An Idempotent Message Protocol
Jeremy Brown

Abstract
We describe a protocol for reliable (exactly-once) data-
gram delivery on a wormhole-routed network that dis-
cards messages in response to congestion and hardware
faults. Our protocol is connectionless – no state needs to
be stored for communicating pairs of processes, only for
datagrams still being managed. Datagrams are not guar-
anteed to be delivered in any particular order.
The protocol is simple enough to be implemented di-
rectly in hardware.

1 Introduction

Our protocol is aimed at a massively parallel computer
architecture with an integral, unreliable inter-node net-
work. Such a system has several characteristics which
have motivated our protocol’s design:

Thousands of nodes, millions of wires: By “massively
parallel computer” we mean a computer that can scale to,
at the very least, several thousand nodes; correspondingly,
there will be tens, hundreds, or even thousands of thou-
sands of wires in the inter-node communications network.
Guaranteeing the reliability of the processing elements
will be a Herculean task, and is the subject of ongoing
research both within our research group and elsewhere;
guaranteeing the reliability of every network wire will be
an impossible task.

An unreliable network with three guarantees: Network
protocols designed to guarantee deadlock-free data-
gram (message) delivery on reliable networks are already

Quite complex, and thus impose a burden of complexity
on the implementation of the routing elements. Combined
with the problem of unreliable wires, the overall complex-
ity of guaranteeing reliable datagram delivery on a net-
work for a massively parallel machine – especially with
good performance (i.e. low latency) – is a daunting task.

Preferring simplicity, we are instead considering net-
work implementations which do not guarantee the deliv-
ery of each message; some mechanism is thus required
to provide reliable datagram delivery on the substrate of
unreliable messages, and this is the niche our protocol is
designed to fill.

Our protocol does depend upon three assumed guaran-
tees of the network:

1. A message, if delivered, is delivered at most once.
2. There exists at least one path with no bad wires be-
tween each pair of processors.
3. All messages which are successfully delivered from
   a node A to a node B are delivered to B in the order
   in which they were sent from A.

We believe these guarantees are fairly easy to provide
in the context of a dedicated network. For example, the
first guarantee is provided by a wormhole-routed network
which resolves congestion simply by discarding messages
randomly. (The implementation of extremely fast router
components for such a network is fairly straightforward.)
The second and third guarantees are ensured, in this exam-
ple, by constraining the network topology. In particular,
the network provides the second guarantee (to a desired
degree of certainty based on the probability of individual
wire failures, etc.) by featuring a topology with multiple,
randomly-selected paths between each pair of processors,
e.g. a FAT tree. The third guarantee arises naturally as
long as all paths between a given pair of processors in
the network are of the same length, as they would be in a FAT
tree.

Since our envisioned target network is, in fact, exactly
this network — a FAT tree featuring wormhole-routing
and discard-based congestion resolution — we shall men-
tion one more useful property: because the distance be-
tween any pair of processors is fixed (and easily com-
puted), it is relatively simple to decide when to give up on
Table 1: The three messages of the reliable-datagram protocol.

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Content</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(D)</td>
<td>&lt; UID&lt;sub&gt;d&lt;/sub&gt;, CONTENT&lt;sub&gt;d&lt;/sub&gt; &gt;</td>
<td>NI&lt;sub&gt;A&lt;/sub&gt; → NI&lt;sub&gt;B&lt;/sub&gt;</td>
</tr>
<tr>
<td>Acknowledge(D)</td>
<td>&lt; UID&lt;sub&gt;d&lt;/sub&gt; &gt;</td>
<td>NI&lt;sub&gt;A&lt;/sub&gt; ← NI&lt;sub&gt;B&lt;/sub&gt;</td>
</tr>
<tr>
<td>Forget(D)</td>
<td>&lt; UID&lt;sub&gt;d&lt;/sub&gt; &gt;</td>
<td>NI&lt;sub&gt;A&lt;/sub&gt; → NI&lt;sub&gt;B&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

receiving a response to a previously-sent message. This will be useful in setting timeout values.

**Unordered datagrams, not streams:** Communications between processing nodes will generally be in the form of datagrams (messages), due to either explicit message-passing or references to remote, shared memory. Each processor may, in short order, send datagrams to a great many other processors.

Additionally, multiple datagrams to a single processor may not have an ordering requirement; for instance, a series of memory operations to independent locations need not be ordered under a weak memory model such as release consistency.

Overall, the semantics of streaming communications – reliable, in-order delivery of a continuous linear sequence of data – are unnecessary for this type of machine, and would add significant complexity to any protocol for reliable inter-node communications.

## 2 Protocol

We target a conceptual architecture in which each processor P has a network interface NI<sub>P</sub> which is responsible for sending and receiving datagrams; it is the responsibility of the NI to implement the reliable-delivery protocol on top of the network’s unreliable messages. Obviously the NI might well be implemented as nothing more than a process running on the processor itself, but for conceptual clarity we will refer to it as a separate component in the following discussion.

### 2.1 Overview

We assign each datagram D an ID so that the datagram is actually a tuple < UID<sub>D</sub>, CONTENT<sub>D</sub> >: UID<sub>D</sub> is an identifier guaranteed to be unique for all datagrams currently in any stage of being sent, and CONTENT<sub>D</sub> is the body of the datagram intended to be delivered to node B. A simple means of generating per-datagram UIDs is for the source node to prepend its node-ID to the value of a counter which is then incremented; the counter need only have enough bits to provide unique IDs for all live datagrams sent from that node.

In order to reliably deliver datagram D from processor A to processor B, a minimum of three messages must be sent between their NIs, and any of these messages may have to be sent more than once in the event that the network discards some due to congestion or bad wires. The messages are tabulated in Table 1.

An informal description of the effects of each message follows:

- **NI<sub>A</sub>** must repeatedly send the Send(D) message at intervals until it has received an Acknowledge(D) from NI<sub>B</sub>, at which time NI<sub>A</sub> must send a Forget(D) message to NI<sub>B</sub> and may forget all about D. This is safe, as NI<sub>B</sub> can not Ack(D) until it has successfully received D. If NI<sub>A</sub> receives any further acks for D from NI<sub>B</sub>, they must also be replied to with Forget(D) messages.

- On the other side of the network, when NI<sub>B</sub> first receives a send from NI<sub>A</sub>, it must deliver D to processor B, then must commence repeatedly sending Ack(D) messages back to NI<sub>A</sub>. Because NI<sub>A</sub> might send additional SendD messages if it has not yet received an Ack(D) message, NI<sub>B</sub> must remember UID<sub>D</sub> so that it will not accidentally re-deliver D to processor B.

- Because we require of our network that messages are delivered in send-order or dropped – never delivered out-of-order – NI<sub>B</sub> can be certain that when it sees a Forget(D) message from NI<sub>A</sub>, it will never again receive a Send(D) message from NI<sub>A</sub>; thus, NI<sub>B</sub> must stop sending Ack(D) messages and may forget all about D when it finally receives a Forget(D) from NI<sub>A</sub>.