At-Most-Once Message Delivery
A Case Study in Algorithm Verification

Butler Lampson       Nancy Lynch       Jørgen Søgaard-Andersen,
Digital Equipment Corp. MIT MIT and
Department of Computer Science
Technical University of Denmark

1 Introduction

The at-most-once message delivery problem involves delivering a sequence of messages submitted by a user at one location to another user at another location. If no failures occur, all messages should be delivered in the order in which they are submitted, each exactly once. If failures (in particular, node crashes or timing anomalies) occur, some messages might be lost, but the remaining messages should not be reordered or duplicated.

This talk examines two of the best-known algorithms for solving this problem: the clock-based protocol of [3] and the five-packet interchange protocol of [2]. It is shown that both of these protocols can be understood as implementations of a common (untimed) protocol that we call the generic protocol. It is also shown that the generic protocol meets the problem specification.

The development is carried out in the context of (timed and untimed) automata [7, 8] and [6], using simulation techniques [7]. It exercises many aspects of the relevant theory, including timed and untimed automata, refinement mappings, forward and backward simulations, history and prophecy variables. The theory provides insight into the algorithms, and vice versa.

In this short paper, we simply give formal descriptions of the problem specification and of the two algorithms, leaving detailed discussion of the proof for the talk and for a later paper.

2 The Specification S

The transitions of the specification we use for the at-most-once message delivery problem are given below. Formally, the object denoted by the specification is an I/O automaton [5, 6]. The notation used is somewhat standard for describing I/O automata (see, for example, [4]). The user interface is a set of external (input and output) actions. Even though we in S have a central, i.e., not distributed, view of the system, the external actions can be logically partitioned into actions on the “sender” side (send_msg, ack, crash, and recover) and actions on the “receiver” side (receive_msg, crash, and recover). Furthermore, there is an internal action lose. All these actions then manipulate shared data structures like, e.g., queue.
send_msg(m)
Effect:
if rec_r = false then
append m to queue
status := ?

receive_msg(m)
Precondition:
rec_r = false
m is first on queue
Effect:
remove first element of queue
if queue is empty and
status = ? then
status := true

ack(b)
Precondition:
rec_s = false
status = b \in \{true, false\}
Effect:
none

crash_r
Effect:
rec_r := true

lose
Precondition:
rec_s = true or rec_r = true
Effect:
delete arbitrary elements of queue
if last element of queue is deleted then
status := false
else optionally
status := false

recover_s
Precondition:
rec_s = true
Effect:
rec_s := false

recover_r
Precondition:
rec_r = true
Effect:
rec_r := false

We specify fairness by partitioning the actions that the protocol controls (output and internal action) in fairness classes. In the execution of the protocol it must not be the case that actions from a fairness class are continuously enabled without actions from that class being executed infinitely often.

For the specification S we use the following five classes:

1. ack actions
2. receive_msg actions
3. recover_s
4. recover_r
5. lose

3 The Clock-Based Protocol C

Code for the clock-based protocol of [3] is given below. Since at this level of abstraction we have a distributed view of the system, the code is partitioned into code for the sender and code for the receiver part of the protocol. Formally, the sender and receiver protocols are timed automata in the style of [8].

In C, the sender protocol associates a time value with each message it wishes to deliver. The time values are obtained from a local clock. The receiver protocol uses
the associated time value to decide whether or not to accept a received message — as a rough strategy, it will accept a message provided the associated time is greater than the time of the last message that was accepted. However, the receiver protocol cannot always remember the time of the last accepted message: it might forget this information because of a crash, or simply because a long time has elapsed since the last message was accepted and it is no longer efficient to remember it. Thus, the receiver protocol uses safe time estimates determined from its own local clock to decide when to accept a message.

Correctness of this protocol requires that the two local clocks be synchronized to real time, to within a tolerance $\epsilon$, when crashes do not occur. It also requires reliability bounds and upper time bounds on the low-level channels connecting the sender and receiver protocols.

Sender

$send\_msg(m)$

Effect:

\[\text{if } mode_s \neq \text{rec} \text{ then} \]

append $m$ to $buf_s$

$choose\_id(m, t)$

Precondition:

$mode_s = \text{acked}$,

$m$ is first on $buf_s$,

time$_s = t$,

t $> last_s$,

Effect:

$mode_s := \text{send}$

remove first element of $buf_s$

current-$msg_s$, := $m$

last$_s := t$

$send\_pkt_s(m, t)$

Precondition:

$mode_s = \text{send}$,

current-$msg_s$, := $m$

last$_s := t$

Effect:

none

$receive\_pkt_r(t, b)$

Effect:

\[
\text{if } mode_r = \text{send} \text{ and} \]

last$_r = t \text{ then} \]

$mode_r := \text{acked}$

current-$ack_r$, := $b$

current-$msg_r$, := nil

$ack(b)$

Precondition:

$mode_r = \text{acked}$

$buf_r$ is empty

current-$ack_r$, := $b$

Effect:

none

$crash_s$

Effect:

$mode_s := \text{rec}$

recover$_s$

Precondition:

$mode_s = \text{rec}$

Effect:

$mode_s := \text{acked}$

last$_s := \text{time}_s$

empty $buf_s$

current-$msg_s$, := nil

current-$ack_s$, := false

$\text{tick}_s(t)$

Effect:

$\text{time}_s := t$

We only need one class of locally controlled actions for the sender protocol:
1. choose_id, send_pkt_r, ack, and recover_r actions

We put an upper time bound of \( l \) on all the classes, meaning that if actions from a class get enabled, then an action from that class must be executed within time \( l \) unless the actions are disabled in the meantime.

**Receiver**

\[
\text{receive pkt}_r(m,t)
\]

Effect:

if \( \text{mode}_r \neq \text{rec} \) then

if \( \text{lower}_r < t \leq \text{upper}_r \) then

\( \text{mode}_r := \text{rcvd} \)

add \( m \) to buffer

\( \text{last}_r := t \)

\( \text{lower}_r := t \)

else if \( \text{last}_r < t \leq \text{lower}_r \) then

add \( t \) to nack-buffer

else if \( \text{mode}_r = \text{idle} \) and

\( t = \text{last}_r \) then

\( \text{mode}_r := \text{ack} \)

\[
\text{crash}_r
\]

Effect:

\( \text{mode}_r := \text{rec} \)

\[
\text{recover}_r
\]

Precondition:

\( \text{mode}_r = \text{rec} \)

\( \text{upper}_r + 2c < \text{time}_r \)

Effect:

\( \text{mode}_r := \text{idle} \)

\( \text{last}_r := 0 \)

empty buff

\( \text{lower}_r := \text{upper}_r \)

\( \text{upper}_r := \text{time}_r + \beta \)

empty nack-buff

\[
\text{receive msg}(m)
\]

Precondition:

\( \text{mode}_r = \text{rcvd} \)

\( m \) is first on buf

Effect:

remove first element of buff

if buff is empty then

\( \text{mode}_r := \text{ack} \)

\[
\text{send pkt}_r(t, true)
\]

Precondition:

\( \text{mode}_r = \text{ack} \)

\( \text{last}_r = t \)

Effect:

\( \text{mode}_r := \text{idle} \)

\[
\text{send pkt}_r(t, false)
\]

Precondition:

\( \text{mode}_r \neq \text{rec} \)

\( t \) is first on nack-buff

Effect:

remove first element of nack-buff

For the receiver protocol we use the following classes of locally controlled actions:

1. receive_msg, send_pkt_r(true), and recover_r actions
2. send_pkt_r(false) actions
3. increase-lower actions
4. increase-upper actions
4 The Five-Packet Protocol $5P$

Code for the five-packet handshake protocol of [2] is given below. As for $C$, the code is partitioned into code for the sender protocol and code for the receiver protocol. For the $5P$ protocol we assume that the sender and receiver protocols communicate via channels that may lose or duplicate packets, the latter only a finite number of times for each packet instance. In order to prove liveness properties of the $5P$ protocol, we furthermore assume that if the same packet is sent an infinite number of times, then it will also be received an infinite number of times.

In this protocol, for each message that the sender protocol wishes to deliver, there is an initial exchange of packets between the sender and receiver protocols to establish a commonly-agreed-upon message identifier. The sender protocol then associates this identifier with the message. The receiver protocol uses the associated identifier to decide whether or not to accept a received message – it will accept a message provided the associated identifier is current. Additional packets are required in order to tell the receiver protocol when it can throw away a current identifier.
4.1 Sender

send_msg(m)
Effect:
if mode ≠ rec then
append m to buf,

choose_jd(jd)
Precondition:
mode = acked,
m first on buf,
jd ≠ jd-used,
Effect:
mode := needid
jd := jd
add jd to jd-used,
remove first element of buf,
current-msg := m

send_pkt(needid,nil,jd)
Precondition:
mode = needid, jd = jd,
Effect:
none

receive_pkt(accept,jd,id)
Effect:
if mode ≠ rec then
if mode = needid and jd = jd, then
mode := send
id := id
add id to the end of used,
else if id ≠ id, then
add id to the end of acked-buf,

send_pkt(send,id,m)
Precondition:
mode = send,
id = id,
m = current-msg,
Effect:
none

receive_pkt(ack,id,b)
Effect:
if mode ≠ rec then
if mode = send and id = id, then
mode := acked
current-ack := b
jd := nil
id := nil
current-msg := nil
if b = true then
add id to acked-buf,

send_pkt(acked,id,nil)
Precondition:
id is first on acked-buf,
Effect:
remove first element of acked-buf

ack(b)
Precondition:
mode = acked, buf is empty,
b = current-ack,
Effect:
none

crash,
Effect:
mode := rec

recover,
Precondition:
mode = rec
Effect:
mode := acked
jd := nil
id := nil
empty buf
current-msg := nil
current-ack := false
empty acked-buf

grow-jd-used,
Precondition:
none
Effect:
add some JDs to jd-used

We define the following fairness classes of the locally controlled actions of the sender
protocol:

1. $ack, choose-jd(jd), send_{pkt_r}(needid, ), send_{pkt_r}(send, ),$ and $recover, actions$
2. $send_{pkt_r}(acked, )$ actions
3. $grow-jd-used$

4.2 Receiver

receive_{pkt_r}(needid, nil, jd)
Effect:
if $mode_r = idle$ then
$mode_r := accept$
choose an id not in issued
$jd_r := jd$
$id_r := id$
add id to issued

send_{pkt_r}(ack, id, false)
Precondition:
$mode_r \neq rec,$
$id$ is first on nack-buf_r
Effect:
remove first element of nack-buf_r

receive_{pkt_r}(acked, id, nil)
Effect:
if $(mode_r = accept$ and
$id = id_r)$ or
$(mode_r = ack$ and
$id = last_r)$ then
$mode_r := idle$
$jd_r := nil$
$id_r := nil$
$last_r := nil$

send_{pkt_r}(ack, id, true)
Precondition:
$mode_r = ack, id = last_r$
Effect:
none

crash_r
Effect:
$mode_r := rec$

recover_r
Precondition:
$mode_r = rec$
Effect:
$mode_r := idle$
$jd_r := nil$
$id_r := nil$
$last_r := nil$
empty buf_r
empty nack-buf_r

grow-issued_r
Precondition:
none
Effect:
add some IDs to issued_r
We define the following three fairness classes of the locally controlled actions of the receiver protocol:

1. \texttt{receive\_msg}, \texttt{recover}_r, \texttt{send\_pkt}_r(\texttt{accept}, \_), and \texttt{send\_pkt}_r(\texttt{ack}, \_\_true) actions.
2. \texttt{send\_pkt}_r(\texttt{ack}, \_\_false) actions.
3. \texttt{grow\_issued}_r.

5 Discussion

Both protocols share a common high-level description: both involve association of identifiers with messages, and acceptance of messages by the receiver based on recognition of "good" identifiers. Both also involve very similar strategies for acknowledgement of messages. It is thus desirable to base correctness proofs on this common structure.

We define a high-level (untimed) generic protocol \( G \), which represents the common structure, and show that both \( C \) and \( SP \) implement \( G \). We also show that the generic protocol meets the problem specification \( S \). The proof that \( G \) satisfies \( S \) uses a backward simulation \cite{7} (or prophecy variables \cite{1}). The proof that \( SP \) implements \( G \) uses a forward simulation \cite{7} (or history variables \cite{9}). The proof that \( C \) implements \( G \) uses a timed forward simulation \cite{7}.

References


This article was processed using the \LaTeX{} macro package with LLNCS style.