

Reliability

Rafael H. Schloming

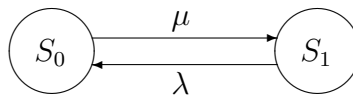
October 19, 2007

1 Network Model

This document examines several techniques for achieving reliable transmission on unreliable networks. To systematically analyze these techniques, we must choose a network model. We assume that the underlying network technology has two distinct failure modes:

1. When the network is functioning normally there is some probability of transmission loss.
2. Additionally there is some probability that the network itself fails, in which case total transmission loss will occur until it is repaired.

To model the failure and repair of networks, we choose a two state continuous time Markov model with a failure rate of λ and a repair rate of μ .



$$S_0 = \text{The network is down.} \quad (1)$$

$$S_1 = \text{The network is up.} \quad (2)$$

The probability that the network is up as a function of time is given by (3):

$$P(S_1) = \left(P_{t_0}(S_1) - \frac{\mu}{\lambda + \mu} \right) e^{-(\lambda + \mu)t} + \frac{\mu}{\lambda + \mu} \quad (3)$$

From this we can also compute the steady state network probabilities by letting $t \rightarrow \infty$:

$$P(\text{steady state up}) = \frac{\mu}{\lambda + \mu} \quad (4)$$

$$P(\text{steady state down}) = 1 - P(\text{steady state up}) \quad (5)$$

$$= \frac{\lambda}{\lambda + \mu} \quad (6)$$

Additionally we let L be the probability of transmission loss under normal operating circumstances.

$$P(\text{nominal loss}) = L \quad (7)$$

It should be noted that L is not actually a constant. On a network with multiple senders, the likelihood of collision increases with added transmission attempts. It is therefore necessary, in practice, to introduce some form of congestion control in order to bound L . For the purpose of this analysis we assume that there is some independent congestion control strategy in place that allows L to be modeled as a constant.

2 Protocol Primitives

The transmission strategies described in this document depend on some or all of the protocol primitives defined here. Each primitive is represented as a distinct class that can be transmitted via the `tx()` method. It is assumed that any such transmit attempts are unreliable. If the attempt succeeds the recipient of the primitive (which may be the sender or receiver of messages) is notified via the `rx(T t)` callback where T is the type of the primitive.

Message

The **Message** primitive has the following fields:

id		The message id.
ttl		The lifespan of the message.
payload		The message payload.

The message id must uniquely identify each message within the set of all possible live messages on the connected network. In practice the connected

network includes the Internet and so the id must be unique within the set of most live messages in the world. In the most general case the message id is assumed to be a UUID.

The `ttl` is the duration for which a sender will attempt to transmit a message. If all transmit attempts fail during this period the sender will give up and the message will be lost. In the general case the `ttl` may be arbitrarily long or even infinite. The longer the `ttl` the more live messages there can be at any given time, and the greater the uniqueness requirement for the message id.

Ack

The **Ack** primitive carries the following field:

id | The id of the message being acknowledged. ¹

When the message sender sees an **Ack** for that message, it knows that at least one of its transmissions has succeeded and it is therefore released from its obligation to retransmit the message.

AckAck

The **AckAck** primitive carries the following field:

id | The id of the message whose **Ack** is being acknowledged. ¹

When the message receiver sees an **AckAck** for that message, it knows that the sender will no longer attempt to retransmit the message, and it is therefore released from its obligation to retain that id in its idempotence barrier. ²

3 One-Way, At-Least-Once

The basis for any reliable transmission mechanism is the repetition of unreliable (or less reliable) transmission attempts. We therefore examine the

¹In practice the **Ack** and **AckAck** primitives often carry a set of ids rather than a single id. For simplicity we omit this optimization.

²On unordered networks the **AckAck** may overtake a prior transmission attempt. On these networks it is necessary to retain the id in the idempotence barrier for longer than the maximum differential delay that can be introduced by the network. The **AckAck** therefore has the effect of reducing the `ttl` to this value.

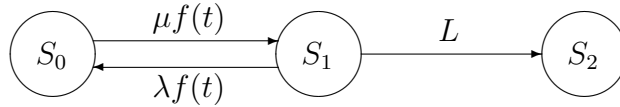
characteristics of a simple sender that periodically transmits each message over the course of the message's lifespan. Such a sender will always make a fixed number of transmission attempts for a given retransmit delay:

$$n = \text{floor}(ttl/\text{delay}) \quad (8)$$

For a message to be lost each one of these attempts must fail. To compute the probability of this we can construct a three state, discrete transition, Markov process to describe our transmission attempts. For brevity we factor out the common expression $f(t)$:

$$f(t) = \frac{1 - e^{-(\lambda+\mu)t}}{\lambda + \mu} \quad (9)$$

We now define the following states and transition probabilities:



$$S_0 = \text{The network is down and no transmission has been seen.} \quad (10)$$

$$S_1 = \text{The network is up and no transmission has been seen.} \quad (11)$$

$$S_2 = \text{The transmission has been seen.} \quad (12)$$

The complete transitional probabilities of this Markov process can be expressed in terms of two transitional probability matrices, a delay matrix $D(t)$ and a transmission matrix T .

$$D(t) = \begin{bmatrix} 1 - \mu f(t) & \lambda f(t) & 0 \\ \mu f(t) & 1 - \lambda f(t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (13)$$

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & L & 0 \\ 0 & 1 - L & 1 \end{bmatrix} \quad (14)$$

Both $D(t)$ and T are left stochastic matrices. $D(t)$ describes the conditional transition probabilities between network outage and repair for a given duration t . T describes the conditional transition probabilities for an isolated transmission attempt. We can now compute the probabilistic distribution of states (P_s) for the Markov process when the sender expires the message.

We start with the steady state network probabilities, and multiply by our conditional transition matrices for each transmission attempt and delay:

$$n = \text{floor}(ttl/\text{delay})$$

$$P_s = (TD(ttl))^{n-1}T \times \begin{bmatrix} P(\text{steady state down}) \\ P(\text{steady state up}) \\ 0 \end{bmatrix} \quad (15)$$

$$= (TD(ttl))^{n-1}T \times \begin{bmatrix} \frac{\lambda}{\lambda+\mu} \\ \frac{\mu}{\lambda+\mu} \\ 0 \end{bmatrix} \quad (16)$$

We can now choose some sample parameter values for our network model and our sender. For the network model we choose an outage rate of once per week, a repair rate of once per hour, and a 10% nominal loss rate. For the sender, we choose a 5 minute timeout and a 3 second retransmit delay — typical values for a TCP connection:

ttl:	300 s	outage:	1 /week
count:	100	repair:	1 /hour
delay:	3 s	loss:	10 %

Based on these values, we can compute the following conditional transition probabilities for our sender. Recall that for a transition matrix, each column and row correspond to a state in the Markov model, and for a left stochastic matrix, each row within a column represents the conditional probability of transitioning to the row state given we start out in the column state.

$$\begin{bmatrix} 0.920812 & 5.07759 \times 10^{-07} & 0 \\ 8.53035 \times 10^{-05} & 4.70385 \times 10^{-11} & 0 \\ 0.079103 & 0.999999 & 1 \end{bmatrix}$$

With these choices it is evident that from S_1 we are quite likely to reach S_2 , but from S_0 it is very improbable. This is not unexpected given that our sender times out at 5 minutes, well short of our one hour average repair rate.

To help us isolate the important characteristics of the sender for this network model, we plot in figure (1) the probability of success as a function of both retransmission *count* and *ttl*. For these plots we choose the same parameters for our network model as above. In figure 1(a) we choose a *ttl* of

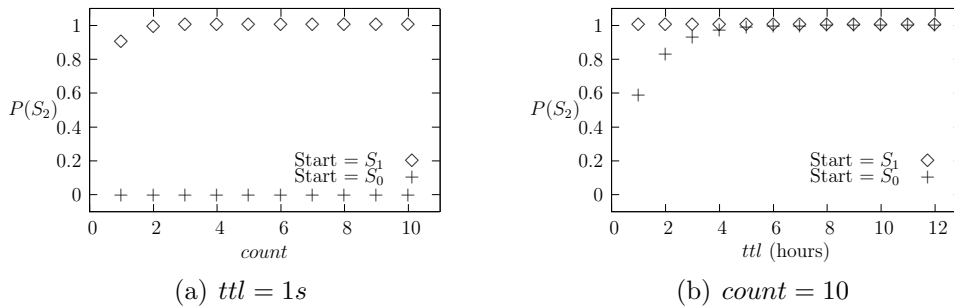


Figure 1: Transitional probabilities vs ttl and count.

1 second and observe the effect of varying the *count* between 1 and 10. In figure 1(b) we choose a *count* of 10 and observe the effect of varying the *ttl* from 1 hour to 12 hours.

From figure 1(a) we can see that if the network is functioning it takes only a few attempts to provide a very high probability of successful transmission, however due to the short *ttl* (1 second), if the network happens to be down, there is almost no chance of success.

From figure 1(b) we can see that even with a small number of attempts (*count* = 10), if we spread these across a span of time in excess of 6 hours, we vastly improve the chances of a network failure preventing our message from being seen.

These results suggest that in addition to repetition, a key aspect of a reliable sender is persistence. Furthermore these two aspects of the sender *independently* address each of the distinct failure modes of the network. Repetition mitigates nominal loss, and persistence mitigates network outage. This means that based on our network characteristics, we can tune our sender to achieve a high a probability of successful transmission using surprisingly few resources, and without taxing the network.

This is as good a guarantee as can be made by any transmission mechanism, however it should be noted that this technique does not provide confirmation of receipt. Even though parameters can be chosen to provide any desired probability of success, the sender never knows for certain whether or not the transmission was actually received.

Obligations

Sender

- The sender MUST retain a message for the duration of its *ttl*.

- The sender MUST transmit the message every *delay* seconds while the *tll* is positive.
- The sender MUST update the message *tll* prior to each transmission attempt.
- The sender MUST NOT transmit a message if the *tll* reaches zero.

Receiver

- The receiver MUST deliver received messages to the application.

Code

Simple Sender

When the application asks for a message to be sent, the sender adds it to the set of live messages:

```
ConcurrentMap<Id,Message> sending =
    new ConcurrentHashMap<Id,Message>();

public void send(Message msg)
{
    sending.put(msg.header.id, msg);
}
```

The sender transmits all live messages every *delay* milliseconds until the message expires:

```
boolean running = true;
while (running)
{
    for (Message msg : sending.values())
    {
        if (msg.header.ttl < 0)
        {
            sending.remove(msg.header.id);
        }
        else if (sending.containsKey(msg.header.id))
        {
            tx(msg);
        }
    }
}
```

```

        msg.header.ttl -= delay;
    }
}
running = sleep(delay);
}

```

Simple Receiver

```

void rx(Message msg)
{
    // deliver message to application
    deliver(msg);
}

```

4 Two-Way, At-Least-Once

In both the one-way at-least-once transmission scheme, the memory requirements for the sender can become large for large message *ttls*. By introducing an acknowledgment from the receiver to the sender, we can allow the sender to know that the message was transmitted successfully, and therefore release the sender from its obligation to retransmit the message. Although this mechanism is likely to require fewer transmission attempts in the successful case, the previously computed probabilities still apply since the sender will still make the same number of transmission attempts over the same span of time before giving up and allowing the overall transmission attempt to fail.

This mechanism can be used purely as an optimization of the one-way scheme, however it also provides the sender the ability to notify the sending application that a message was received. In other words it has an additional semantic property — confirmation of receipt. The sender can know for certain that the message was seen by the receiver.

It should be noted that although there is no explicit attempt to reliably transmit the **Ack**, i.e. it is a simple echo produced when a message is received, the **Ack** is still reliably communicated because if any given **Ack** is lost a subsequent retransmission will prompt another transmission attempt for the **Ack**.³

³The exact guarantee for the **Ack** is actually slightly less than for the **Message** itself since the **Ack** transmission attempts are only made when a **Message** transmission attempt succeeds.

Obligations

In addition to the prior obligations of section 3:

Sender

- The sender **MUST** set the message `ttl` to zero when it sees an **Ack**.

Receiver

- The receiver **MUST** produce an **Ack** for each **Message** it sees.

Code

TODO

5 One-Way, Exactly-Once

The repetition and persistence required for reliable transmission necessarily increases the probability of duplicate transmissions being seen at the receiver. To address this inherent risk of duplication, we introduce an idempotence barrier at the receiver. This barrier serves to prevent multiple deliveries of the same message to the application at the receiver.

We construct the idempotence barrier from the message ids contained in incoming message transmissions. We require the receiver to retain the id of any seen message so that it may discard duplicate transmissions. The duration it must retain any given id can be computed from the message `ttl` and the maximum differential delay that can be introduced by the network:⁴

$$ttl_{id} = ttl_{msg} + MDD \tag{17}$$

In practice the exact value for `MDD` depends on the details of the network topology and lower level protocol in use. In some cases this may be zero or strictly bounded, and in some cases we may need to choose an approximation. If the receiver fulfills this obligation, then the probability of duplicates is bounded by the probability that the differential delay introduced by the network exceeds the chosen `MDD`:

$$P(\text{duplicates}) \leq P(\text{differential delay} > MDD) \tag{18}$$

⁴For an ordered transport `MDD` is zero.

This is the same exactly-once guarantee provided by the two-way and three-way schemes described later in this document. Probability of loss and duplicates can be minimized to any degree desired. However, just as with the at-least-once mechanism described in section 3, there is no confirmation of receipt, so the sender will never know for certain whether the transmission succeeded or not.

An optimized form of this mechanism is the basis for reliable multicast protocols such as PGM. Rather than spontaneously retransmitting, the sender retains the live message and retransmits on demand when prompted by a **Nack**⁵. This optimization depends on the ability to construct a predictable sequence of message-ids so that missing messages can be detected at the receiver.

Obligations

In addition to the prior obligations of section 3:

Receiver

- The receiver **MUST** retain a message id for the duration of the message *tll* plus the maximum differential delay for the network.
- The receiver **MUST** deliver an unseen message to the application.
- The receiver **MUST** ignore a previously seen message.
- The receiver **MUST** ignore a message with zero or negative *tll*.

Code

Idempotent Receiver

By extending the simple receiver with an idempotence barrier we can ignore duplicate messages:

```
ConcurrentMap<Id,Header> seen =
    new ConcurrentHashMap<Id,Header>();

void rx(Message msg)
{
```

⁵A **Nack** is a protocol primitive used by the receiver to request transmission of a given message from the sender.

```

    Header old = seen.putIfAbsent(msg.header.id, msg.header);
    if (old == null)
    {
        super.rx(msg);
    }
}

```

The receiver periodically garbage collects the set of seen ids:

```

void gc(Header hdr)
{
    if (hdr.ttl < -MSL)
    {
        seen.remove(hdr.id);
    }
    else
    {
        hdr.ttl -= delay;
    }
}

```

6 Two-Way, Exactly-Once

By combining the two-way at-least-once receiver of section 4 and the idempotent receiver of section 5, we can create a two-way exactly-once mechanism.

Obligations

The obligations of the sender and receiver are simply the union of the obligations from section 4 and 5.

Code

Two-Way Receiver

The two-way receiver transmits an **Ack** each time it receives a **Message**.

```

void rx(Message msg)
{

```

```

    super.rx(msg);
    tx(new Ack(msg.header.id));
}

```

Two-Way Sender

On receipt of an **Ack** the two-way sender removes the identified **Message** from the set of live messages. The sender also has the option to confirm receipt of the message to the sending application.

```

void rx(Ack ack)
{
    sending.remove(ack.id);
    // Sender has the option to notify the app that the
    // message was definitely received.
}

```

7 Three-Way, Exactly-Once

In both the one-way and two-way exactly-once schemes the receiver has an obligation to retain seen message ids for the full duration of $t_{tl} + MDD$. By introducing an **AckAck** from the sender to the receiver, the sender can inform the receiver that it has ceased transmission of the given message. This permits the receiver to zero the t_{tl} for that message and garbage collect its id early.⁶

Once again for a message to be lost all transmission attempts over the full duration of the t_{tl} must fail, so despite the early release of the sender and receiver's obligations, the probabilities remain the same.

This mechanism may be used purely as an optimization to the two-way scheme to permit larger timeouts, however it has two additional properties that are important.

1. This is the only exactly-once mechanism that permits infinite message timeouts.
2. Just as the two-way mechanism adds confirmation of receipt, the three-way mechanism adds another semantic property — confirmation of

⁶The id must still be retained for MDD after receiving the **AckAck** since a prior transmission attempt might overtake the **AckAck**.

closure. The receiver can know for certain that the sending application has ceased transmission of a message.⁷

It should be noted that the sender's obligation to produce an **AckAck** never ends. This only works because the sender can fulfill that obligation based on the absence of information, and so it requires no ongoing resources to be maintained in order to respond to an **Ack**. In other words if you receive an **Ack** for a message you know nothing about, you can assume that you have already retired the given message.

Obligations

In addition to the prior obligations of section 6:

Sender

- The sender **MUST** transmit one **AckAck** for each **Ack** transmission received.

Receiver

- The receiver **MUST** set the *tll* to *MDD* for each **AckAck** transmission received.

Code

Three-Way Sender

The three-way sender extends the two-way by responding to each **Ack** with an **AckAck** after ceasing to transmit the identified message.

```
void rx(Ack ack)
{
    super.rx(ack);
    tx(new AckAck(ack.id));
}
```

⁷It may not be obvious why this property is useful. For an example, see section 10

Three-Way Receiver

The three-way receiver extends the two-way receiver by setting the ttl to zero for the specified message id upon receipt of an **AckAck**.⁸

```
void rx(AckAck ackack)
{
    // On ordered transports we could remove this right away
    // since there is zero probability of overtake. We could
    // also probably compute a lower bound for this than the
    // MSL.
    Header hdr = seen.get(ackack.id);
    if (hdr.ttl > 0)
    {
        hdr.ttl = 0;
    }
}
```

In addition, the three-way receiver sends an **Ack** each time it attempts to garbage collect an id:

```
void gc(Header hdr)
{
    super.gc(hdr);
    if (seen.containsKey(hdr.id))
    {
        tx(new Ack(hdr.id));
    }
}
```

8 Layering of Reliability

As we have seen, reliable transmission protocols depend on repetition by the sender, and idempotence at the receiver. The receiver's idempotence in turn depends on retaining the message id for the duration of the message

⁸On ordered transports the id can be cleared from the seen set immediately since there is zero probability of overtake. For unordered transports the id must be kept for a period of time equal to the maximum possible differential the overall network can introduce between two sequential transmissions.

t_{tl}, or until an **AckAck** is received. Because of this, two problems can arise when layering one reliable protocol over another. If the higher layer repeats a transmission attempt when the lower layer times out without confirmation of receipt, then the receiver's idempotence barrier may fail for two reasons:

1. The receiver's idempotence barrier will no longer contain the given message id after the lower layer's message *t_{tl}* has expired.
2. The lower layer may generate a new message id for each of the upper layer's transmission attempts.

Consequently the receiver for the upper layer must maintain its own idempotence barrier that retains the message id for the full duration that the upper layer's sender will attempt transmission of the given message.

An example of this problem occurs with SMTP layered over TCP.⁹ If the TCP connection times out prior to the SMTP layer's OK, the application will resend using a new TCP connection. In this case the sending application has improved the at-least-once characteristics of the transmission beyond what TCP provides by layering its own repetition. However the upper layer has no idempotence barrier at the receiver, and so the exactly-once guarantee is lost and duplicate messages may be seen by the receiver's application.

What can be concluded from this is that in general the exactly-once property of a reliable protocol does not layer. In any given stack of protocols, each layer that provides repetition at the sender, must also provide an equally strong guarantee of idempotence at the receiver.

9 Sequential Naming

We can optimize any of the schemes presented here by choosing our message ids such that id sets can be compactly represented. This is typically done by choosing a two part name for a message consisting of a named context for an ordered sequence of messages¹⁰, and a sequence-id within that context:

context-id		A unique name for an ordered sequence of messages.
sequence-id		The message number within the sequence.

Given this choice of naming, the set of seen ids at the receiver can be represented as a single gap encoded set of numbers for each context-id, along with the maximum t_{tl} for any seen message. If the **AckAck** is only being

⁹See <http://tools.ietf.org/rfc/rfc1047.txt> for a detailed description of the problem.

¹⁰Typically referred to as a connection, session, or stream.

used for optimization, and not semantic reasons, this permits us to defer the **AckAck** until a context is inactive or retired.¹¹

Additionally, if a set of ids rather than a single id is used for both the **Ack** and **AckAck** primitives, a single primitive can be used to transmit many logical **Acks** or **AckAcks** in a single compact physical structure.

If a single context is being used for full duplex communication, the overhead can be compressed even further by permitting a peer's receiver to include the id sets for the **Ack** and **AckAck** primitives with the outgoing message traffic generated by the peer's sender. This permits the **Ack** and **AckAck** primitives to share the same context-id carried by the **Message**.

10 Unconstrained Naming

The sequential naming optimization described in section 9 can be used to provide extremely high performance implementations that provide expedient confirmation of receipt for the sender and closure for the receiver. However, this optimization cannot be directly applied to applications that require free choice of message-id such as UUIDs.

For these applications we consider what happens when we choose a message id consisting of two fully unique names, one provided by the application, and one provided by a sequential message transport as described in section 9:

message-id		The application provided message name.
context-id		A unique name for an ordered sequence of messages.
sequence-id		The message number within the sequence.

The **message-id** and the pair (**context-id**, **sequence-id**) are each unique names for the same message. In other words (**context-id**, **sequence-id**) is an alias for **message-id**. This means we can refer to any given message by either the **message-id**, or the pair (**context-id**, **sequence-id**), or both. We now make the following choices:

- For the **Message** primitive we choose to include both names.
- At the receiver we choose to maintain two idempotence barriers, one using the sequential message names, and one using the application provided names.
- For the **Ack** and **AckAck** primitives on the wire, we choose to use only the set representation of the (**context-id**, **sequence-id**) names.

¹¹The session, connection, or stream is closed.

With these choices, we can now move the application defined idempotence barrier out of the transport layer and into the application layer. So long as transport layer can provide the application layer with expedient confirmation of closure for incoming messages, the application layer can maintain a minimally sized idempotence barrier even for the most general naming schemes such as UUID based message ids.¹²

This allows the application layer to, under normal operation, entirely omit the expensive **Ack** and **AckAck** primitives required for reliable transmission with a generalized naming scheme. The application layer need only employ these primitives if the sequential transport should fail while there are still message ids in application's idempotence barrier.

11 Notes

- Reliability against nominal packet loss is cheap and easy.
- Reliability against network outage requires a lengthy commitment of resources from both the sender and receiver. This requires authentication and authorization, and must therefore be modeled as part of the top level application design.
- Layering of reliability has limited utility. Strong upper layers will redo the work of lower layers, weak upper layers will waste the effort of lower layers. The net guarantee is always exactly what is provided by the top layer of the stack.
- Congestion control is hard. It's still worth layering over TCP and/or SCTP just to get congestion control.
- The top level reliability layer directly modeled in AMQP is the dialog between an application and a Queue.

12 Complete Code

```
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.ConcurrentHashMap;

abstract class Reliability
{
    static final long DELAY = 20;
```

¹²The term application is used in the general sense of any client of this layer. In practice it may be another reliable protocol layer such as a persistent message service, or a top level application.

```

static final long MSL = 60*1000;

static boolean sleep(long duration)
{
    try
    {
        Thread.sleep(duration);
    }
    catch (InterruptedException e)
    {
        return false;
    }

    return true;
}

interface Id {}

interface Transmittable {}

class Header
{
    Id id;
    volatile long ttl;

    Header(Id id, long ttl)
    {
        this.id = id;
        this.ttl = ttl;
    }
}

class Message implements Transmittable
{
    Header header;
    byte[] payload;

    Message(Id id, long ttl, byte[] payload)
    {
        this.header = new Header(id, ttl);
        this.payload = payload;
    }
}

class Ack implements Transmittable
{
    Id id;

    Ack(Id id)
    {
        this.id = id;
    }
}

class AckAck implements Transmittable
{
    Id id;

    AckAck(Id id)
    {
        this.id = id;
    }
}

```

```

}

abstract void tx(Transmittable t);

class SimpleSender implements Runnable
{
    ConcurrentMap<Id,Message> sending =
        new ConcurrentHashMap<Id,Message>();
    long delay = DELAY;

    public void send(Message msg)
    {
        sending.put(msg.header.id, msg);
    }

    public void run()
    {
        boolean running = true;
        while (running)
        {
            for (Message msg : sending.values())
            {
                if (msg.header.ttl < 0)
                {
                    sending.remove(msg.header.id);
                }
                else if (sending.containsKey(msg.header.id))
                {
                    tx(msg);
                    msg.header.ttl -= delay;
                }
            }

            running = sleep(delay);
        }
    }
}

abstract class SimpleReceiver
{
    void rx(Message msg)
    {
        // deliver message to application
        deliver(msg);
    }

    abstract void deliver(Message msg);
}

abstract class IdempotentReceiver extends SimpleReceiver
implements Runnable
{
    ConcurrentMap<Id,Header> seen =
        new ConcurrentHashMap<Id,Header>();
    long delay = DELAY;

    void rx(Message msg)
    {
        Header old = seen.putIfAbsent(msg.header.id, msg.header);
        if (old == null)
        {

```

```

        super.rx(msg);
    }
}

public void run()
{
    boolean running = true;
    while (running)
    {
        for (Header hdr : seen.values())
        {
            gc(hdr);
        }

        running = sleep(delay);
    }
}

void gc(Header hdr)
{
    if (hdr.ttl < -MSL)
    {
        seen.remove(hdr.id);
    }
    else
    {
        hdr.ttl -= delay;
    }
}
}

class TwoWaySender extends SimpleSender
{
    void rx(Ack ack)
    {
        sending.remove(ack.id);
        // Sender has the option to notify the app that the
        // message was definitely received.
    }
}

abstract class TwoWayReceiver extends IdempotentReceiver
{
    void rx(Message msg)
    {
        super.rx(msg);
        tx(new Ack(msg.header.id));
    }
}

class ThreeWaySender extends TwoWaySender
{
    void rx(Ack ack)
    {
        super.rx(ack);
        tx(new AckAck(ack.id));
    }
}

abstract class ThreeWayReceiver extends TwoWayReceiver
{

```

```

void gc(Header hdr)
{
    super.gc(hdr);

    if (seen.containsKey(hdr.id))
    {
        tx(new Ack(hdr.id));
    }
}

void rx(AckAck ackack)
{
    // On ordered transports we could remove this right away
    // since there is zero probability of overtake. We could
    // also probably compute a lower bound for this than the
    // MSL.
    Header hdr = seen.get(ackack.id);
    if (hdr.ttl > 0)
    {
        hdr.ttl = 0;
    }
}
}
}
}

```

13 Simulation

```

import java.util.Random;

class Simulation extends Reliability
{
    ThreeWaySender sender = new ThreeWaySender();
    ThreeWayReceiver receiver = new ThreeWayReceiver() {
        void deliver(Message msg)
        {
            System.out.println("message received: " + new String(msg.payload));
        }
    };

    void rx(AckAck ackack)
    {
        System.out.println("id cleared: " + ackack.id);
        synchronized (Simulation.this)
        {
            Simulation.this.notifyAll();
        }
    }
};

private Random random = new Random();
private float loss;

Simulation(float loss)
{
    this.loss = loss;
}

void start()
{
    Thread t = new Thread(sender);
    t.setDaemon(true);
}

```

```

        t.start();
        t = new Thread(receiver);
        t.setDaemon(true);
        t.start();
    }

    void tx(Transmittable t)
    {
        System.out.println("tx attempt: " + t);
        if (random.nextFloat() < loss)
        {
            return;
        }

        if (t instanceof Message)
        {
            receiver.rx((Message) t);
        }
        else if (t instanceof Ack)
        {
            sender.rx((Ack) t);
        }
        else if (t instanceof AckAck)
        {
            receiver.rx((AckAck) t);
        }
    }

    public static final void main(String[] args)
        throws InterruptedException
    {
        Simulation sim = new Simulation(Float.parseFloat(args[0]));
        sim.start();

        sim.sender.send(sim.new Message(new Id() {}, 60*1000,
            "this is a test".getBytes()));

        synchronized (sim)
        {
            sim.wait();
        }
    }
}

```