Hula: An Efficient Protocol for Reliable Delivery of Messages

Umesh Maheshwari
Technical Report MIT-LCS-TR-720
MIT Laboratory for Computer Science
umesh@lcs.mit.edu

July 1997

Abstract

We present a new protocol for reliable delivery of messages over a network that might lose, duplicate, reorder, or arbitrarily delay packets. It is the first protocol that guarantees exactly-once and ordered delivery on a connection while avoiding precursory handshakes. Avoiding handshakes reduces the overhead for sending small, intermittent messages as in remote procedure calls and protocols like HTTP. Like other practical protocols, it permits discarding information for idle connections.

The protocol works by combining existing handshake-based and time-based protocols. It uses loosely synchronized clocks to avoid handshakes. A handshake is executed only upon an unexpectedly long packet delay or clock skew. Thus, unexpected conditions degrade performance but do not compromise reliability. The resultant protocol has the reliability of handshake-based protocols and the efficiency of time-based protocols.

1 Introduction

Reliable delivery of messages is useful for many facilities such as remote procedure calls and file transfer. These facilities require that messages be delivered exactly once and in the order they were sent. Often, such delivery must be provided using an underlying network that may lose, duplicate, reorder, or arbitrarily delay packets; the Internet is a well-known example of such a network. Many protocols have been designed for this purpose, such as TCP [Pos81], Birrell and Nelson's RPC [BN84], Delta-t [Wat89], and SCMP [LSW91], but they fall short of either reliability or efficiency.

In particular, protocols that guarantee reliable delivery over unreliable networks with unbounded delays require an exchange of mutual information, called a *handshake*, before two processes can communicate [Tom75, Bel76]. The handshake delays communication and consumes resources. This cost is amortized over the messages sent between the communicating processes until they discard mutual information. The cost is acceptable when sending large messages or a large number of messages, as in transferring large files or in a remote login session. However, the cost is significant when sending a few small messages at a time as in remote procedure calls [BN84] and in protocols such as HTTP used in the Web. The increasing use of these facilities makes it important to avoid initial handshakes.

On the other hand, some time-based protocols avoid handshakes by making assumptions about the time characteristics of the underlying network, but they fail when those assumptions do not hold. For example, some protocols assume that no copy of a packet may be present in the network after its maximum lifetime has passed [FW78, BN84, Wat89]. If a duplicate packet survives for longer, these protocols might deliver duplicate messages. Another protocol relies on loosely-synchronized clocks at hosts and bounded packet delay so that old information can be discarded [LSW91]. If clock skews or packet delays are longer, this protocol might reject valid messages.

Thus, existing time-based protocols do not guarantee reliable delivery in a network where packet delays or clock skews may vary unexpectedly. The Internet is such a network: packet delays vary geographically with distances and physical links, and also temporally with network congestion. Mobile computing increases this variance since processes may change locations and there may be periods of poor connectivity between them. At the same time, the use of the Internet for critical applications such as electronic commerce increases the need for a reliable protocol.

In essence, handshake-based protocols are pessimistic since they assume that delays and clock skews cannot be predicted. Thus, they are reliable but inefficient. On the other hand, time-based protocols are optimistic since they assume upper bounds on packet delays and clock skews. Thus, they are efficient but unreliable.

This paper presents a new protocol, *Hula*, that has the reliability of handshake protocols, and efficiency close to that of time-based protocols. This protocol has two parts. The first part uses the synchronized-clock protocol to classify packets as either new, duplicate, or suspected. A packet is suspected only when it seems to have arrived too late; this might happen because of an unexpected delay or an unexpected clock skew. The second part checks whether a suspected message is new or duplicate using a handshake. (Hula stands for "Handshake

Upon Late Arrival.") The handshake ensures reliability but is executed infrequently. Thus, if time-based assumptions do not hold, performance degrades but correctness is not affected.

Like other practical protocols designed for large networks, Hula allows processes to discard mutual information when they are not communicating actively. In addition, Hula provides control over the interval for which communicating processes maintain mutual information. It exposes a continuum of tradeoff between discarding mutual information and avoiding handshakes. In fact, the handshake protocol can be viewed as an extreme case of Hula when mutual information is discarded as soon as possible.

Also, like other practical protocols, Hula tolerates host crashes while maintaining a weaker form of reliable delivery. Specifically, Hula might lose messages upon a crash, but it does not duplicate or reorder messages. This is known as "at-most once" delivery, as opposed to "exactly-once" delivery.

The rest of this paper is organized as follows. Section 2 defines the problem and the underlying system more precisely. Section 3 describes existing protocols since our protocol is based on them. Section 4 describes the new protocol. Section 5 contains our conclusions and indicates directions for future work.

2 The Model

We use the simple model shown in Figure 1. An application that uses reliable delivery is called a *process* and the underlying system that provides this service is called the *host*. We assume that each process has a globally unique id for communicating with other processes. To send a message, a process passes the message and the id of the remote process to its host. The host uses the network to exchange *packets* with the remote host. A packet may carry a message or some control information or both. Eventually, the remote host passes the message to the remote process.

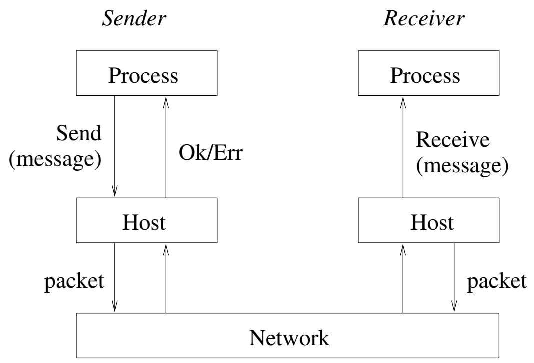


Figure 1: The model.

This paper ignores issues such as full-duplex transfer, flow control, and fragmentation and reassembly of messages. While these issues are important in a transport protocol, previously known solutions can be applied to our protocol.

2.0.1 Requirements

Here we specify the safety, fault-tolerance, liveness, and performance requirements for a reliable-delivery protocol.

Safety. Reliable delivery means that messages sent by one process to another must be received exactly once and in the order they were sent. We say that a reliable *connection* exists between the two processes.

Fault-tolerance. A host might crash and lose the information stored in volatile memory, but it retains the information stored in stable memory such as the disk. It is expensive to guarantee reliable delivery across host crashes, because that requires logging information on stable storage every time a message is sent or received [LLSA93]. Therefore, we weaken the reliability requirement upon crashes. When a host *S* crashes, the last message sent by a process on *S* might be lost. When host *R* crashes, the last messages sent to a process on *R*—until *R* recovers—might be lost. Because of this possibility, it is desirable to inform the sender process whether or not its last message was successfully delivered to the receiver. Therefore, the sender host returns an *Ok* or *Error* to the sender process. However, it is impossible to provide an accurate acknowledgement to the sender unless information is updated stably every time a message is delivered. Therefore, in practice, there might be two discrepancies:

False positives An *Ok* is returned for a message not delivered to the receiver process.

False negatives An *Error* is returned for a message delivered to the receiver process.

Usually, false negatives are considered the lesser evil [Bel76, LLSA93]. Thus, our safety and fault-tolerance requirements are as follows:

- As long as there is no crash: exactly-once and ordered delivery.
- Upon a crash: possible loss of messages and false negative acknowledgements until recovery.

Note that when the sender crashes, the crash serves as an implicit *Error* for unacknowledged messages.

Liveness. A message sent by a process must be eventually received by the receiver unless one of the hosts crashes. Further, after coming back from a crash, a process must be able to resume reliable delivery eventually.

Performance. The protocol must not require the hosts to maintain information about a connection when the associated processes are not communicating actively, *i.e.*, when the connection is idle for an arbitrarily long period. We say that a connection is *open* at a host if the host has stored some information about it; otherwise, we say that the connection is *closed*. A *connection table* in each host maps open connections to the associated information. A host may close a connection heuristically or on a directive from the communicating process. In the absence of either, the host may close a connection after every message sent. A connection between two processes may be opened and closed many times, but we regard the open intervals as different phases of the same connection. The safety requirement is that messages sent on a connection must be delivered exactly once and in order—even if they were sent in different phases, since phases are merely a performance optimization.

If a protocol required the hosts to maintain connection information, *i.e.*, if it did not allow hosts to close and re-open connections, the connection table would grow very large due to processes such as Web servers and clients, which communicate with many other processes in their lifetimes. A big connection table is undesirable because the connection table must be accessed whenever a packet is sent or received and indexing a large table is inefficient. If the table is paged out to disk, disk accesses would overwhelm packet delays.

2.0.2 Underlying Support

The network may delay, lose, duplicate, or reorder packets. We assume that corrupt packets are detected using checksums and are rejected. Further, if a packet is sent repeatedly, an un-corrupted copy of it is eventually received by the remote host. We also assume that a crashed host eventually recovers.

Sites are equipped with clocks that are loosely synchronized. Low overhead protocols such as the Network Time Protocol are already in use for synchronizing clocks within 100 milliseconds [Mil88]. The clock survives host crashes. It is fast enough that it ticks at least once between successive messages sent. For current systems, a 1 μs tick would suffice. Also, the clock is slow enough and has enough bits such that its wrap-around period is much longer than the maximum expected delivery time or lifetime of packets. At 1 μs tick, a 64-bit clock would suffice for foreseeable future. Therefore, we ignore all problems related with wrap-around.

2.0.3 Notation

To describe various protocols, we consider messages sent by a process on host *S*, the sender, to a process on host *R*, the receiver. The information at *S* for this connection is denoted by *s*, and that at *R* is denoted by *r*. A connection *s* has a state, *s.state*, which may be CLOSED, and additional fields when the state is not CLOSED. The current clock values at *S* and *R* are denoted by *S.t* and *R.t*.

The directives at the process-host interface are Send, Receive, Ok and Error. In addition, either the sender process or some heuristic within *S* issues the directive Close as a hint to close the connection.

Each packet contains a label identifying the kind of packet and some fields; it is denoted as *label[f1, ..., fn]*. Each packet also contains the ids of the communicating processes to identify the connection, but we do not show these fields explicitly.

We describe each protocol using a state machine. For each state, we give the possible transitions using a notation similar to Dijkstra's guarded command language. The transition, *input*→*actions*, means that it is enabled on the given *input* and results in the given *actions*. An input may be a packet received with specified label and fields or an input directive from the process. An action may be updating the connection, sending a packet, or an output directive to the process. Consider the following example for transitions from STATE1:

<u>*s.state* = STATE1</u>
Send(d) → *s.data := d, s.state :=* STATE2
pkt1[s.i, j] → *s.j := j*
other[i] → *pkt2[i]*
* → *pkt3[s.i]*

This code means that four transitions are possible when *s* is in state STATE1. The first transition is enabled on receiving the directive Send; it results in storing the associated data in *s.data* and switching *s.state* to STATE2. The second transition is enabled on receiving *pkt1[i, j]* but only when *i* matches *s.i*. It results in setting *s.j* to *j*. The packet label *other* in the third transition matches any packet not matched by previous transitions. This transition sends *pkt2[i]*, where *i* is the first field in the packet received. The fourth transition has the special input *, which means it is enabled periodically. It results in sending the packet *pkt3[s.i]* repeatedly. If a packet arrives for which there is no matching input, it is ignored.

3 Background

This section describes existing protocols for reliable delivery since our protocol is based on them.

3.1 Stenning's Protocol

Stenning's protocol is the simple basis for reliable delivery [Ste76]. *S* stamps each message sent on a connection sequentially, while *R* accepts a message only if it has the expected stamp. *S* sends a message repeatedly until it receives an acknowledgement with the same stamp from *R*.

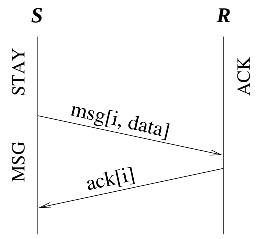


Figure 2: Stenning's protocol (time increases downward).

Figure 2 illustrates the protocol. The connection at *S* has two states: STAY and MSG. The stamp of the last message sent on *s* is stored in *s.i*. The state machine for *s* is given below.

<u>*s.state* = STAY</u>
Send(d) → *s.i := s.i+1, s.data := d, s.state :=* MSG

<u>*s.state* = MSG</u>
* → *msg[s.i, s.data]*
ack[s.i] → *Ok(), s.state :=* STAY

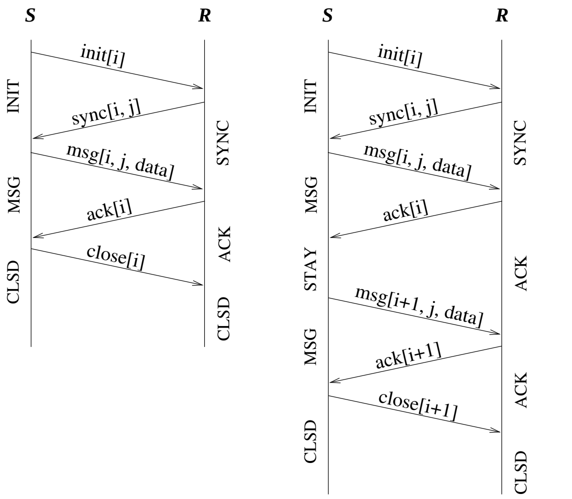
The state machine for the connection at *R* is given below. The stamp of the last message received on *r* is stored in *r.i*. Initially, *r.i* and *s.i* are set to the same value, say, zero.

<u>*r.state* = ACK</u>
msg[r.i, d] → *ack[r.i]*
msg[r.i+1, d] → *r.i := r.i+1, Receive(d), ack[r.i]*

The problem with this simple protocol is that it does not allow hosts to close and re-open a connection: *S* must remember *s.i* and *R* must remember *r.i*. Also, it does not handle host crashes.

3.2 Handshake-based Protocols

Handshake-based protocols allow hosts to close a connection and re-open it by sending extra packets to re-initialize mutual information. The most common handshake protocol is the one by Tomlinson [Tom75]. This protocol sends four packets when a connection is opened to send a single message. Belsnes added a fifth packet to the protocol to provide the crash semantics specified in Section 2, *i.e.*, to avoid false positive acknowledgements. Figure 3(i) shows a slightly modified version of Belsnes's protocol. TCP uses a similar protocol [Pos81, JBB92], but it differs operationally because of duplex transfer, because TCP is meant to transfer a stream of bytes rather than messages, and because of the possibility of sequence numbers wrapping around.



(i) sending one message (ii) sending two messages

Figure 3: The five-packet handshake protocol.

A generic packet in this protocol has the format *[i, j, data]*, where *i* and *j* are stamps generated by *S* and *R* respectively. When a host opens a connection, it generates a stamp using its clock. The clock acts as a shared initializer for all connections at the host and avoids the need to remember connection-specific information while the connection is closed.

When a single message is sent during a phase, the connection at *S* goes through states INIT, MSG, and CLOSED. We have extended the protocol by adding a state, STAY, which allows the sender process to send more messages before closing the connection. Each additional message results in two more packets (*msg* and *ack*); this is illustrated in Figure 3(ii). The state machine for the connection at *S* is given below. Transitions that represent corrective steps due to the unreliability of the underlying system are marked with †. Thus, the best-case behavior of the protocol is given by transitions not marked in this way. We do not explain the transitions in words due to lack of space; we urge the reader to read the code below.

<u>*s.state* = CLOSED</u>
Send(d) → *s.i := S.t, s.data := d, s.state :=* INIT
†*other[i]* → *close[i]*

<u>*s.state* = INIT</u>
* → *init[s.i]*
sync[s.i, j] → *s.j := j, s.state :=* MSG
†*other[i]* → *close[i]*

<u>*s.state* = MSG</u>
* → *msg[s.i, s.j, s.data]*
ack[s.i] → *Ok(), s.state :=* STAY
†*close[s.i]* → *Error(), s.state :=* CLOSED

<u>*s.state* = STAY</u>
Send(d) → *s.i := s.i+1, s.data := d, s.state :=* MSG
Close() → *close[s.i], s.state :=* CLOSED
†*sync[i, j]* ∧ *i*≠*s.i* → *close[i]*

The state machine for the connection at *R*, denoted by *r*, works as follows.

<u>*r.state* = CLOSED</u>
init[i] → *r.i := i, r.j := R.t, r.state :=* SYNC
†*msg[i, j, d]* → *close[i]*

<u>*r.state* = SYNC</u>
* → *sync[r.i, r.j]*
msg[r.i, r.j, d] → *Receive(d), ack[r.i], r.state :=* ACK
†*close[r.i]* → *r.state :=* CLOSED

<u>*r.state* = ACK</u>
* → *ack[r.i]*
close[r.i] → *r.state :=* CLOSED
msg[r.i+1, j, d] → *Receive(d), r.i := r.i+1*

This protocol has the desired crash semantics. Upon recovery from a host crash, all connections at the host are closed due to loss of information. Further, the protocol ensures that these connections will be closed at peer hosts and that negative acknowledgements are sent as needed. For a formal proof of correctness of this protocol, see [LLSA93].

3.3 Time-based Protocols

Time-based protocols allow hosts to discard connection information based on assumptions about the underlying network. There are two kinds of such protocols. The first assumes that packets do not linger in the network for longer than some maximum lifetime and discards connection information after this period [BN84, Wat89]. However, if duplicate messages linger in the network for longer, this protocol might accept them. The second kind of time-based protocol assumes that hosts have loosely synchronized clocks and that packets are delivered within some period [LSW91]. We describe this protocol below because our protocol uses it.

The synchronized-clock protocol is illustrated in Figure 4. S stamps the message packets with its clock. R remembers the stamp of the last message received on the connection until the stamp is older than the current time at R by more than a chosen δ. The maximum stamp associated with any connection so closed is remembered in a variable, *max_closed.* If R receives a message packet on a closed connection, it checks the stamp. If the stamp is above *max_closed*, the packet must be new and is accepted. Otherwise, the packet is likely to be a duplicate and is rejected. This protocol assumes that the sum of the maximum packet delay (including retransmission) and the maximum clock skew between hosts is less than δ. Otherwise, a new message that seems to arrive too late might be rejected.

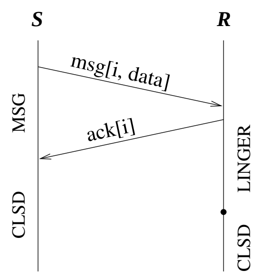


Figure 4: A time-based protocol.

Below, we have modified the protocol slightly from its original description to provide an explicit negative acknowledgement when a message might have been lost. The connection at the sender works as follows.

<u>*s.state* = CLOSED</u>
Send(d) $\rightarrow$ *s.i := S.t, s.data := d, s.state :=* MSG

<u>*s.state* = MSG</u>
* $\rightarrow$ *msg[s.i, s.data]*
ack[s.i] $\rightarrow$ *Ok(), s.state :=* CLOSED
†*close[s.i]* $\rightarrow$ *Error(), s.state :=* CLOSED

The connection at the receiver works as follows.

<u>*r.state* = CLOSED</u>
msg[i, d] $\wedge$ *i > max_closed* $\rightarrow$ *Receive(d), ack[i]*
r.i := i, r.state := LINGER
†*msg[i, d]* $\wedge$ *i* $\leq$ *max_closed* $\rightarrow$ *close[i]*

<u>*r.state* = LINGER</u>
msg[i, d] $\wedge$ *i > r.i* $\rightarrow$ *Receive(d), ack[i], r.i := i*
†*msg[r.i, d]* $\rightarrow$ *ack[i]*
r.i < R.t–δ $\rightarrow$ *max_closed := max(r.i, max_closed),*
r.state := CLOSED

If host R crashes, all connections at R are closed. In order not to accept a packet after the crash that is stamped below any packet accepted before the crash, R maintains a stable variable, *max_received*, such that *max_received* $\geq$ *msg.i* for any msg packet accepted. When R receives a msg packet such that *msg.i > max_received*, it must not accept *msg* before updating *max_received* stably. To avoid updating *max_received* frequently, R stores a higher value than required. The prescribed value is $R.t{+}\beta$, where β depends on the desired frequency of stable updates.

After a crash, R avoids the risk of accepting duplicates by setting *max_closed* to *max_received.* This means that messages whose stamps are lower than the value of *max_received* stored before the crash are not accepted, even though they might be new. There is a tradeoff for β: The larger it is, the less frequently is *max_received* updated, but the more the messages that must be rejected upon recovery from a crash.

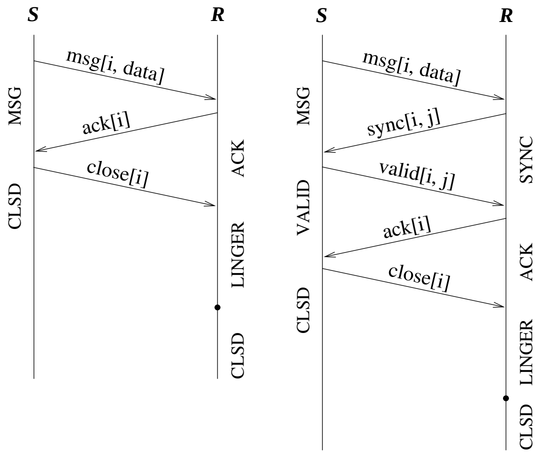
4 Hula

The Hula protocol has two parts. The first uses the synchronized-clock protocol to classify messages as either new, duplicate, or suspected. The second part checks if a suspected message is in fact duplicate using a handshake. Section 4.1 describes the basic protocol in the absence of host crashes, Section 4.2 describes how remote procedure calls may be conducted using Hula, and Section 4.3 extends the protocol to deal with host crashes.

4.1 The Basic Protocol

In the synchronized-clock protocol, the receiver accurately identifies a message packet as new or duplicate, except when the connection is closed and the packet is stamped less than or equal to *max_closed.* In Hula, these packets are called "suspected duplicates" and are treated differently from packets that are known to be duplicates for sure. Specifically, the receiver host treats a suspected message packet as the open packet in the handshake protocol. That is, it buffers the message without delivering it to the receiver process, and sends a *sync* packet to the sender. It delivers the buffered data only upon receiving a *valid* packet that correctly responds to the sync packet. Figures 5(i) and (ii) show the packets sent to handle a non-suspect and a suspect message respectively.

Because of the clock protocol, Hula executes a handshake only when packet delays or clock skews are unexpectedly large. Hula could alternatively use the maximum-lifetime protocol, which does not use synchronized clocks. However, any message sent after a sufficient gap since the last message



(i) A non-suspect message *(ii) A suspect message*

Figure 5: The Hula protocol.

on the connection would be "suspected new" and would require a handshake to check if it is in fact new. Therefore, if Hula used the maximum-lifetime protocol, it would execute handshakes more frequently.

Whether or not a connection is opened with a handshake, Hula requires an explicit close packet as in the five-packet protocol. Unlike the five-packet protocol, Hula requires this packet even if host crashes were not an issue. The close packet ensures that the receiver will not close the connection until the sender has received the ack.

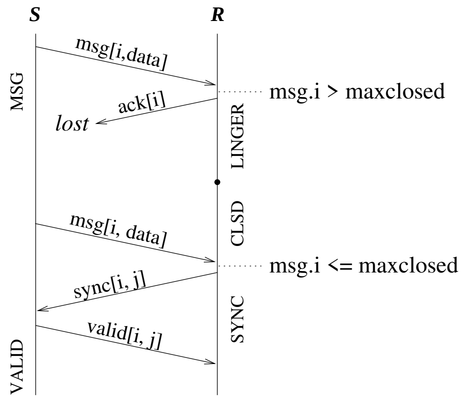


Figure 6: Incorrect execution in the absence of *close*.

Figure 6 illustrates what might go wrong if the receiver were to close the connection after sending the ack. When *S* first sends *msg[i, data]*, *R* finds that $msg.i > max\_closed$. So *R* accepts it and sends *ack[msg.i]*, but the ack is lost. After some time, *S* sends *msg[i, data]* again. However, in the meantime, *R* closed *r*, and now $msg.i \leq max\_closed$. Now, *R* suspects *msg* and sends *sync[i, j]*. *S* then sends *valid[i, j]*, which causes *R* to accept *msg* again.

The state machine for the connection at the sender is presented below. As in the extended five-packet protocol, it includes a STAY state to allow *S* to send more than one message before closing the connection. Figure 7 shows the state diagram for *s*. In this diagram, transitions are labeled as "input / actions". For simplicity, it does not show the actions related to updating the connection's fields and corrective transitions. Corrective transitions are marked by † in the code.

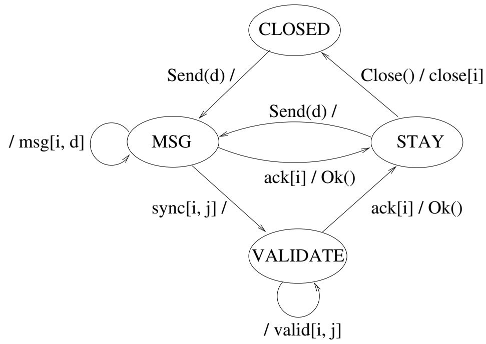


Figure 7: State diagram for Hula sender.

<u>*s.state =* CLOSED</u>
Send(d) $\rightarrow$ *s.i := S.t, s.data := d, s.state :=* MSG
†*other[i]* $\rightarrow$ *close[i]*

<u>*s.state =* MSG</u>
* $\rightarrow$ *msg[s.i, s.data]*
ack[s.i] $\rightarrow$ *Ok(), s.state :=* STAY
sync[s.i, j] $\rightarrow$ *s.j := j, s.state :=* VALIDATE
†*other[i]* $\rightarrow$ *close[i]*

<u>*s.state =* VALIDATE</u>
* $\rightarrow$ *valid[s.i, s.j]*
ack[s.i] $\rightarrow$ *Ok(), s.state :=* STAY

<u>*s.state =* STAY</u>
Send(d) $\rightarrow$ *s.i := s.i+1, s.data := d, s.state :=* MSG
Close() $\rightarrow$ *close[s.i], s.state :=* CLOSED
†*sync[i, j]* $\wedge$ $i \neq s.i$ $\rightarrow$ *close[i]*

The connection at the receiver works as follows. Figure 8 shows the state diagram.

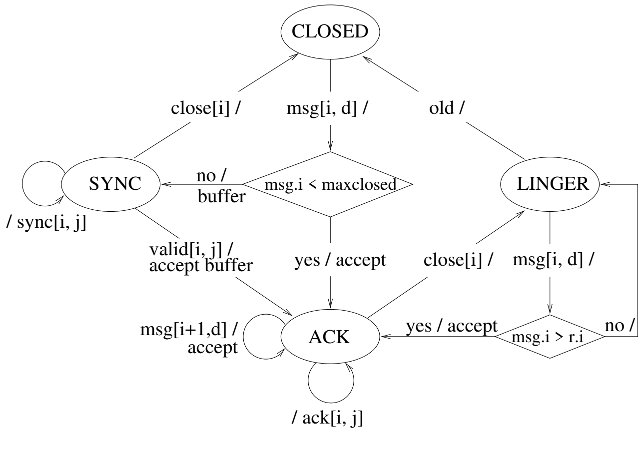


Figure 8: State diagram for Hula receiver.

r.state = CLOSED

$msg[i, d] \wedge i > max\_closed \rightarrow Receive(d),$
$r.i := i,\ r.state :=$ ACK

$msg[i, d] \wedge i \leq max\_closed \rightarrow r.buf = d,\ r.i := i,$
$r.j = R.t,\ r.state :=$ SYNC

$\dagger valid[i, j] \rightarrow close[i]$

r.state = ACK

$* \rightarrow ack[r.i]$

$close[r.i] \rightarrow r.state :=$ LINGER

$msg[r.i{+}1, d] \rightarrow Receive(d),\ r.i := r.i{+}1$

r.state = LINGER

$msg[i, d] \wedge i > r.i \rightarrow Receive(d),\ r.i := i,\ r.state :=$ ACK

$r.i < R.t{-}\delta \rightarrow max\_closed = max(r.i, max\_closed),$
$r.state :=$ CLOSED

r.state = SYNC

$* \rightarrow sync[r.i, r.j]$

$valid[r.i, r.j] \rightarrow Receive(r.buf),\ r.state :=$ ACK

$\dagger close[r.i] \rightarrow r.state :=$ CLOSED

4.2 Remote Procedure Calls

As mentioned, an important use of reliable delivery is in executing remote procedure calls, or RPCs [BN84]. An RPC, say from *S* to *R*, involves a call message from *S* to *R* and a reply message from *R* to *S*. Although, this paper does not discuss duplex transfer using Hula, an isolated RPC can be performed using three packets as shown in Figure 9. The first contains the call message from *S* to *R*, the second contains the reply from *R* to *S* and provides the *ack* for the first, and the third closes the connection and provides the *ack* for the reply from *R* to *S*.

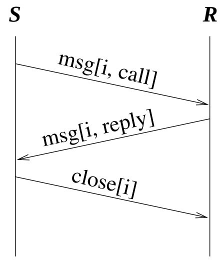


Figure 9: Executing an RPC.

4.3 Crash Recovery

The protocol described so far does not tolerate crashes of the receiver host, *R*. Upon recovery, all connections at *R* are closed and *R* might then accept a message already accepted before the crash. As in the clock protocol, *R* maintains a stable variable, *max_received*, such that $max\_received \geq msg.i$ for any *msg* packet accepted. When *R* receives a message such that $msg.i > max\_received$, it must update *max_received* stably before accepting the message. After a crash, *R* sets *max_closed* to *max_received*. However, this alone is not sufficient to avoid duplicate delivery in Hula because packets timestamped below *max_closed* may be accepted if their validation succeeds. Thus, the scenario shown in Figure 10 is possible. (This is similar to the the scenario in Figure 6.)

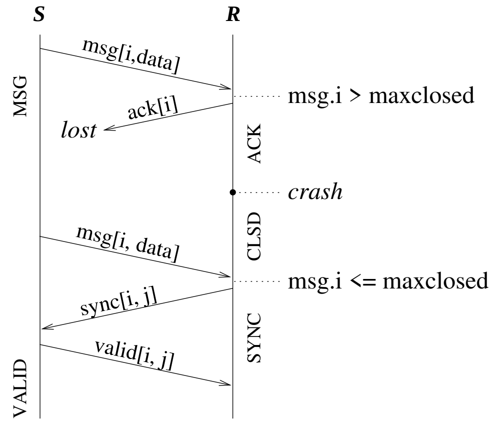


Figure 10: Incorrect execution when receiver crashes.

Hula solves this problem by keeping another variable, *max_crashed*, which is at least as high as the stamp of any packet accepted before the last time the host crashed. By definition, this variable is simply the value of *max_received* at the last crash. After a crash, a host sets both *max_crashed* and *max_closed* to *max_received*. Thereafter, *max_crashed* remains the same, while both *max_closed* and *max_received* advance as needed. Any message stamped below *max_crashed* is not accepted. Instead, *R* returns a *close* packet for such messages. On receiving the close packet, *S* returns a negative acknowledgement to the sender process. Changes to the state machine are shown below.

r.state = CLOSED

$msg[i, d] \wedge i \leq max\_crashed \rightarrow close[i]$

$msg[i, d] \wedge max\_crashed < i \leq max\_closed$
$\rightarrow r.buf = d,\ r.i := i,\ r.j = R.t,\ r.state :=$ SYNC

$msg[i, d] \wedge max\_closed < i \leq max\_received$
$\rightarrow Receive(d),\ r.i := i,\ r.state :=$ ACK

$valid[i, j] \rightarrow close[i]$

s.state = MSG

...

$close[s.i] \rightarrow Error(),\ s.state :=$ CLOSED

s.state = VALIDATE

...

$close[s.i] \rightarrow Error(),\ s.state :=$ CLOSED

In the state machine described above, *R* ignores packets stamped above *max_received*. Since *S* keeps retransmitting the message and *R* increases *max_received* periodically, the message is finally accepted. An obvious optimization here is that *R* can buffer the message and send a packet to the sender to stop retransmitting.

4.4 Comparative Discussion

Figure 11 tabulates the level of reliability provided by various protocols in two cases: when there are no crashes and upon crashes. Belsnes showed that two different four-packet protocols could be constructed with different crash semantics [Bel76]. His five-packet protocol has the desirable property that it is reliable while there is no crash and only loses messages or causes false negative acknowledgement upon a crash. Protocols that close connections after the maximum lifetime of packets might duplicate messages if packets outlive the expected maximum [FW78]. The synchronized clock protocol might lose messages if packets arrive late, or seem to arrive late because of clock skews [LSW91]. Finally, Hula provides the same reliability as the best handshake protocol.

Protocol	**In absence of crash**	**Upon crash**
4-pkt handshake	Reliable & false -ve ack	Loss & false -ve ack
4-pkt handshake	Reliable	Loss & false +ve ack
5-pkt handshake	Reliable	Loss & false -ve ack
max pkt-lifetime	Duplication	Loss & false -ve ack
sync. clock	Loss & false -ve ack	Loss & false -ve ack
Hula	Reliable	Loss & false -ve ack

Figure 11: Reliability of various protocols.

All protocols might lose messages upon crashes. A potential question is whether it is worthwhile selecting a protocol that might fail only upon a crash over another protocol that might fail, additionally, when packets are delayed. The selection is justified for the following reasons. First, crashes can often be prevented using reliable hardware and software at the end hosts, but it is difficult to guarantee reliable network performance because the network is a heavily shared resource. Second, software techniques such as warm reboots [CNC$^+$96], or hardware techniques such as non-volatile RAM [BAD$^+$92] can be used to prevent loss of connection information upon crashes. If such techniques are used, protocols that fail only upon losing connection information will be highly reliable.

Figure 12 tabulates the number of packets exchanged to deliver a single message. The counts shown are achieved in the best case, *i.e.*, when the underlying network does not lose packets or delay them unexpectedly. The packets are categorized as *foreground* packets if they must be sent before the message is delivered to the remote process (including the packet bearing the message), or *background* packets if they are sent later. Foreground packets contribute to the latency of message delivery, while background packets contribute only to resource utilization and can often be piggybacked on other packets. The handshake protocols require three foreground packets, while purely time-based protocols and Hula require only one.

Hula requires only one more background packet than previous time-based protocols. Furthermore, Hula is about as efficient as the time-based protocols for an important class of applications: RPC. As mentioned before, Hula performs an isolated RPC using three packets. Purely time-based protocols must also either employ the third packet, *close*, or buffer the reply message for a conservatively long interval. Therefore, most time-based protocols do use three packets [BN84, LSW91]. If a number of RPCs are executed successively, Hula as well as these protocols use only two packets for each RPC except for the last one.

Protocol	**Foreground packets**	**Background packets**
4-pkt handshake	3	1
5-pkt handshake	3	2
sync. clock	1	1
Hula	1	2

Figure 12: Number of packets sent in the best case.

Hula allows a host to control the number of connections that are kept open in the LINGER state. In particular, a host might reduce δ arbitrarily to close more connections without compromising reliability. Previous time-based protocols would fail increasingly frequently on reducing δ. A host might want to close connections in order to reduce the size of the connection table, say, for efficient access or for keeping the table within a small non-volatile RAM. The performance of Hula degrades gracefully as δ is reduced, since more messages would require a handshake. In fact, when δ is reduced to zero, Hula degenerates to the handshake protocol since the LINGER state is eliminated.

5 Conclusions

The Hula protocol provides reliable delivery of messages over an underlying network that might lose, duplicate, reorder, or delay messages. It retains the best features of handshake-based and time-based protocols and avoids their pitfalls. It has the reliability of handshake-based protocols: messages are delivered exactly once and in the order sent (in the absence of host crashes). Further, its efficiency is close to that of time-based protocols since it avoids handshakes. This is particularly advantageous for sending small intermittent messages, as in RPC and protocols like HTTP, where the overhead of handshakes can be significant. Hula executes a handshake only when a message packet seems to arrive too late—either because of an unexpectedly long packet delay or an unexpectedly large clock skew. Such handshakes ensure that reliability is not compromised when the network or clocks behave unexpectedly. Handshakes also allow hosts to reduce the period for which inactive connections must be kept open. When this period is minimized, Hula degenerates to a purely handshake-based protocol.

These benefits come at the cost of only one more background packet than a purely time-based protocol. Further, even this disadvantage is masked in an important application: executing an RPC, in which case both Hula and purely time-based protocols send three packets.

Below we indicate some important directions for future work on Hula. First, while we have provided informal arguments for the correctness of our protocol, it needs to be proven more formally. (The five-packet handshake protocol and the synchronized clock protocol have been proven to satisfy their specifications [LLSA93].) Second, Hula needs to be extended to incorporate common features of transport protocols such as full-duplex transfer and fragmentation of messages. Finally, a simulation or a real implementation is necessary to measure performance gains from using this protocol.

Acknowledgement

The author is grateful to Butler Lampson for encouraging discussions. David Murphy provided useful comments on an earlier draft of this paper. This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research, contract N00014-91-J-4136.

References

[BAD+92] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 10–22, October 1992.

[Bel76] D. Belsnes. Single message communication. *IEEE Transactions on Communications*, 24(2), Feb. 1976.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), Feb. 1984.

[CNC+96] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The rio file cache: Surviving operating system crashes. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 1996.

[FW78] J. G. Fletcher and R. W. Watson. Mechanisms for a reliable timer-based protocol. In *Proc. Symp. Computer Network Protocols*, Feb. 1978.

[JBB92] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. Network-Working-Group RFC 1323, May 1992.

[LLSA93] B. Lampson, N. Lynch, and J. Søgaard-Andersen. Correctness of at-most once message delivery protocols. In *Proc. Conf. Formal Description Techniques*, Oct. 1993.

[LSW91] B. Liskov, L. Shrira, and J. Wroclawski. Efficient at-most-once messages based on synchronized clocks. *ACM Transactions on Computer Systems*, 9(2):125–142, May 1991.

[Mil88] D. L. Mills. Network time protocol (version 1) specification and implementation. Internet RFC 1059, July 1988.

[Pos81] J. Postel. Transmission control protocol. Internet RFC 793, Sept. 1981.

[Ste76] N. V. Stenning. A data transfer protocol. *Computer Networks*, 1(2):99–110, Sept. 1976.

[Tom75] R. S. Tomlinson. Selecting sequence numbers. *ACM SIGCOM/SIGOPS Interprocess Communications Workshop*, 9(3):11–23, July 1975.

[Wat89] R. W. Watson. The delta-t transport protocol: Features and experience. In *Proc. 14th Conf. Local Computer Networks*, pages 399–407. IEEE, Oct. 1989.