \text{send} \\
& \quad \text{and } last_s \not\in \{ gr \cup \{last_s\} \cup \text{inflight}_{rs} \} \\
(\text{ack}, +) & \text{if } mode_s = \text{idle} \\
\end{cases} \\
\end{align*}
$$

rec_s/r = rec_s/r
<table>
<thead>
<tr>
<th>Name</th>
<th>Guard</th>
<th>Effect</th>
<th>Name</th>
<th>Guard</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>put(m)</strong></td>
<td></td>
<td>append m to buf&lt;sub&gt;s&lt;/sub&gt;</td>
<td><strong>Name</strong></td>
<td><strong>Guard</strong></td>
<td><strong>Effect</strong></td>
</tr>
<tr>
<td>prepare(m)</td>
<td>msg = nil, m first on buf&lt;sub&gt;f&lt;/sub&gt;, g&lt;sub&gt;s&lt;/sub&gt; ≤ g&lt;sub&gt;r&lt;/sub&gt; or rec&lt;sub&gt;r&lt;/sub&gt;</td>
<td>buf&lt;sub&gt;f&lt;/sub&gt; := tail(buf&lt;sub&gt;f&lt;/sub&gt;), msg := m, mark := +</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>choose(i)</td>
<td>msg ≠ nil, i ∈ g&lt;sub&gt;s&lt;/sub&gt;, last&lt;sub&gt;s&lt;/sub&gt; = nil, last&lt;sub&gt;r&lt;/sub&gt; := i, used&lt;sub&gt;s&lt;/sub&gt; := i</td>
<td>g&lt;sub&gt;s&lt;/sub&gt; := {j</td>
<td>j ≤ i}, last&lt;sub&gt;s&lt;/sub&gt; := i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>send&lt;sub&gt;s&lt;/sub&gt;(i, m)</td>
<td>i = last&lt;sub&gt;s&lt;/sub&gt; ≠ nil, m = msg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rcv&lt;sub&gt;s&lt;/sub&gt;(i, m)</td>
<td>if i ∈ g&lt;sub&gt;r&lt;/sub&gt; then append m to buf&lt;sub&gt;r&lt;/sub&gt;, sendAck := false, g&lt;sub&gt;r&lt;/sub&gt; := {j</td>
<td>j ≤ i}, last&lt;sub&gt;r&lt;/sub&gt; := i, else if i ∉ g&lt;sub&gt;r&lt;/sub&gt; ∪ {last&lt;sub&gt;r&lt;/sub&gt;} then optionally nack-buf := ⟨i⟩ else if i = last&lt;sub&gt;r&lt;/sub&gt; then sendAck := true</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>send&lt;sub&gt;r&lt;/sub&gt;(i, a)</td>
<td>if i = last&lt;sub&gt;r&lt;/sub&gt; then sendAck := true, buf&lt;sub&gt;r&lt;/sub&gt; := tail(buf&lt;sub&gt;r&lt;/sub&gt;)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rcv&lt;sub&gt;r&lt;/sub&gt;(i, a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*get(m)</td>
<td>m first on buf&lt;sub&gt;f&lt;/sub&gt;, i = last&lt;sub&gt;r&lt;/sub&gt;, sendAck := false, sendAck := true, buf&lt;sub&gt;r&lt;/sub&gt; := tail(buf&lt;sub&gt;r&lt;/sub&gt;)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*get&lt;sub&gt;s&lt;/sub&gt;(a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>crash&lt;sub&gt;s&lt;/sub&gt;</strong></td>
<td>rec&lt;sub&gt;s&lt;/sub&gt; := true</td>
<td><strong>crash&lt;sub&gt;r&lt;/sub&gt;</strong></td>
<td>rec&lt;sub&gt;r&lt;/sub&gt; := true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*recover&lt;sub&gt;s&lt;/sub&gt;</td>
<td>rec&lt;sub&gt;s&lt;/sub&gt;</td>
<td>*recover&lt;sub&gt;r&lt;/sub&gt;</td>
<td>last&lt;sub&gt;r&lt;/sub&gt; := nil,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>last&lt;sub&gt;s&lt;/sub&gt; := nil,</td>
<td>used&lt;sub&gt;r&lt;/sub&gt; ⊇ g&lt;sub&gt;s&lt;/sub&gt; ∪ used&lt;sub&gt;s&lt;/sub&gt;,</td>
<td>mark := #, buf&lt;sub&gt;r&lt;/sub&gt; := ⟨⟩,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>msg := nil, buf&lt;sub&gt;s&lt;/sub&gt; := ⟨⟩,</td>
<td></td>
<td>nack-buf := ⟨⟩, rec&lt;sub&gt;r&lt;/sub&gt; := false</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ack := lost, rec&lt;sub&gt;s&lt;/sub&gt; := false</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shrink&lt;sub&gt;s&lt;/sub&gt;(i)</td>
<td>g&lt;sub&gt;s&lt;/sub&gt; := [i]</td>
<td>shrink&lt;sub&gt;r&lt;/sub&gt;(i)</td>
<td>i ∉ g&lt;sub&gt;s&lt;/sub&gt;, i ≠ last&lt;sub&gt;s&lt;/sub&gt;</td>
<td>g&lt;sub&gt;r&lt;/sub&gt; := [i]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or mark = #</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>grow&lt;sub&gt;s&lt;/sub&gt;(i)</td>
<td>i ∉ used&lt;sub&gt;s&lt;/sub&gt;,</td>
<td>g&lt;sub&gt;s&lt;/sub&gt; := [i]</td>
<td>grow&lt;sub&gt;r&lt;/sub&gt;(i)</td>
<td>i ∉ used&lt;sub&gt;r&lt;/sub&gt;,</td>
<td>g&lt;sub&gt;r&lt;/sub&gt; := [i],</td>
</tr>
<tr>
<td></td>
<td>g&lt;sub&gt;s&lt;/sub&gt; := [i]</td>
<td></td>
<td></td>
<td></td>
<td>used := [i]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>grow-</td>
<td>i ∉ used&lt;sub&gt;s&lt;/sub&gt; ∪ g&lt;sub&gt;s&lt;/sub&gt;,</td>
<td>used := [i]</td>
<td>cleanup</td>
<td>last&lt;sub&gt;r&lt;/sub&gt; ≠ last&lt;sub&gt;s&lt;/sub&gt;,</td>
<td>last&lt;sub&gt;r&lt;/sub&gt; := nil</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>used&lt;sub&gt;s&lt;/sub&gt;(i)</td>
<td>i ∈ used&lt;sub&gt;s&lt;/sub&gt; ∪ rec&lt;sub&gt;r&lt;/sub&gt;</td>
<td></td>
<td>unmark</td>
<td>g&lt;sub&gt;s&lt;/sub&gt; ⊆ g&lt;sub&gt;r&lt;/sub&gt;, last&lt;sub&gt;s&lt;/sub&gt; ∈ g&lt;sub&gt;r&lt;/sub&gt; ∪ {last&lt;sub&gt;r&lt;/sub&gt;, nil}</td>
<td>mark := +</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 10.9. G with honest atomic actions