# Understanding Network Connections

Butler Lampson

There are lots of protocols for establishing connections (or equivalently, doing at-most-once message delivery) across a network that can delay, reorder, duplicate and lose packets. Most of the popular ones are based on three-way handshake, but some use clocks or extra stable storage operations to reduce the number of messages required. It's hard to understand why the protocols work, and there are almost no correctness proofs; even careful specifications are rare.

I will give a specification for at-most-once message delivery, an informal account of the main problems an implementation must solve and the common features that most implementations share, and outlines of proofs for three implementations. The specifications and proofs based on Lamport's methods for using abstraction functions to understand concurrent systems, and I will say something about how his methods can be applied to many other problems of practical interest.

# Understanding Network Connections

Butler Lampson

October 30, 1995

This is joint work with Nancy Lynch.

The errors in this talk are mine, however.

# Overview

Specify at-most-once message delivery.

Describe other features we want from an implementation

Give a framework for thinking about implementations.

Show how to prove correctness of an implementation.

# The Problem

## Network Connections
## or
## Reliable At-Most-Once Messages

Messages are delivered in FIFO order.

A message is not delivered more than once.

A message is acked only if delivered.

A message or ack is lost only if it is being sent between crash and recovery.

# Pragmatics

"Everything should be made as simple as possible, but no simpler."
*A. Einstein*

Make progress: regardless of crashes,
if both ends stay up a waiting message is sent, and
otherwise both parties become idle.

Idle at no cost: an idle agent
has no state that changes for each message, and
doesn't send any packets.

Minimize stable storage operations — <<1 per message.

Use channels that are easy to implement:
They may lose, duplicate, or reorder messages.

# Pragmatics

Some pragmatic issues we won't discuss:

- Retransmission policy.
- Detecting failure of an attempt to send or ack, by timing it out.

# Describing a System

A system is defined by a safety and a liveness property:

*Safety*: nothing bad ever happens. Defined by a state machine:
A set of *states*. A state is a pair (external state, internal state)
A set of *initial* states.
A set of *transitions* from one state to another.

*Liveness*: something good eventually happens.

An *action* is a named set of transitions; actions partition the transitions.
For instance: *put*(*m*); *get*(*m*); $crash_s$

A *history* is a possible sequence of actions, starting from an initial state.
The *behavior* of the system is the set of possible histories.

An *external* action is one in which the external state changes.
Correspondingly there are external histories and behaviors.

# Defining Actions

An action is:

A *name*, possibly with parameters: *put*(*“red”*).

A *guard*, a predicate on the state which must be true for this action to be a possible transition: $q$ < > and $i > 3$.

An *effect*, changes in some of the state variables: $i := i + 1$.

The entire action is atomic.

Example:

| | | |
|---|---|---|
| *get*(*m*): | *m* first on *q* | take first from *q*, if *q* now empty and *status* = ? then *status* := *true* |
| **name** | **guard** | **effect** |

# Specifying At-Most-Once Messages

| | | | |
|---|---|---|---|
| State: | $q$ | : sequence[$M$] | := < > |
| | $status$ | : {$true$, $false$, ?} | := $true$ |
| | $rec_{s/r}$ | : Boolean | := $false$ |

| "Sender Actions" | | | "Receiver Actions" | | |
|---|---|---|---|---|---|
| **Name** | **Guard** | **Effect** | **Name** | **Guard** | **Effect** |
| $put(m)$** | | if $\sim rec_s$ then append $m$ to $q$, $status$ := ? | $get(m)$* | $\sim rec_r$, $m$ first on $q$ | take first from $q$, if $q$ now empty and $status$ = ? the $status$ := $true$ |
| $getAck(b)$* | $\sim rec_s$ , $status = b$ | none | | | |
| $crash_s$** | | $rec_s$ := $true$ | $crash_r$** | | $rec_r$ := $true$ |
| $recover_s$* | $rec_s$ | $rec_s$ := $false$ | $recover_r$* | $rec_r$ | $rec_r$ := $false$ |
| $lose$ | $rec_s$ or $rec_r$ | delete any element of $q$; if it's the last then $status$ := $false$ or $status$ := $false$ | | | |

# Histories for AMO Messages

| **Action** | *q* | *status* | **Action** | *q* | *status* | **Action** | *q* | *status* |
|---|---|---|---|---|---|---|---|---|
| **Initially** | < > | *true* | | | | | | |
| *put*(“red”) | <“red”> | ? | | | | | | |
| *get*(“red”) | < > | *true* | | | | | | |
| *getAck*(*true*) | < > | *true* | | | | | | |
| *put*(“green”) | <“green”> | ? | | | | | | |
| $\mathit{crash}_r$, *lose* | <“green”> | ? | $\mathit{crash}_r$, *lose* | <“green”> | *false* | $\mathit{crash}_r$, *lose* | < > | *false* |
| | | ? | *getAck*(*false*) | <“green”> | *false* | *getAck*(*false*) | < > | *false* |
| *put*(“blue”) | <“green”, “blue”> | ? | *put*(“blue”) | <“green”, “blue”> | ? | *put*(“blue”) | <“blue”> | ? |
| *get*(“green”) | <“blue”> | ? | *get*(“green”) | <“blue”> | ? | | | |
| *get*(“blue”) | < > | *true* | | | | | | |
| *getAck*(*true*) | < > | *true* | | | | | | |

# Channels

State $sr$ : multiset[$P$] := {} $P = I \times M$ or $I \times \text{Bool}$ $rs$ : multiset[$P$] := {}

| Name | Guard | Effect | Name | Guard | Effect |
|---|---|---|---|---|---|
| $snd_{sr}(p)$ | | $sr := sr \cup \{p\}$ | $snd_{rs}(p)$ | | $rs := rs \cup \{p\}$ |
| $rcv_{sr}(p)$ | $p \in sr$ | $sr := sr - \{p\}$ | $rcv_{rs}(p)$ | $p \in rs$ | $rs := rs - \{p\}$ |
| $lose_{sr}$ | $\exists\, p \mid p \in sr$ | $sr := sr - \{p\}$ | $lose_{rs}$ | $\exists\, p \mid p \in rs$ | $rs := rs - \{p\}$ |

# Stable Implementation

| State: | | | | | |
|---|---|---|---|---|---|
| $mode_S$ | : {*acked, send, rec*} | := *acked* | $mode_r$ | : {*idle, rcvd, ack*} | := *idle* |
| $good_S$ | : set[ *I* ] | := *I* | $good_r$ | : set[ *I* ] | := *I* |
| $last_s$ | : *I* | | $last_r$ | : *I* | |
| *cur* | : *M* | | *buf* | : sequence[*M*] | := <> |

| Name | Guard | Effect |
|---|---|---|
| *put*(*m*)** | *mode* = *acked*<br>and ∃ *i* \| *i* ∈ *good* | *cur* := *m*, *last* := *i*,<br>*mode* := *send*,<br>take *i* from *good* |
| $snd_{sr}$(*i*, *m*)* | *mode* = *send*,<br>*i* = *last*, *m* = *cur* | none |
| $rcv_{rs}$(*i*, –)<br>** | | if *mode* = *send* and<br>*i* = *last* then<br>*mode* := *acked* |
| *getAck*<br>(*true*)* | *mode* = *acked* | none |

| Name | Guard | Effect |
|---|---|---|
| $rcv_{sr}$(*i*, *m*)<br>** | | if *i* ∈ *good* then<br>*mode* := *rcvd*, take *i* from *good*,<br>*last* := *i*, append *m* to *buf*<br>else if *i* = *last* and *mode* = *idle*<br>then *mode* := *ack* |
| *get*(*m*)* | *mode* = *rcvd*,<br>*m* first on *buf* | take first from *buf*;<br>if it's now empty,<br>*mode* := *ack* |
| $snd_{rs}$(*i*,<br>*true*)* | *mode* = *ack*,<br>*i* = *last* | *mode* := *idle* |

# What Does “Implements” Mean?

Divide actions into *external* (marked * or **) and *internal* (unmarked).
External actions change external state, internal ones don’t.

An external history is a history (sequence of actions) with all the internal actions removed.

*T* implements *S* if

every external history of *T* is an external history of *S*, and

*T*’s liveness property implies *S*’s liveness property.

# Abstraction Functions

Suppose we have a function $f$ from $T$'s state to $S$'s state such that:

$f$ takes initial states to initial states;

$f$ maps every transition of $T$ to a sequence of transitions of $S$, perhaps empty (i.e., the identity on $S$);

$$\begin{tikzcd} f(t) \arrow[r, "S \text{ or skip}"] & f(t') \\ t \arrow[u] \arrow[r, "T"] & t' \arrow[u] \end{tikzcd}$$

$f$ maps every external action of $T$ to a sequence containing the same external action of $S$ and no other external actions.

$f$ maps every internal action of $T$ to a sequence of internal actions.

Then $T$ implements $S$.
Why bother? Transitions are simpler than histories.

# Proof of Stable Implementation

## Invariants

(1) $good_s \cap ( \{last_s\} \cup \mathrm{ids}(sr) \cup \mathrm{ids}(rs) ) = \{\}$

(2) $good_r \cap \mathrm{ids}(rs) = \{\}$

(3) $good_r \supseteq good_s$

(4) $((i, m) \in sr$ and $i \in good_r)$ implies $m = cur$

## Abstraction function

$$q = \begin{cases} \langle cur \rangle & \text{if } mode_s = send \text{ and } last_s \in good_r \\ \langle\, \rangle & \text{otherwise} \end{cases} + buf$$

$$status = \begin{cases} ? & \text{if } mode_s = send \\ true & \text{otherwise} \end{cases}$$

# Methodology for Proofs

*Simplify* the spec and the implementations.
Save clever encodings for later.

Make a "working spec" that's easier to handle:
It implements the actual spec.
It has as much non-determinism as possible.
All the prophecy is between it and the actual spec.

**actual spec** $\xleftarrow{\text{implements}}$ **working spec** $\xleftarrow{\text{implements}}$ **implementation**

Find the abstraction function. The rest is automatic.

Give names to important functions of your state variables.

To design an implementation, first invent the guards you need, then figure out how to implement them.

# History Variables

If you add a variable $h$ to the state space such that

If $s$ is an old initial state then there's an $h$ such that $(s, h)$ is initial;

If $(s, h) \rightarrow (s', h')$ then $s \rightarrow s'$;

If $s \rightarrow s'$ then for any $h$ there's an $h'$ such that $(s, h) \rightarrow (s', h')$

then the new state machine has the same histories as the old one.

# Predicting Non-Determinism

Suppose we add *mode* := *acked* to $crash_s$.

Consider the sequence *put*("red"), *snd*, $crash_s$, *put*("blue"), *snd*.

Now we have $sr$ = {(1, "red"), (2, "blue")}. We need an ordering on identifiers to order these packets and maintain FIFO delivery. On $rcv_{sr}(i, m)$ the receiver must remove all identifiers $i$ from $good_r$.

But now "red" is lost if (2, "blue") is received first. If we use the obvious abstraction function

$$q = \text{the } m\text{'s from } \{(i, m) \in sr \cup (last_s, cur) \mid i \in good_r\} \text{ sorted by } i,$$

this loss doesn't happen between $crash_s$ and $recover_s$, as allowed by the spec, but later at the *rcv*.

We give a working spec that makes this delay explicit.

# Delayed-Decision Specification

| State: | | | | |
|---|---|---|---|---|
| | $q$ | : sequence[($M$, $Flag$)] | := < > | $Flag = \{OK, ?\}$ |
| | $status$ | : ({$true$, $false$, ?}, $Flag$) | := $true$ | |
| | $rec_{s/r}$ | : Boolean | := $false$ | |

| **Name** | **Guard** | **Effect** | **Name** | **Guard** | **Effect** |
|---|---|---|---|---|---|
| $put(m)$** | | if $\sim rec_s$ then<br>add ($m$, $OK$) to $q$,<br>$status$ := (?, $OK$) | $get(m)$* | $\sim rec_r$,<br>($m$, –) first on $q$ | take first from $q$,<br>if $q$ now empty<br>and $status$ = (?, $f$)<br>$status$ := ($true$, $f$ |
| $getAck(b)$* | $\sim rec_s$ ,<br>$status$ = ($b$, –) | none | | | |
| *mark* | $rec_s$ or $rec_r$ | in some element of<br>$q$ or in $status$, set<br>$flag$ := ? | *unmark* | | in some element o<br>$q$ or in $status$, set<br>$flag$ := $OK$ |
| *drop* | *true* | delete some element of $q$ with $flag$ := ?,<br>and if it's the last, $status$ := ($false$, $OK$)<br>or if $status$ = (–, ?), $status$ := ($false$, $OK$) | | | |

# Prophecy Variables

If you add a variable $p$ to the state space such that

If $s$ is an old initial state then there's a $p$ such that $(s, p)$ is initial;

If $(s, p) \to (s', p')$ then $s \to s'$;

If $s \to s'$ then for any $h'$ there exists an $h$ such that $(s, h) \to (s', h')$

If $s$ is an old state, there's a $p$ such that $(s, p)$ is a new state.

then the new state machine has the same histories as the old one.

If $T$ implements $S$, you can always add history and prophecy variables to $T$ and then find an abstraction function to $S$.

# Prophecy for Delayed-Decision

Extend *Flag* to include a *lost* component drawn from the set {*OK*, *lost*}.

The *lost* component prophesies whether *drop* will attack or not.

The abstraction function is

$q_S$ = the first components of elements of $q_{DP}$ that are not lost

$status_S$ = the first component of $status_{DP}$ if it is not lost, else *false*.

# Delayed-Decision with Prophecy DP

State: $q$ : sequence[($M$, $Flag$)] := < >  $Flag$ = ({$OK$, ?}, {$OK$, $lost$})
$status$ : ({$true$, $false$, ?}, $Flag$) := ($true$, $OK^2$)
$rec_{s/r}$ : Boolean := $false$

| Name | Guard | Effect | Name | Guard | Effect |
|---|---|---|---|---|---|
| *put*(*m*)** | | if $\sim rec_s$ then add ($m$,$OK^2$) to $q$, $status$ := (?, $OK^2$) | *get*(*m*)* | $\sim rec_r$, ($m$, –) first on $q$ and not lost | take first from $q$, if $q$ now empty and $status$ = (?, $f$) $status$ := ($true$, $f$ |
| *getAck*(*b*)* | $\sim rec_s$ , $status$ = ($b$, –), not lost | none | | | |
| *mark* | $rec_s$ or $rec_r$, $\exists\ x \in$ {$OK$, $lost$} | in some element of $q$ or in $status$, set $flag$ := (?, $x$). If last of $q$ is lost, set $status$ flag (?, $lost$) | *unmark* | | in some element o $q$ or in $status$ that isn't lost, set $flag$ := $OK$ |
| *drop* | *true* | delete some lost element of $q$ or if $status$ is lost, $status$ := ($false$, $OK^2$) | | | |

# Generic Implementation

State:

| | | |
|---|---|---|
| $mode_s$ | : {*acked, send, rec*} | := *acked* |
| $good_s$ | : set[ $I$ ] | := {} |
| $last_s$ | : $I$ | |
| *cur* | : $M$ | |
| *used* | : set[$I$] | := {} |
| *ack* | : Boolean | := *false* |

| | | |
|---|---|---|
| $mode_r$ | : {*idle, rcvd, ack*} | := *idle* |
| $good_r$ | : set[ $I$ ] | := {} |
| $last_r$ | : $I$ | |
| *buf* | : sequence[$M$] | := <> |
| *nacks* | : sequence[$I$] | := <> |

| Name | Guard | Effect |
|---|---|---|
| *put*(*m*)** | *mode* = *acked* | *mode* := *needid*, *cur* := *m*, *good*:={} |
| *choose-id* (*i*, *m*) | *mode* = *needid*, $i \in$ *good*, *m*=*cur* | *mode* := *send*, *last* := *i*, move *i* to from *good* to *used* |
| $snd_{sr}(i, m)$* | as usual | |
| $rcv_{rs}(i, b)$ ** | | if *mode* = *send* and *i* = *last* then *ack*:=*b* *mode* := *acked* |
| *getAck*(*b*)* | *mode* = *acked*, *b* = *ack* | none |

| Name | Guard | Effect |
|---|---|---|
| $rcv_{sr}(i, m)$ ** | if $i \in$ *good* then *mode* := *rcvd*, take all *I*s   *i* from *good*, *last* := *i*, append *m* to *buf* else if *i*   *last* then add *i* to *nacks* else if *i* = *last* and *mode* = *idle* then *mode* := *ack* | |
| *get*(*m*)* | as usual | as usual |
| $snd_{rs}(i,T)$ | *mode* = *ack*, *i* = *last* | *mode* := *idle* |
| $snd_{rs}(i,F)$ | *i* first on *nacks* | take first from *nacks* |

# Generic Magic

State: $good_S$ : set[ $I$ ] := {}
$last_s$ : $I$
$used_s$ : set[$I$] := {}

$good_r$ : set[ $I$ ] := {}
$last_r$ : $I$
*issued* : set[$I$] := {}

| Name | Guard | Effect |
|---|---|---|
| *shrink*-$g_s(i)$ | *true* | take $i$ from *good* |
| *grow*-$g_s(i)$ | $i \in$ *good*,<br>$i \notin$ *used* | add $i$ to *good* |
| $recover_s$ | *rec* | *mode* := *acked*,<br>*last* := *nil*,<br>*ack* := *false* |

| Name | Guard | Effect |
|---|---|---|
| *shrink*-$g_r(i)$ | $i \notin good_s \cup \{last_s\}$ | take $i$ from *good* |
| *grow*-$g_r(i)$ | $i \notin$ *issued* | add $i$ to *good*, *issued* |
| *cleanup* | *mode* *rcvd*,<br>*last* $last_s$ | *last* := *nil* |
| $recover_r$ | *rec* | *mode* := *idle*,<br>*last* := *nil*, take some<br>$I$s from *good*,<br>clear *buf*, *nacks* |

# Generic Abstraction Function

msg($id$) = $\{m \mid$ ($id = last_s$ and $m = current$ ) or $id \in$ ids($sr$)
or ($id = last_r$ and $m$ is last on $buf$)$\}$

This defines a partial function msg: $ID \rightarrow M$.

$current\text{-}q$ = $\langle(current, OK)\rangle$ if $mode_s = send$ and $last_s \in good_r$
or $mode_s = needid$ and $good_s$ $good_r$
$\langle(current, ?)\rangle$ if $mode_s = needid$ and not $good_s$ $good_r$
$\langle\rangle$ otherwise

$inflight_{sr} = \{id \mid$ $id \in$ ids($sr$) and $id \in good_r$
and not ($id = last_s$ and $mode_s = send$) $\}$,
sorted by $id$ to make a sequence

$inflight\text{-}m$ = the sequence of $M$s gotten from msg of each $I$ in $inflight_{rs}$.

$inflight_{rs} = \{last_s\}$ if $(ack, last_s, true) \in rs$ and not $last_s = last_r$

# Generic Abstraction Function

| | | |
|---|---|---|
| *queue* | | the elements of $buf_r$ paired with *OK* |
| | | + the elements of *inflight-m* paired with ? |
| | | + *current-q* |
| *status* | (*false*, *OK*) | if $mode_s = rec$ |
| else | (?, *f*) | if $current\text{-}q = \langle(m, f)\rangle$ |
| | (?, *OK*) | if $mode_s = send$, $last_s = last_r$ and $buf$ $\quad$ $empty$ |
| | (*true*, *OK*) | if $mode_s = send$, $last_s = last_r$ and $buf = empty$ |
| | (*true*, ?) | if $mode_s = send$ and $last_s \in inflight_{rs}$ |
| | (*false*, *OK*) | if $mode_s = send$ and $last_s \notin (good_r \cup \{last_r\} \cup inflight_{rs})$ |
| | (*ack*, *OK*) | if $mode_s = acked$ |

# Practical Implementations

| Generic | Five-packet handshake | Liskov-Shrira-Wroclawski |
|---|---|---|
| $good_s$ | $\{id \mid (accept, jd_s, id) \in rs\}$ if $mode = needid$<br>$\{\}$ otherwise | $\{times_s\} - \{last_s\}$ |
| $good_r$ | $\{last_r\}$ if $mode_r = accept$<br>$\{\}$ otherwise | $\{i \mid \quad lower < i$<br>and ($i \leq upper$ or $mode_r = rec$) $\}$ |
| $last_r$ | $last_r$ if $mode \quad accept$<br>$nil$ otherwise | $last_r$ |
| $shrink$-$good_s$ | $mode = needid$ and $lose_{rs}(accept, jd_s, id)$ of last copy or $receive_{rs}(accept, jd_s, id)$, $ids = good_s - \{id\}$ | $tick(id)$, $id = times_s$ |
| $grow$-$good_s$ | $send_{rs}(accept, jd_s, id)$ | $tick(id)$ |
| $cleanup$ | $receive_{sr}(acked)$, $mode = ack$ | $cleanup$ |
| $shrink$-$good_r$ | $mode = accept$ and $receive_{sr}(acked, id_r, nil)$ | $increase$-$lower(id)$,<br>$ids = \{i \mid lower < i \quad id\}$ |
| $grow$-$good_r$ | $mode = idle$ and $receive_{sr}(needid, ...)$, | $increase$-$upper(id)$,<br>$ids = \{i \mid upper < i \quad id\}$ |

# Summary

Client specification of at-most-once messages.

Working spec which maximizes non-determinism.

Generic implementation with *good* identifiers maintained by magic.

Two practical implementations

- Five-packet handshake
- Liskov-Shrira-Wrowclawski

# To follow up ...

You can find these slides on research.microsoft.com/lampson, as well as a paper (item 47 in the list of publications).

L. Lamport, A simple approach to specifying concurrent systems, *Comm. ACM* **32**, 1, Jan. 1989.

M. Abadi and L. Lamport, The existence of refinement mappings, DEC SRC research report 29, Aug. 1988.