A Multicast Transport Protocol

J. Crowcroft
K. Paliwoda

Department of Computer Science
University College London, Gower Street, London, UK.

ABSTRACT

This paper presents the design of a reliable multicast transport protocol. The aim of the protocol is to provide a service equivalent to a sequence of reliable sequential unicasts between a client and a number of servers, whilst using the broadcast nature of some networks to reduce both the number of packets transmitted and the overall time needed to collect replies.

The service interface of the protocol offers several types of service, ranging from the collection of a single reply from any one of a set of servers to the collection of all replies from all known servers. The messages may be of effectively arbitrary size, and the number of servers may be quite large. To support this service over real networks, special flow control mechanisms are used to avoid multiple replies overrunning the client. Reliable delivery is ensured using timeouts and a distributed acknowledgement scheme. The protocol is implemented over a network layer which supports multicast destination addressing and packet delivery. The behaviour of the protocol over both LANs and LANs interconnected by WAN lines is discussed. We also include some notions for possible future support from network interface hardware.

Keywords: Distributed Systems, Multicast, Transport Protocols

1. Introduction

This protocol supports a sequence of exchanges of arbitrarily sized request and response messages between a client and a large number of servers.[1] It is intended to replace existing protocols which employ either sequential unicasts or broadcast. One of the most important uses is to support replicated procedure calls[2,3] but it would also be appropriate as an underlying communications layer for replicated database access, for dissemination of information such as routing tables or electronic mail, or for location of services such as nameservers or gateways.[4]

The aim of this paper is solely to present and evaluate the design of the protocol. We do not discuss, in any detail, the applications that might use multicast, nor do we discuss the management of multicast groups at the network or host level.

The first section of the paper examines the range of multicast semantics required by distributed applications to be supported by the protocol. The next section describes the underlying network layer, together with its support for multicast destination addressing and packet delivery.

The next two sections present the messaging service interface and operation. This sublayer of the protocol is designed to ensure reliable delivery of multicasts using a distributed acknowledgement scheme and retransmission after timeouts. It also provides multiple source and destination flow control using a coupled window scheme.

The next section describes the request and response sublayer, and the use of *up calls* to provide *Voting* on replies.

In the last two sections, we discuss a pilot implementation of the protocol and present some conclusions about the operation of the protocol.

The annexes include a formal analysis of the effect of many reply packets to a multicast request, the state required for the protocol and the packet formats used in the pilot implementation. We also discuss what kind of hardware support might make the protocol implementation simpler.

2. Multicast Semantics

A range of multicast semantics and how they relate to the application requirements have been discussed in the literature.[5,6,7]

Here, we are only concerned with the transport service semantics. We assume that group membership is managed by some other mechanism, and that it can change during a single exchange. However, we allow the user of this multicast protocol to state the group membership, and *lock*

onto it for the duration of each single exchange. By this we mean that once a server is replying, the group management mechanisms will not allow it to leave the group, or that if it does, a failure will be reported by this protocol, since the server process involved will no longer be correctly addressable. This will involve interaction between the application, the protocol and the group management mechanisms.

This protocol is geared towards client-server type applications, in which a client requests a service from a group of which it is not a member. However, it is possible to use the protocol in peer-to-peer type applications, where members of a group communicate amongst themselves. In this case, each member will also receive its own multicast requests.

The protocol supports a variety of *types of service*, ranging from the collection of a single reply from any of a set of (possibly identical) servers, to the collection of all replies from all known servers. On receiving the initial part of a reply, the protocol will lock onto this given server, and may cancel the request to other servers.

When more than one, or all the replies are required, the protocol will periodically retransmit the request until it has received all the replies. Failure modes are exactly analogous to those of a sequence of reliable unicasts, one to each server.

If a multicast request only requires replies from a subset of the group addressed, and this subset is indicated by the user (either by size, or by a list of specific addresses) then the client will lock onto servers as they reply. The protocol will inform non-members of the subset that their replies are not required. This is necessary for performance reasons, since dynamic regrouping into sub-groups could involve to many interactions with the group management system.

2.1 N-Reliable Maximal Service

The protocol presented here is intended to support *n-reliable* multicast requests and replies, where *n* denotes the number of servers guaranteed to see the request. In most cases this will correspond with the number of replies required. The client may specify *how many* and *which members* of a group it is interested in.

How many may be wild-carded meaning *all current members*.

Which members is a list of zero or more normal unicast addresses, indicating a subset of the multicast group of servers.

Many applications require only *1-reliable* multicast. These are usually *location* type services, where the exchange is of the form:

— Multicast Req: Where is Service X?

— N Unicast Replies: Service X is at Y.

Any reliable transport protocol may be enhanced to support this by simply delaying binding the destination address to a particular member of a multicast group, until some satisfactory part of the reply message or connection setup (if stream oriented) has been received from any or a particular member. (This may however have consequences for the rate at which sequence numbers can be recycled).

For applications requiring n-reliable multicast, where n is more than one, the user may also specify a *voting function*, which is *up-called* from the receiving code when one reply has been received from each of the (possibly individually specified) servers.

An example of an application requiring this service might be failure recovery in a replicated system, where voting on the replies would be used for a re-build request - the vote function might be based on timestamps.

So if the user specifies 1 out of any of the servers, with no voting, we have the weakest semantics. If the user specifies n, "out of all", and no voting, we have the equivalent of sequential unicast requests (with perhaps less chance of the group changing). If the user specifies voting, the protocol will return some single reply, or subset of replies based on the user supplied voting function.

In this sense, as far as the receipt and *reaction* to a multicast request is concerned, we may call the protocol "At most once - At least all".

As far as the successful operation of the protocol is concerned in packet delivery, we may call the protocol "At least once - At most all".

3. Underlying Network Service

We assume that the underlying network service has three characteristics:

It provides a datagram service such as the ISO connectionless network service or the DoD Internet Datagram Protocol.

Host multicast or *host group* addressing is supported, and where possible, a packet addressed to a group address will use a physical broadcast or multicast facility on the network (e.g. on broadcast LANs or Satellite networks).

It is possible, by some out of band mechanism, to ascertain the individual addresses of exactly all the members of a group, if required. This may itself be an application using some form of this multicast protocol.

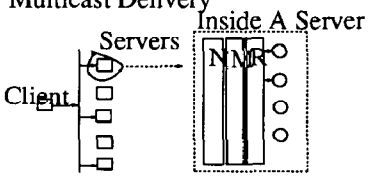
The assumption of a physical broadcast facility obviously does not hold for most WANs and some non-broadcast LANs. For these types of networks, Deering[8] suggests a collection of *multicast agents* to forward multicast packets over point-to-point links, with no change to the best attempt datagram protocol provided by the Internet.

The agents for WAN multidestination delivery will usually form some spanning tree to route the packets.[9] The client is unlikely to be at the root of such a tree, in which case the route that reply packets follow has more bottlenecks than just that at the client. (There may be some part of the network that is far slower than any actual client host). The operation of an adaptive retransmission timer and the aggregate worst case window scheme in the transport protocol will alleviate the problem of packets imploding at such a bottleneck.

The reliable delivery guaranteed by some connection oriented (e.g. X.25) networks is not necessarily an advantage to our multicast transport: When the multicast group is large, it is possible that the agents, responsible for multidestination packet delivery at the network level, may not support adequate network level connections to reach all destinations at once. Rather than enhance them to buffer all the packets, and round robin deliver through connections as they become available, it may be wiser to leave this functionality in the transport level.[10]

In this way we see that the transport should be similar whatever the underlying service, and there is little point in adding more than a multidestination addressing and routing capability to any underlying network service.

Picture 1.
Multicast Delivery



N = Network
M = Message
R = Request/Response

4. The Messaging Service Interface

Given that a datagram service is not reliable, we must introduce some acknowledgement, timeout and retransmission scheme to ensure delivery of a message to the required number of destinations. message.[11,1,12] We must also deal with the fact that large messages have to be broken into segments, which may then be lost, duplicated and misordered. These problems are solved by the messaging service and are offered as facilities by the service interface at this level.

The messaging service is in effect a simplex data protocol, which is then used by another layer to associate a request with a number of responses.

4.1 Message Primitives

There are two varieties of send primitive, and two varieties of receive primitive. All message primitives may take an arbitrary size buffer of data, and reliably deliver it once (or indicate failure) to each destination. There is a mechanism to abort the sending of any message, which results in an error report being sent to the receiver.

1. CMSend(Mid, Buf)

   This is for a user to send a multicast message (Buf) to a group identified by Mid, a multicast identifier.

   When the client calls CMSend, it is effectively blocked until everything has been sent and acknowledged (except for the last windows worth of packets in the message: see section 5.1).

2. CMRecv(Mid, IdList, BufList, VoteFn, Timeout)

   This is for a user to receive a list of replies (BufList) from a list of possibly specified members of the group (Mid, IdList). IdList may be a wild-card to mean *any n*. VoteFn is the (optionally) user supplied function to vote on received messages.

   When the client calls CMRecv it is blocked again until all the packets from *all specified servers* have arrived, or the timeout expires.

3. SMSend(Id, Buf)

   This is intended for a server to reply and looks like a simple reliable unicast message.

4. SMRecv(Id, Buf, Timeout)

   This is for the server to receive a request.

5. Abort(IdList)

   This cancels any outstanding messages to the list of receivers.

Applications can choose the weakest semantics they can possibly use, to achieve the maximum performance.

4.2 Buffering and Markers in the Interface

The buffer parameters to the service interface are not contiguous bytes in memory. Buffers are linked lists of descriptors which hold pointers to sections of contiguous memory, together with the length of that section of memory.

This avoids the need for a presentation layer type conversion to place the network format values in a contiguous buffer. Where the host presentation format is the same as the network format, or of the same size, this means that no buffer copying is required at all. In the latter case, type conversions may be done in place where necessary.

Each message carries an end of message marker in the last packet, and the protocol guarantees that boundaries indicated by the sender's marks will be preserved in calls to the receiver routines. Therefore the (possibly automatically generated) presentation layer does not have to check for application object boundaries overlapping buffer boundaries.

4.3 Failure Modes

CMSend can fail if the number of retransmissions of any part of the multicast message exceeds some threshold. CMRecv can fail if the requested number of replies are not forthcoming inside the timeout. SMSend and SMRecv fail in similar ways.

When a multipacket multicast transport protocol operates over an internet, it is susceptible to more complex failure modes of the underlying network. The protocol should make use of network reachability information, especially if this changes during a CMSend or CMRecv. If a message is received from any agent or router that a requested member, or more than an acceptable number of requested members, are not reachable, the protocol pre-

empts the timeout and reports failure to the user.

5. Messaging Protocol Operation

CMSend and SMSend cut large messages into packets for transmission over the network. CMSend uses the multicast network service to transmit packets. SMSend is essentially a unicast version of CMSend with simpler acknowledgement and retransmission schemes.

Both choose a packet size appropriate to the worst case network route (from a routing table if possible). All packets carry a packet sequence number, and this is used by the receivers to filter duplicates, to sort misordered packets and to detect missing packets. The last packet carries a flag to indicate the end of a message.

5.1 Acknowledgement Scheme for Multipacket Messages

The acknowledgement scheme used at the message level depends on the number of transmitters and receivers.

Acknowledgements of a multicast message are unicast by *each* of the receivers to the transmitter, rather than multicast to the group, to minimise the number of packets on an internet and minimise host processing on a local area network.

Acknowledgements also carry the receivers' advertised window. The client generally adapts its transmission rate to smallest window (see section 5.4). However, lagging servers may selectively acknowledge out of sequence packets, for the client to unicast missing packets.

Since the higher layer of the protocol may send a reply, the protocol will piggyback acknowledgements to the last windows worth of packets in a message on the next message in the opposite direction. A timeout operates to trigger an *explicit* acknowledgement if a further message is not generated soon enough.

At one point we considered using a rotating primary receiver scheme based on a design by Chang and Maxemchuk.[13] This involves designating a special server (the primary), which multicasts acknowledgements to the current request (the primary moves to another server for the next packet). If a secondary server sees the multicast acknowledgement for a packet sequence number greater than the last one it received, it can trigger a negative acknowledgement to the client, who then (unicast) retransmits the missing packets.

We now think this is inappropriate on LANs for the following reason: Since this protocol supports the exchange of large messages, the number of multicast acknowledgements would be large, causing more work for the group. This is very undesirable when the underlying multicast delivery in the network is not based on broadcast (i.e. agents in the internet).

The window and selective retransmission scheme operated by the client part of this protocol have the same functional effect without the overhead.

5.2 Retransmission Schemes

Retransmissions are necessary if a packet remains unacknowledged for a set period of time.

CMSend uses a dynamic retransmission timer in the same fashion as TCP.[14] This timer is set as the upper bound of the round trip times estimated for each destination in the multicast group.

When the retransmission timer fires in CMSend, the client calculates the proportion of hosts which have not yet acknowledged the outstanding packets.

If this proportion is beyond some threshold, CMSend *multicasts* the retransmission. Otherwise we can optimise the retransmission scheme by unicasting the packet only to those hosts which did not send acknowledgements.

The threshold is set depending on the distance and number of the servers, lower for LANs and higher for internet use. This is done to minimise timer events in the client and packets on a local network.

The last packet of a message is retransmitted periodically to check for client/server liveness with a bit set to indicate that this is deliberate.

SMSend uses a timer similar to that in CMSend, but based on the single server-client path.

5.3 Duplicate Filtering

Each packet carries a 32 bit sequence number as well as source and destination identifiers. Duplicate packets are discarded but acknowledged, in case the duplicate arose from a lost acknowledgement, and also to allow the transmitter to keep an up to date round trip time measurement.

5.4 Window Scheme

A window scheme operates on the segments of a message both to limit flow and to utilise long delay networks effectively. The scheme operates in two different ways:

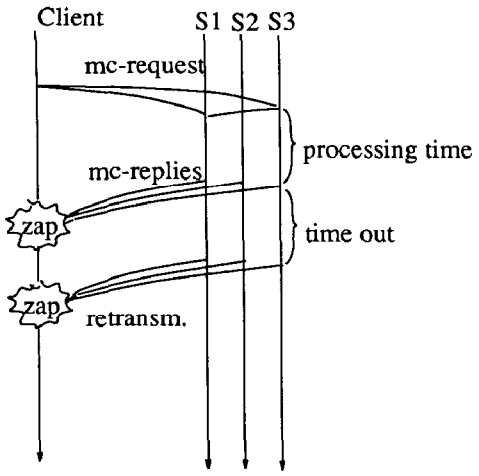
1. CMSend operates an overall transmit window for the group and a separate transmit window for each of the servers.

   The overall window is the lower bound of the smallest of the advertised receive windows and the estimated number of packets that the pipeline with the shortest delay times bandwidth path to a server will hold.

   The separate windows are used to limit the number of unicast retransmitted packets to any server.
2. CMRecv keeps a receive window for each server, plus an overall receive window for the group. When the servers start to use SMSend, the client allows some of the servers to send some small number of packets more than others. The protocol closes individual windows if one server is too far ahead of the others in a reply or if the aggregate reply window is too large.

   This is to enable a host receiving many replies to pass parts of each reply, *in order* and also *in step*, up to the next level.

Picture 2.
The Implosion Problem



5.5 The Implosion Problem

A common problem with distributed systems is the tendency of collections of related events to synchronise. In particular, if servers responding to a multicast request do so nearly all at the same time, they may swamp both the network and the client, causing a high number of packets to be dropped. If they then time out and retransmit at nearly the same time the same problem will occur again, leading to very poor overall performance.

Statistical analysis of this problem on an Ethernet shows that the limiting capacity is likely to be the client rather than the network. This analysis attempts to quantify what a *large* group size is for Ethernet operation of the protocol, so that we may choose appropriate values for timers and windows.[15,16]

Suppose that a multicast request has been received correctly by n servers, all at the same time t. Each of them, will try to *acquire the ether* or access the network during the time interval $[t,t+\delta t]$ with probability α. (This is equivalent to assuming that the servers respond with some kind of randomised delay).

If one of the servers succeeds in acquiring the ether, the others wait until it finishes transmitting, at a time t_2 say. They then compete during the next interval, $[t_2,t_2+\delta t]$, in the same way as they did for the first. If, on the other hand, there was a collision during the first contention interval, all servers involved in the collision stop transmitting at once, and repeat the contention process at time $t+\delta t$. We first assume that background traffic is so light as to be negligible. We also assume that the replies are of similar size.

With these assumptions we can calculate the mean number of contention slots needed to transmit n replies, as well as the mean number of packets needed overall. (The formulae are given in the annex).

Looking at the mean number of contention slots, which is shown in figure 1. we can see that it varies with α, which is the probability that a server will attempt to transmit during a given contention slot.

The No. of Contention Slots Needed for n Replies to a Multicast

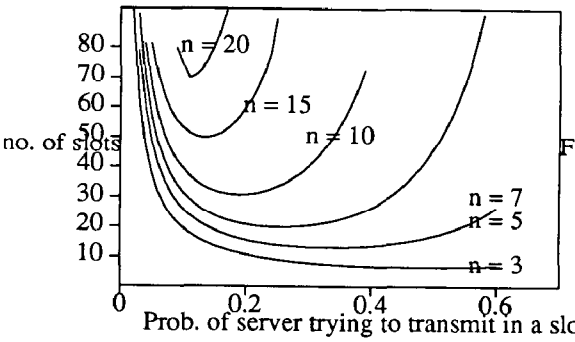


Figure 1

For fixed n, if α is small (less than 0.1 say), the number of contention slots needed is high. We would expect this to be due to the fact that many contention slots go by unused. As we increase α the number of contention slots decreases initially as fewer are left idle. Eventually, however, further increases in α lead to more and more collisions and hence to a sharp increase in the number of contention slots required.

This behaviour is similar for all n, except that the trough of the curve becomes increasingly more narrow, and occurs at a lower range of α.

Looking at the graphs for the mean number of packets sent, we see that their behaviour tallies up with our interpretation of the contention slots: When α is low, few collisions occur and only one or two packets are needed for each response. As α increases, so does the number of packets sent.(figure 2)

No. of Packets Needed by Each of Respondents to a Multicast

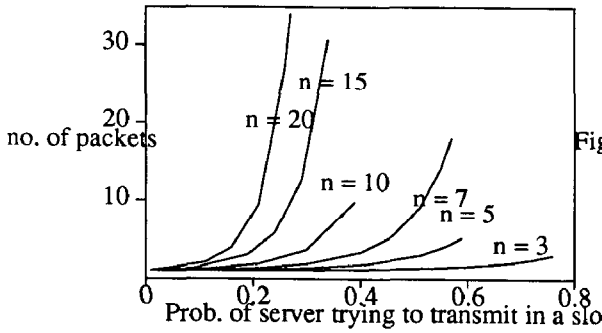


Figure 2

Note, of course, that in practice the number of slots and packets is limited by some maximum number of retries for each server. This has not been incorporated in the our analysis, but is scarcely important in cases where the cut-off point is high compared to the mean number of retries. Since we are interested in an upper bound on the number of servers which do not cause an implosion problem, i.e. for which the number of retries is low, the above analysis is adequate.

This upper bound can be obtained as follows: Looking at a curve in fig.1 for a fixed number of servers, we expect them to be safe for all values of α for which the curve is downward sloping. Inversely, for a given value of α, any number of servers n will be safe, as long as the vertical line above α intersects the downward sloping part of the curve for n.

Values for α depend partly on the number of servers involved and partly on the length of the packets they are trying to send: the length of contention slots in the Ethernet is of the order of *Contend*μs. Assume that typical reply times after a receipt of a multicast lie in the range *Range*μs. The actual values will be determined by experiment.

Since we are in the downward sloping part of the curves, collisions and hence retransmissions are rare and can be neglected for this calculation. Assuming furthermore that the reply times are evenly distributed, then the probability of an attempted reply in any contention slot can be taken as approximately

$$\alpha = \frac{Contend}{(Range - nl/Contend)}$$

where n is the no of servers and l is the length of their replies in bits.

5.5.1 Differing Background Loads

Medium or high levels of background traffic during a multicast transaction can be modelled as follows: we denote the probability that any host not involved in the multicast transaction will transmit during a current contention slot as β. Substituting and evaluating for different values of β seems to show that β affects mainly the number of slots needed: the number of packets needed increases only at very high values.

5.6 Scheduling replies

A client may able to accept the aggregate rate of reply from some number of servers. However, if the servers' reply packets tend to clump together, then the client may well be swamped by back-to-back packets.

A good implementation of the selective unicast acknowledgement scheme used by this protocol, will randomise the scheduling of replies from each of the servers.

In the literature, this problem has been solved by introducing independent random back-off schemes at each server.[8] This has the disadvantage that it cannot be adapted dynamically: reply times will be delayed even if changing circumstances make it unnecessary. This can be overcome by driving the scheduling scheme from the client.

A simple algorithms for this is to take the current aggregate window and divide it evenly over all the servers. Then round robin schedule acknowledgements to each server with a mean and variance of delay based on the time for processing one server's window worth of packets.

In the local network case, this may not be feasible, since the processing times per packet need to be fast, and the variance on packet transmission times by each server tend to be too low for such a complex statistical calculation to be done dynamically.

6. Request and Response

The reliable message protocol can be used by a request response protocol.

This has five service primitives:

1. Request(Mid, IdList, Buf, Flags, Timeout)

   This is used by the client to send a part of a request. Mid is the group identifier. IdList is a possible subset. Buf contains some of the request data. Flags indicates the service wanted and an indication if this is the last part of the request.

   Timeout here is set by a fault tolerant application, which may give up a request after some time. Normally it would be infinite.

2. Response(IdList, BufList, VoteFn, Flags, Timeout)

   This is used by the client to collect reply messages. Flags is used to indicate the type of multicast receive service. Depending how it is set, Response can return with each buffer from each server, or with the list of buffers (if servers are deemed equivalent) for this part of the message, or with all the entire messages. Timeout allows the user to poll for replies.

   An indication in Flags is set for the end of a reply.

3. ReqAbort(IdList)

   This allows the user to abort a request to a given number of servers. It results in an error packet being sent to each of those servers.

   Request, Response and Abort are used together by a client. The might be part of an automatically generated stub for a remote procedure call.

4. Listen(Id, Buf, Timeout)

   This is used by the server to listen for calls.

5. Reply(Buf)

   This is used by the server to send a reply.

   Listen and Reply are used by a server, and may be used as the counterpart of the client's stub described above.

In addition to these functions which build fairly straightforwardly on the message level, the first packet of a reply message can be used to acknowledge delivery of a request to that server, and a new request from the client to the server can be taken to indicate that the client has received *all* the replies that it is interested in, and that servers may discard any held replies.

Requests and responses carry a conversation identifier to associate them, and a special bit to distinguish them.

6.1 Voting Functions

The voting function is provided by the user to the request-response level. Most frequently, this is a function

that takes a list of buffer descriptors and simply compares some particular field to some value. It may also compare the list of replies against each other and return the most common.

6.2 Operation of Request/Response using Message Level

The request-response level uses the message level. This operates as follows:

- Request sets up state for the request, and uses CMSend to issue each part of the request.
- Response uses the state from Request, and calls CMRecv to collect appropriate parts of replies from servers. When unwanted replies arrive at a client, CMRecv may call Abort to send error reports to save servers unwanted work. Similarly, if the VoteFn has been applied successfully, Abort will be called for any remaining servers.

  Each call to Response hands the equivalent segments of replies from different servers to the user
- ReqAbort is called by the user when some out of band event means that the Request is no longer useful. It uses Abort to cancel all the servers' effort.
- Listen uses SMRecv to collect each part of a request from a client. It saves the source address of the request.

  Reply simply uses SMSend to send each part of the reply message to the saved address.

6.3 Failure Modes and Cancelled Requests

The failure modes of the request response layer are derived from the failure modes of the message layer in the obvious manner.

If the client aborts a request while receiving a reply, the protocol sends a message level abort to all the servers in the group. This is to enable the servers to tidy up any state and discard the replies they may have buffered.

7. Implementation

There is an experimental implementation of this protocol in *user space* under Berkeley Unix. It is similar in some ways to the ones designed in[17,18]

Initial use and performance of this protocol on a single LAN is under investigation.

For a single packet call and reply, initial performance against number of servers behaves as one would expect from the theoretical analysis. We have not yet experimented with multipacket calls and replies, either on a single LAN or on an Internet.

8. Conclusions

We have presented the design of a new transport protocol which supports an n-Reliable multicast Request Response type service. Initial implementation experience shows that this kind of protocol can reduce the number of packets on the network, whilst more conveniently providing a similar type of service to sequential unicast requests and responses.

Analysis supports this experience for LAN use of the protocol.

Based on this analysis, the protocol implements a multiple coupled windowing scheme of flow control for n to one communication, which is designed to solve the problem of overruns in both a client host, and in intermediate nodes in an internet.

The implementation and use of voting functions requires further investigation. The request response layer should not have to buffer all the parts of all the replies until the vote function has been applied. This is why we introduced a message level vote function. This is in some way derived (hopefully automatically) from the higher level vote function, so that the client message layer may cancel unwanted replies as soons as possible in an exchange. This derivation may be non-trivial, and so for the present, the vote function may only be a simple comparison of some portion (or all) of some number of reply messages, which returns the list of addresses of servers whose replies are acceptable.

9. Annex 1 - Analysis of Implosion for Single LAN

We make the assumptions presented in section 5.5 above.

Let $p_{i,i-1}$ be the probability that out of i receivers still trying to respond, one is successful during the current contention slot (thus reducing the number of outstanding responses to $i-1$). This will be the case if only one tries to transmit while all the others are silent:

$$p_{i,i-1} = i\,\alpha(1-\alpha)^{i-1} \qquad \text{where} \quad 0<i\leq n.$$

Let p_{ii} be the probability that no message is transmitted during the current contention interval (either because no receiver tried to transmit or because of a collision).

$$p_{ii} = 1 - i\,\alpha\,(1-\alpha)^{i-1}$$

Then the probability $Pr(n)$ that all n messages are transmitted using only n contention slots is the probability of a successful transmission following every slot:

$$Pr(n) = p_{n,n-1}\,p_{n-1,n-2}\cdots p_{21}\,p_{10} = \prod_{i=1}^{n} p_{i,i-1}$$

The probability $Pr(n+1)$ that $(n+1)$ slots are needed can be calculated as follows: out of $n+1$ slots n must have been used for successful transmissions whereas the remaining one was badly used in the sense that either it went by unused or it contained a collision. This badly used slot may have been the first when all n receivers still had to respond, or the second, with $(n-1)$ receivers still trying to respond, or indeed any of the others. Hence:

$$Pr(n+1) = p_{nn}\prod_{i=1}^{n} p_{i,i-1} + p_{n-1,n-1}\prod_{i=1}^{n} p_{i,i-1} + \cdots + p_{11}\prod_{i=1}^{n} p_{i,i-1}$$

$$= \sum_{i=1}^{n} p_{ii} \prod_{i=1}^{n} p_{i,i-1}$$

Similarly, it can be shown that:

$$Pr(n+2) = [\,p_{nn}(p_{11}+p_{22}+\cdots+p_{nn}) + p_{n,n-1}(p_{11}+p_{22}+\ldots.+p_{n-1,n-1}) + \cdots + p_{22}(p_{11}+p_{22}) + p\hat{n}\,]\prod_{i=1}^{n} p_{i,i-1}$$

$$= \left(\sum_{i=1}^{n} p_{ii} \sum_{j=1}^{i} p_{jj}\right) \prod_{i=1}^{n} p_{i,i-1}$$

and:

$$Pr(n+3) = \left(\sum_{i=1}^{n} p_{ii} \sum_{j=1}^{i} p_{jj} \sum_{k=1}^{i} p_{kk}\right) \prod_{i=1}^{n} p_{i,i-1}$$

In general, these and further expressions can be obtained by using a lower triangular matrix P and vectors $\underline{p}$ and $\underline{u}$:

$$P = \begin{bmatrix} p_{11} & 0 & 0 & \cdots & 0 \\ p_{22} & p_{22} & 0 & & 0 \\ p_{33} & p_{33} & p_{33} & & 0 \\ . & & . & & . \\ . & & . & & . \\ . & & . & & . \\ p_{nn} & p_{nn} & p_{nn} & \cdots & p_{nn} \end{bmatrix}, \quad \underline{p} = \begin{bmatrix} p_{11} \\ p_{22} \\ p_{33} \\ . \\ . \\ . \\ p_{nn} \end{bmatrix} \quad \text{and} \quad \underline{u} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ . \\ . \\ . \\ 1 \end{bmatrix}$$

Using the dot product of $\underline{p}$ and $\underline{u}$, we can write:

$$Pr(n+1) = \prod_{i=1}^{n} p_{i,i-1} \, (\underline{p}.\underline{u})$$

$$Pr(n+2) = \prod_{i=1}^{n} p_{i,i-1} \, (P\underline{p}).\underline{u}$$

and, in general,

$$Pr(n+i) = \prod_{i=1}^{n} p_{i,i-1} \, (P^{i-1}\underline{p}).\underline{u}$$

Then the mean number of slots needed is μ_S:

$$\mu_s = \sum_{i=0}^{\infty} (n+i) Pr(n+i)$$

$$= \left(n + \left(\sum_{i=1}^{\infty} (n+i) P^{i-1}\right) \underline{p}.\underline{u}\right) \prod_{i=1}^{n} p_{i,i-1}$$

Now P can be diagonalised as $Q \Lambda Q^{-1}$, where $\Lambda = diag(\lambda_i)$, the diagonal matrix with the eigenvalues of P on the diagonal, and Q is the matrix whose columns are the eigenvectors of P, Q^{-1} being the inverse of Q.

We then have :

$$\mu_S = \left[n + \left(\sum_{i=1}^{\infty} (n+i) (Q \Lambda Q^{-1})^{i-1}\right) \underline{p}.\underline{u}\right] \prod_{i=1}^{n} p_{i,i-1}$$

$$= \left[n + Q \left(\sum_{i=1}^{\infty} (n+i) \Lambda^{i-1}\right) Q^{-1} \underline{p}.\underline{u}\right] \prod_{i=1}^{n} p_{i,i-1}$$

$$= \left[n + Q \, diag \left(n \sum_{j=1}^{\infty} \lambda_j^{j-1} + \sum_{j=1}^{\infty} i \, \lambda_j^{j-1}\right) Q^{-1} \underline{p}.\underline{u}\right] \prod_{i=1}^{n} p_{i,i-1}$$

The eigenvalues λ_i of P are $p_{11} p_{22} \cdots p_{nn}$, and hence $|\lambda_i| < 1$, for all i.

Therefore the series

$$\left(n \sum_{j=1}^{\infty} \lambda_j^{j-1} + \sum_{j=1}^{\infty} i \, \lambda_j^{j-1}\right) = \frac{n}{1-p_{ii}} + \frac{1}{(1-p_{ii})^2} = \frac{n p_{i,i-1} + 1}{p_{i,i-1}^2}$$

It can then be shown that

$$Q \, diag(d_i) Q^{-1} \underline{p}.\underline{u} = \sum_{i=1}^{n} \frac{p_{ii} d_i}{\prod_{j=1, j \neq i}^{n} (p_{ii} - p_{jj})} \quad \text{where} \quad d_i = \frac{n p_{i,i-1} + 1}{p_{i,i-1}^2}$$

Note that this formula cannot be applied if any $p_{i,i}$, $p_{j,j}$ are equal for any different i,j. In this case, it is impossible to diagonalise the matrix P. However, because the spectral radius of P is less than one, μ_s is still guaranteed to be finite.

The mean no. of packet transmissions can now be obtained as follows: out of μ_S contention slots needed to transmit all replies, we know that there were n slots in which a transmission was successful, incurring one packet each. Out of the remaining $(\mu_S - n)$ slots, a proportion were idle slots, and the rest involved collisions.

The probability of i receivers being silent during a current contention slot is $(1-\alpha)^i$. So at a stage where i receivers still have to respond, the probability of them being silent given that there was no packet transmitted during the current slot is:

$$\frac{(1-\alpha)^i}{1 - i\alpha(1-\alpha)^{i-1}}$$

Over all stages, this averages to:

$$s = \frac{1}{n} \sum_{i=1}^{n} \frac{(1-\alpha)^i}{1 - i\alpha(1-\alpha)^{i-1}}$$

The probability of collisions is $1-s$.

Trivially, the mean number of packets transmitted in an unused contention slot is 0.

The mean number of packets involved in a collision at a stage when i receivers still have to respond is:

$$Col_i = \sum_{j=2}^{i} j \binom{i}{j} \alpha^j (1-\alpha)^{i-j} \quad where \quad i \geq 2$$

Hence over all stages the mean number of packets per collision is: $ppc = \frac{1}{n} \sum_{i=2}^{n} Col_i$.

So the mean number of packets sent in μ_S contention slots is μ_P:

$$\mu_P = n + (\mu_S - n) s * 0 + (\mu_S - n)(1-s) ppc = n + (\mu_S - n)(1-s) ppc$$

9.1 Differing Background Loads

Medium or high levels of background traffic during a multicast transaction can be modelled as follows: we denote the probability that any host not involved in the multicast transaction will transmit during a current contention slot as β. The probability of a successful response to the multicast then changes to

$$p_{i,i-1} = i\alpha(1-\alpha)^{i-1}(1-\beta) \qquad where \quad 0 < i \leq n.$$

Consequently

$$p_{ii} = 1 - i\alpha(1-\alpha)^{i-1}(1-\beta)$$

Substituting these new values into μ_S and μ_P and evaluating for different values of β seems to show that β affects mainly the number of slots needed: the number of packets needed increases only at very high values.

10. Annex 2 - Potential Hardware Support

For large multicast requests, it would be convenient to have network interfaces that filtered not only on network

address, broadcast and multicast address, but also on sequence number of packets within messages. This would obviate the need for a complex selective retransmission scheme.

Multicast protocols are most attractive where the underlying technology is broadcast. One of the main problems on a broadcast medium is the excess work for hosts not interested in a current broadcast packet. Current hardware support for *filtering* packets is based simply on per host multicast address lists.

A convenient extension of this would be to filter on packet source and sequence number. The Cambridge ring minipacket mechanism[19] provided the former, but no LAN or WAN interfaces provide sequence number filtering. Mockapetris suggests a scheme for a pseudo "alternating bit protocol" to filter unwanted multicast packets.[20] Mogul, Rashid and Accetta[21] suggest a more general but similar mechanism within the operating system software to support efficient user level protocol implementation.

This could be extended down to hardware, to use multiple filters per multicast address. Hosts would accept all packets within some sequence space addressed to it, and the hardware would roll the sequence number filter forward as each packet was successfully passed up to the application.

11. Annex 3 - Protocol State Required

```
-- Multicast semantics

constant ANY_N_RELIABLE 1
constant KNOWN_N_RELIABLE 2
constant SOME_KNOWN_N_RELIABLE 4

constant WHICH_MASK 0x07

def Any_N(x)
    ((x->NReliable&WHICH_MASK)==ANY_N_RELIABLE)
def Known_N(x)
    ((x->NReliable&WHICH_MASK)==KNOWN_N_RELIABLE)
def SomeKnown_N(x)
    ((x->NReliable&WHICH_MASK)==SOME_KNOWN_N_RELIABLE)

constant LESS_N_RELIABLE 8
constant MAJORITY_RELIABLE 16

def Reliable(x)
    ((x->NReliable&HOW_MANY_MASK)==0)
def LessThanN(x)
    ((x->NReliable&HOW_MANY_MASK)==LESS_N_RELIABLE)
def Majority(x)
    ((x->NReliable&HOW_MANY_MASK)==MAJORITY_RELIABLE)

constant HOW_MANY_MASK 0x70

-- Acknowledgement Style

constant FLOOD_STYLE 0
constant MACK_STYLE 1
constant SACK_STYLE 2
constant NACK_STYLE 3

-- Per server info kept at client

type PerServer {
    Mid    Server;
    int    State;
    Seqno  PktSeq;      -- Seq no. ack'd by this server so far --
    Window RxWindow;    -- Window for this server            --
    Window TxWindow;    -- Server's rx window                --
    Pkt    *Rep;
-- Stats on Calls --
    int    Rtx;
    int    ReplyCount;
} PS;

-- Client state table holds all info for
-- this client & forall servers

type ClientState {
-- Addressing info
    Mid    Client;      -- My address
    Mid    GroupAddr;   -- Group Address
    Seqno  AllPktSeq;   -- Latest seqno over all
    Window AllWindow;   -- Overall window on all svrs
-- Info required Per Server in the Group
    PS     *Member;
    int    Grouplen;    -- Length of group list
-- Fn supplied by user to vote on replies if thats whats wanted
    function voter();   -- User supplied voting fn
-- Protocol Specific Info
    int    NReliable;   -- Type of multicast
    int    Style;       -- Ack style
    Tim    Timeout;     -- Curr Timeout for each member
    int    State;       -- Overall state
-- Request Info
    Seqno  This;        -- Request Sequence number
    Pkt    Req;         -- Current request
-- Stats on Calls
    int    TotalRtxs;   -- Count of all rtxs
-- Handy network things
    int    Socket;      -- Handle on network
    int    AddrSize;    -- sizeof a group addr
} CPCB;


-- State for each Member server in Group
-- Total state is some convolution of the member states

constant IDLE_STATE   0
constant REQ_STATE    1
constant REP_STATE    2
constant ACK_STATE    3
constant NACK_STATE   4

-- Overall state of call

constant PROGRESS_STATE  0
constant REPLIED_STATE   1
constant FAILED_STATE    2

constant OVERSUBSCRIBED 64

-- Server state = 1 of these foreach client + general info

type PerClient {
    Mid    Client;      -- Address of a past client
    Seqno  Last;        -- Last msg # from that client
    Seqno  PktSeq;      -- Pkt in msg so far
    Window Window;      -- Client's rx window for us
    int    State;       -- Our state for that Client
    Pkt    Rep;         -- Saved reply for that client
    PerClient *Next;
} PC;

type ServerState {
-- Address Info
    Mid    Server;      -- This Server
-- Per Client Info
    PC     *PerClient;
    int    ClientCount;
    Window Window;      -- My current rx window
    Pkt    Req;         -- Current request
-- Protocol Info
    Tim    Timeout;     -- Timeout on Replies ??
    int    State;       -- state of server
-- Fn supplied by user to do server work
    function Work();
-- General things of use
    int    Socket;      -- Network handle
    int    AddrSize;    -- Length of addr
} SPCB;


  -- Window Size base values

  -- This is typically for a unicast multipacket msg
  constant AGREGATE_WINDOW      16

  -- While this is a max for a fast replying server
  -- to be ahead of slow ones
  constant EACH_SERVER_WINDOW    4
```

12. Annex 4 - Packet Formats

```
-- Packet structure

type p {
    Seqno  seq;   -- Req/Rep Seq matching
    char   type;  -- Sort of pkt
-- Per pkt info
    Seqno  pseq;  -- Pkt# or Ack# within a
                  -- single Req/Rep
    Window win;   -- Current advertised rx window
    short  len;   -- This pkt len in bytes
    short  flags; -- indicates last pkt now
    Buffer dat;
                  -- User Data
                  -- Also used for bitmap in SACKS
                  -- Sack pkt has list of holes,
                  -- Seqno + Len of each hole
} Pkt;

-- Packet Types

constant REQ  1
constant REP  2

-- Client to Server REP ACKS --
constant ACK  3
constant NACK  4
constant SACK  5

-- Server to Client Req Pkt ACKS --
constant PACK  6

constant LAST  1
constant MARK  2
```

13. References

References.

1. J Crowcroft and M Riddoch, "Sequenced Exchange Protocol", *UCL Internal Note 1824, ADMIRAL Project Note A.341*, (1985).

2. Andrew D. Birrell and Bruce J. Nelson, "Implementing Remote Procedure Calls", *ACM Trans.Comp.Sys.* **2**(1) pp. 39-59 (Feb 1984).

3. S Wilbur and B Bacarisse, "Building Distributed Systems with Remote Procedure Call", *IEE Software Engineering Journal* **2**(5) pp. 148-159 (September 1987).

4. S.Wilbur and P.J.M.Polkinghorne, "Distributed Robust Filestore", *Internal Note*, (1987).

5. Kenneth P. Birman and Thomas A. Joseph, "Reliable Communication in the Presence of Failures", *ACM Trans.Comp.Syst.* **5**(1) pp. 47-76 (Feb 1987).

6. D.R.Cheriton and Willi Zwaenepool, "Distributed Processes in the V-kernel", *ACM Transactions on Computer Systems* 3pp. 77-107 (May 1985).

7. L. Hughes, "A Multicast Transmission Taxonomy", *Technical Report Series no. 221*, pp. 1-15, Newcastle University (Aug 1986).

8. S. E. Deering and Dave E. Cheriton, "Host Groups: A Multicast Extension to the Internet Protocol", *RFC-966*, pp. 1-27 (Dec 1985).

9. Y. Dalal, "Broadcast Protocols", *SU Ph.D Thesis*, .

10. JH Saltzer, DP Reed, and DD Clark, "End-to-End Arguments in System Design", *ACM Transactions on Computer Systems* **2**(4) pp. 277-288 (November 1984).

11. D.R. Cheriton, "VMTP: a transport protocol for the next generation of communication systems", *Computer Communications Review* **16**pp. 406-15 (5-7 August 1986).

12. F Panzieri and S Shrivastava, *Rajdoot: A Remote Procedure Call Machanism Supporting Orphan Detection and Killing*, Univerity of Newcastle upon Tyne, Computing Laboratory (1985).

13. J. Chang and N.F. Maxemchuk, "Reliable Broadcast Protocols", *ACM Trans. Comp. Systems.* **2**, 3pp. 251-273 (Aug. 1983).

14. J. Postel, "Transmission Control Protocol", RFC 793, DARPA (September 1981).

15. A. J. Frank, L. D. Whittie, and A. J. Bernstein, "Multicast Communication on Network Computers", *IEEE Software*, pp. 49-61 (May 1985).