At-Most-Once Message Delivery

A Case Study in Algorithm Verification

Butler Lampson
Digital Equipment Corp.

Nancy Lynch
MIT

Jørgen Søgaard-Andersen,
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, $crash_s$, and $recover_s$) and actions on the "receiver" side ($receive\_msg$, $crash_r$, and $recover_r$). Furthermore, there is an internal action $lose$. All these actions then manipulate shared data structures like, e.g., $queue$.

send_msg(m)
  Effect:
    if $rec_s = false$ 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

$crash_s$
  Effect:
    $rec_s := true$

$crash_r$
  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 ϵ, 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:
if $mode_s \neq$ rec then
append m to buf_s

choose_id(m, t)
Precondition:
$mode_s =$ acked,
m is first on buf_s,
$time_s = t$,
$t > last_s$
Effect:
$mode_s :=$ send
remove first element of buf_s
$current\text{-}msg_s := m$
$last_s := t$

send_pkt$_{sr}(m, t)$
Precondition:
$mode_s =$ send,
$current\text{-}msg_s = m$,
$last_s = t$
Effect:
none

receive_pkt$_{rs}(t, b)$
Effect:
if $mode_s =$ send and
$last_s = t$ then
$mode_s :=$ acked
$current\text{-}ack_s := b$
$current\text{-}msg_s :=$ nil

ack(b)
Precondition:
$mode_s =$ acked
buf_s is empty
$current\text{-}ack_s = b$
Effect:
none

crash$_s$
Effect:
$mode_s :=$ rec

recover$_s$
Precondition:
$mode_s =$ rec
Effect:
$mode_s :=$ acked
$last_s := time_s$
empty buf_s
$current\text{-}msg_s :=$ nil
$current\text{-}ack_s := false$

tick$_s(t)$
Effect:
$time_s := t$

We only need one class of locally controlled actions for the sender protocol:

1. *choose_id*, $send\_pkt_{sr}$, *ack*, and $recover_s$ 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

$receive\_pkt_{sr}(m, t)$
  Effect:
    if $mode_r \neq$ rec then
      if $lower_r < t \leq upper_r$ then
        $mode_r :=$ rcvd
        add m to $buffer_r$
        $last_r := t$
        $lower_r := t$
      else if $last_r < t \leq lower_r$ then
        add t to $nack\text{-}buffer_r$
      else if $mode_r =$ idle and
          $t = last_r$ then
        $mode_r =$ ack

$receive\_msg(m)$
  Precondition:
    $mode_r =$ rcvd,
    m is first on buf_r
  Effect:
    remove first element of buf_r
    if buf_r is empty then
      $mode_r :=$ ack

$send\_pkt_{rs}(t, true)$
  Precondition:
    $mode_r =$ ack,
    $last_r = t$
  Effect:
    $mode_r :=$ idle

$send\_pkt_{rs}(t, false)$
  Precondition:
    $mode_r \neq$ rec
    t is first on $nack\text{-}buf_r$
  Effect:
    remove first element of $nack\text{-}buf_r$

$crash_r$
  Effect:
    $mode_r :=$ rec

$recover_r$
  Precondition:
    $mode_r =$ rec,
    $upper_r + 2\epsilon < time_r$
  Effect:
    $mode_r :=$ idle
    $last_r := 0$
    empty buf_r
    $lower_r := upper_r$
    $upper_r := time_r + \beta$
    empty $nack\text{-}buf_r$

$increase\text{-}lower(t)$
  Precondition:
    $mode_r \neq$ rec,
    $lower_r \leq t < time_r - \rho$
  Effect:
    $lower_r := t$

$increase\text{-}upper(t)$
  Precondition:
    $mode_r \neq$ rec,
    $upper_r \leq t = time_r + \beta$
  Effect:
    $upper_r := t$

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

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

1. *receive_msg*, $send\_pkt_{rs}(, true)$, and $recover_r$ actions
2. $send\_pkt_{rs}(, false)$ 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 dublicate 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 $mode_s \neq$ rec then
      append m to buf_s

choose_jd(jd)
  Precondition:
    $mode_s =$ acked,
    m first on buf_s,
    $jd \notin jd\text{-}used_s$
  Effect:
    $mode_s :=$ needid
    $jd_s := jd$
    add jd to $jd\text{-}used_s$
    remove first element of buf_s
    $current\text{-}msg_s := m$

send_pkt$_{sr}$(needid, nil, jd)
  Precondition:
    $mode_s =$ needid, $jd = jd_s$
  Effect:
    none

receive_pkt$_{rs}$(accept, jd, id)
  Effect:
    if $mode_s \neq$ rec then
      if $mode_s =$ needid and
        $jd = jd_s$ then
        $mode_s :=$ send
        $id_s := id$
        add id to the end of $used_s$
      else if $id \neq id_s$ then
        add id to the end
          of $acked\text{-}buf_s$

send_pkt$_{sr}$(send, id, m)
  Precondition:
    $mode_s =$ send,
    $id = id_s$,
    $m = current\text{-}msg_s$
  Effect:
    none

receive_pkt$_{rs}$(ack, id, b)
  Effect:
    if $mode_s \neq$ rec then
      if $mode_s =$ send and
        $id = id_s$ then
        $mode_s :=$ acked
        $current\text{-}ack_s := b$
        $jd_s :=$ nil
        $id_s :=$ nil
        $current\text{-}msg_s :=$ nil
      if $b = true$ then
        add id to $acked\text{-}buf_s$

send_pkt$_{sr}$(acked, id, nil)
  Precondition:
    id is first on $acked\text{-}buf_s$
  Effect:
    remove first element of $acked\text{-}buf$

ack(b)
  Precondition:
    $mode_s =$ acked, buf_s is empty,
    $b = current\text{-}ack_s$
  Effect:
    none

crash$_s$
  Effect:
    $mode_s :=$ rec

recover$_s$
  Precondition:
    $mode_s =$ rec
  Effect:
    $mode_s :=$ acked
    $jd_s :=$ nil
    $id_s :=$ nil
    empty buf_s
    $current\text{-}msg_s :=$ nil
    $current\text{-}ack_s := false$
    empty $acked\text{-}buf_s$

grow-jd-used$_s$
  Precondition:
    none
  Effect:
    add some JDs to $jd\text{-}used_s$

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

1. *ack*, *choose_jd*(jd), $send\_pkt_{sr}$(**needid**, ,), $send\_pkt_{sr}$(**send**, ,), and $recover_s$ actions
2. $send\_pkt_{sr}$(**acked**, ,) actions
3. $grow\text{-}jd\text{-}used_s$

4.2 Receiver

$receive\_pkt_{sr}$(needid, nil, jd)
  Effect:
    if $mode_s$ = idle then
      $mode_r$:= accept
      choose an id not in $issued_r$
      $jd_r := jd$
      $id_r := id$
      add id to $issued_r$

$send\_pkt_{rs}$(accept, jd, id)
  Precondition:
    $mode_r$ = accept,
    $jd = jd_r$,
    $id = id_r$
  Effect:
    none

$receive\_pkt_{sr}$(send, id, m)
  Effect:
    if $mode_r \neq$ rec then
      if $mode_r$ = accept and
        $id = id_r$ then
        $mode_r$:= rcvd
        append m to buf_r
        $last_r := id$
      else if $id \neq last_r$ then
        append id to $nack\text{-}buf_r$

$receive\_msg$(m)
  Precondition:
    $mode_r$ = rcvd, m first on buf_r
  Effect:
    remove the first element of buf_r
    if buf_r is empty then
      $mode_r$:= ack

$send\_pkt_{rs}$(ack, id, $true$)
  Precondition:
    $mode_r$ = ack, $id = last_r$
  Effect:
    none

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

$receive\_pkt_{sr}$(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

$crash_r$
  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 $nack\text{-}buf_r$

$grow\text{-}issued_r$
  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. *receive_msg*, $recover_r$, $send\_pkt_{rs}$(**accept**, ,), and $send\_pkt_{rs}$(**ack**, , *true*) actions
2. $send\_pkt_{rs}$(**ack**, , *false*) actions
3. $grow\text{-}issued_r$

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 $5P$ implement G. We also show that the generic protocol meets the problem specification S. The proof that G satisfies S uses a *backward simulation* [7] (or *prophecy variables* [1]). The proof that $5P$ implements G uses a *forward simulation* [7] (or *history variables* [9]). The proof that C implements G uses a *timed forward simulation* [7].

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. In *Proceedings of the 3rd Annual Symposium on Logic in Computer*, pages 165–175, Edinburgh, Scotland, July 1988.
2. D. Belsnes. Single message communication. *IEEE Transactions on Communications*, Com-24(2), February 1976.
3. B. Liskov, L. Shrira, and J. Wroclawski. Efficient at-most-once messages based on synchronized clocks. Technical Report MIT/LCS/TR-476, Laboratory for Computer Science, Massachusetts Institute of Technology, April 1990.
4. N. Lynch and I. Saias. Distributed Algorithms. Fall 1990 Lecture Notes for 6.852. MIT/LCS/RSS 16, Massachusetts Institute of Technology, February 1992.
5. N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Techical Report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, 02139, April 1987.
6. N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.
7. N. Lynch and F. Vaandrager. Forward and backward simulations for timing-based systems. In *Proceedings of REX Workshop "Real-Time: Theory in Practice"*, Mook, The Netherland, 1992. Springer-Verlag, LNCS 600.
8. F. Modugno, M. Merritt, and M. Tuttle. Time constrained automata. In *CONCUR'91 Proceedings Workshop on Theories of Concurrency: Unification and Extension*, Amsterdam, August 1991.
9. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6(4):319–340, 1976.

This article was processed using the LaTeX macro package with LLNCS style