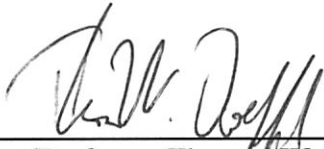


Reliable Multicast for Small Wireless Networks

Reliable Ad Hoc Communication for the Electronic Student Notebook

Roberto Almanza
Department of Computer Science
Brown University
roalmanz@cs.brown.edu
May 2003

Submitted in partial fulfillment of the requirements for the Degree of Master of Science in the
Department of Computer Science at Brown University



Signature (Professor Thomas W. Doepfner Jr.)

5/16/03

Date

1. Introduction

The goal of this project is to provide a reliable multicast client for use in the Electronic Student Notebook (ESN) project. The ESN project is a research project dedicated to exploring the use of ad hoc collaborative electronic notebooks deployed on Table PC's.

Imagine a standard classroom; a teacher stands at the front of the classroom writing notes and dictating. As the teacher lectures, the students rapidly write down notes on their spiral notebooks or lecture slides provided by the instructor. Although most people are comfortable with this particular setting, few would disagree that there are several glaring inefficiencies and annoyances with this particular method. We have all experienced the annoyance of having to photocopy, or even hand copy, a classmate's notes when we are unable to attend lecture. There have also been plenty of times when we have crowded around one or two classmates as they illustrate a concept from lecture by referencing their notes from class.

The Electronic Student Notebook solves many of the shortcomings of the standard paper notebook. ESN allows individuals to easily share documents and annotations. These documents and annotations can also be saved for future use.

Assuming no fixed infrastructure, the Electronic Student Notebooks allows users to meet in any location sharing documents and annotations via the ad hoc network consisting of the users in the meeting. This project provides the communication substrate needed to provide group data dissemination for settings with no fixed infrastructure.

2. Overview

The primary goals of the Reliable Multicast Client provided to the ESN project are as follows:

1. Reliable Data Transfer – all data transmitted is received by all participating nodes

2. In-order delivery – data transmitted from node x is received in the same order at all participating nodes

3. Efficient Data Transfer – the client provides an adequate data transfer rate

4. Scalable Performance – the client performs well in networks of varying size

5. Resource Efficient – Since the client is running in the same process as the ESN project, it is important that the client not consume large amounts of resources

To achieve the goals specified above it was essential to minimize the amount of control messages being sent across the networks. This essentially meant that whenever possible we would avoid doing things like broadcasting *hello* messages. This particular design choice will be evident throughout the following discussion of our implementation.

2.1 Rationale

The rationale behind our design of the RMC is largely based on two interrelated problems: feedback implosion and coordinating agreement using unreliable communication.

Feedback implosion is the problem of overwhelming nodes in the system with messages meant to relay the current status of the system. For example this would happen if a data-transmitting node required all the receiving nodes to respond with acknowledgments. In this case the transmitting node might send a message to n nodes who then respond with n *ack*'s, thus overwhelming the originator of the message.

To avoid feedback implosion, we chose to have nodes transmit gossip requests for missing data instead of *ack*'s.

Maintaining precise group membership in order to allow the system to know precisely who should have received the data at any given point is also a very difficult problem to solve. Normally, when a node transmits data it is preferable to have the receiving nodes be well known. The sender would then know which nodes should have received the transmitted data. There are two problems with this approach: feedback implosion and maintaining accurate membership. The problem of feedback implosion is clearly specified above.

Maintaining an accurate membership in an asynchronous network using unreliable communication is a problem that is regarded as a difficult problem to solve. Many control messages would need to be transmitted to maintain a group membership that would likely only approximate the actual membership. With nodes joining, leaving and failing the algorithm needed to coordinate the acknowledgment from the current proposed group members quickly becomes very complicated. The complexity of this was not the sole factor behind our choice to avoid coordinating precise agreement among the set of nodes receiving data. The performance of the system would also be degraded by the control messages that would likely need to be retransmitted, since we are using unreliable communication. The system would also likely suffer significant delays when data is not transmitted until nodes have accurately determined the current group membership and then reached agreement.

Our design rationale was to avoid coordinating unreliable nodes with

unreliable communication. Thus we allow for nodes to probabilistically decide on the next step. This choice is ideal for the particular problem we are addressing, since the system is able to deal with multiple uncoordinated nodes. For example, if several nodes probabilistically decide to respond to one node's request for missing data, then the node receiving the responses gracefully deals with duplicate responses.

The cost of dealing with multiple responses is less than the cost of coordinating a single response from a dynamic set of nodes.

2.2 Changes in Group Membership

Hierarchical structures are often used to allow for multicast on wired networks. We did not pursue the use of such structures due to the dynamic nature of the network we are working with.

Since we can not assume that the structure of our network is fixed, it is not feasible to simply use a hierarchical structure for our group data dissemination. If we had used a hierarchical structure, something as simple as a user leaving a meeting could potentially cause serious performance degradation.

For example, assume Alice is participating in a meeting using the ESN client and the multicast client has chosen to use a hierarchical structure. Further assume that Alice is placed near the top of the hierarchy. If Alice were to leave the meeting the system would require a restructuring based on the departure of Alice from the group. This restructuring would require the nodes in the system to agree on the new hierarchical structure of the group without Alice. This coordination would likely introduce some lag, during the restructuring, for messages being transmitted that rely on the hierarchical structure. Additionally, the coordination would introduce a large

number of control messages into the system. This is something we clearly want to avoid.

The RMC does not require any such coordination of hierarchies; therefore, for the example given above the multicast client would only need to adjust the group membership once it determines that Alice is gone. The modification of the group membership is definitely not a change that needs to take place immediately. The only repercussions that would arise from the inaccurate membership would be that the RMC might reply with slightly lower probability. (*Lower than the ideal based on the exact group membership.*)

In the case stated above, with members leaving a session, we would also incur unnecessary messaging if we had chosen to coordinate acknowledgements from all users for every packet transmitted. If a member, Bob, was transmitting data in the above scenario for the entire duration of Alice's transition from member to non-member, then Bob would need to wait for Alice's acknowledgment while Bob still believes Alice to be a part of the group. For this case, it is quite possible that Bob might block or send unnecessary control messages. The potential for blocking amplifies the cost of requiring acknowledgments from all members.

For algorithms that rely heavily on precise group membership it is also possible to adjust the membership timeout so that membership departures are easily detected. This solution would not, however, come without a cost. By lowering the group membership timeout, even non-faulty members that are slightly lagged could be removed from the membership. The membership would then become volatile and inaccurate. Thus, adjusting the timeout value for membership does not necessarily

improve the performance of the system. Our approach avoids the sometimes intractable problem of maintaining precise group membership.

Our decision to use gossip requests rather than gossip acknowledgements from the members in the group would clearly perform better in our example of user departing from a meeting. With our protocol Bob would transmit his data during, through and after Alice's departure and not suffer any decreased performance. Essentially, Bob would simply not hear a request from Alice for missing data (*Gossip Request*). Therefore the performance of the system remains stable during the departure of members from the session. The system performs the same whether the departure was announced or due to a failure.

For these reasons we believe the decentralized approach that we have taken with Gossip will perform well.

We further discuss alternative algorithms at the end of the paper.

The rest of the paper will provide details on the implementation of our ad hoc Reliable Multicast Client. I will conclude with a brief discussion on the results obtained using the client for communication in the ESN project.

3. Implementation

The Reliable Multicast Client (RMC) was implemented on Windows XP using C# and the Microsoft .Net platform. We made the assumption that all nodes are within reach of one another. Given the hardware that we are working with, as well as the reach of 802.11b, this is a fair assumption. The client uses UDP, which is an unreliable connectionless transfer protocol.

The RMC provides the following interface:

```
void Close()
byte[] Receive1(ref IPEndPoint end);
byte[] Receive2(ref IPEndPoint end,
                 int waitTime);
void Send(byte[] data);
```

Close shuts down the client. *Receive₁* waits indefinitely for incoming data, setting *end* to the corresponding endpoint from which the data was received. *Receive₂* waits until data is received or until the number of milliseconds prescribed by *waitTime* expires. *Send* transmits the specified byte array.

3.1 Packet Structure

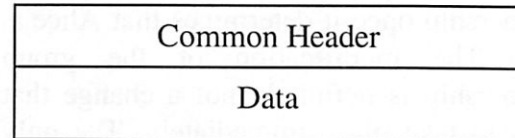
All packets transmitted onto the network follow a strict packet format. The structure of each packet is specified by a struct. The structs are then converted to byte arrays and from byte arrays back into structs using the *unsafe* keyword in C#, which allows for pointer manipulation. (*Pointers are not allowed in C# without the unsafe keyword around the body of code using the pointer*) The more important packet structures are specified below, along with a brief description as needed.

Common Header- Header for all messages

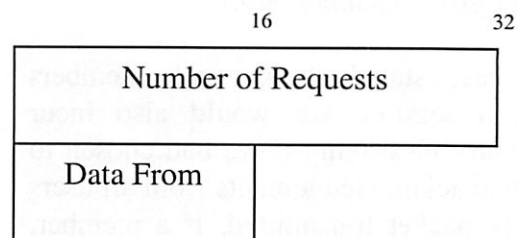
		16	32
Message Type	Source Node	Quad1	Quad2
Quad3	Quad4	Data Length	
Data Length			

Source Node specifies the origin of the packet. The *Quad*'s specify the multicast

address that this packet is associated with. The 32 bit *Data Length* specifies the length of the data following the header. Messages sent over the network have the following format:

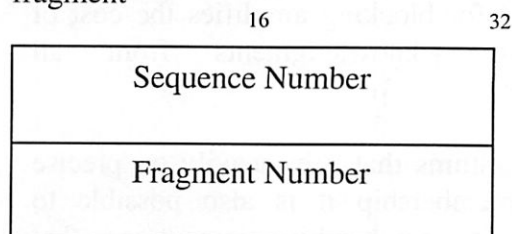


Gossip Request Header – header for each gossip request

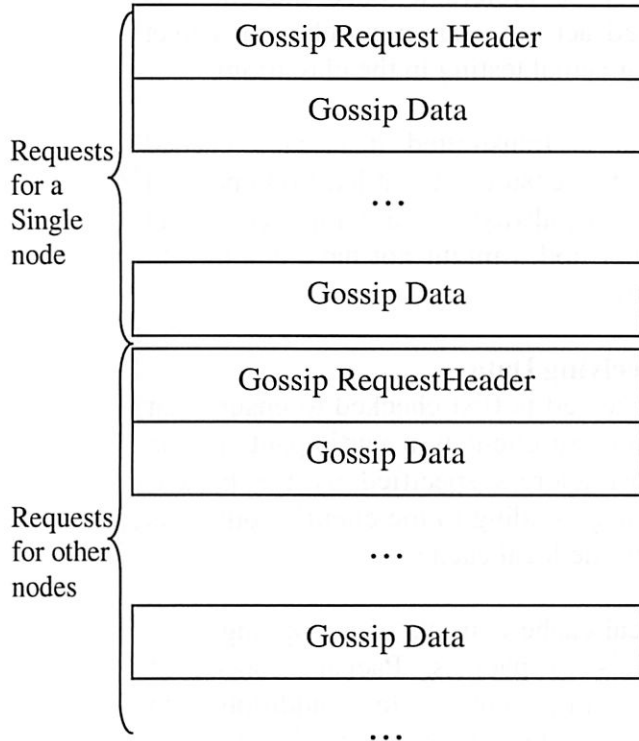


Number of Requests specifies how many requests will follow this header. *Data From* denotes the node associated with the fragments being requested.

Gossip Data- a gossip request for a missing fragment



Each gossip is of the following format:

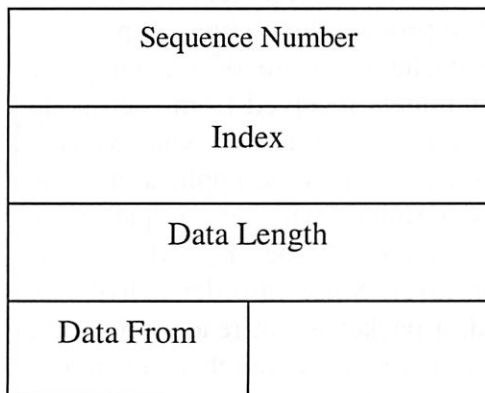


There is a size limit on the size for any given gossip of the format specified above. If a request is too large it is fragmented into several gossip requests.

Data Info- uniquely identifies fragment

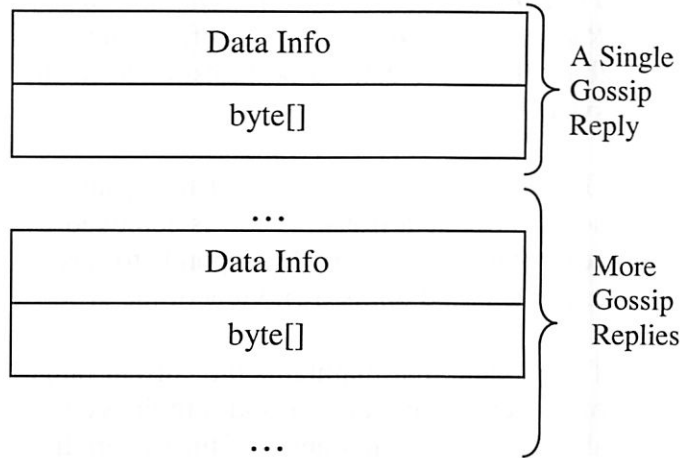
16

32



The $\langle \text{Data From}, \text{Sequence Number}, \text{Index} \rangle$ triplet uniquely identifies a data fragment.

Gossip Replies use *Data Info* to identify incoming gossip replies. The format of the gossip reply is:



Although multiple replies are supported, we limit the number of replies in a single transmission to one. This limitation increases the probability of the response being successfully delivered. This is largely done because in the wireless environment large UDP packets have a higher probability of failing to be delivered. Any type of gossip packet is not retransmitted, therefore it is important to increase the probability of these messages arriving successfully.

The structs listed above are the most critical packets in the Reliable Multicast Client infrastructure.

3.1.1 Component Overview

In the following sections we will outline the major components of the Reliable Multicast Client. These components are:

1. Data Transmission
2. Receiving Data
3. Gossip Requests
4. Gossip Replies
5. Group Membership
6. Maintaining Sequence Numbers
7. Canceling Gossip Requests

8. Caching Recent Requests
9. Cache Eviction

The first four components (*Data Transmission, Receiving Data, Gossip Requests and Gossip Replies*) form the core that allows for data to be transmitted in the system.

Group Membership allows for the system to adjust how data transmission is achieved by adjusting how each node responds to gossip requests based on its own view of the group.

The module that maintains the current range of sequence numbers for nodes in the system allows for the components of the system that deal with retransmission to be aware of missing data.

The last three components allow the system to minimize unnecessary data transmission, as well as unnecessary caching.

3.2 Sending Data

For data transmission, data is first assigned a unique monotonically increasing sequence number (unique per node). A thread object is then created that encapsulates the process of actually fragmenting the data and transmitting the fragmented packets on the network. This thread object is then queued onto a thread-pool.

The fragmentation consists of fragmenting the data into fragments of fixed size, currently 1024 bytes. Each fragment is uniquely identified by the triplet *<Node ID, Sequence Number, Fragment Number>*, where *Node ID* is unique for each node. Currently unique node ID's are assigned by a randomly generated number. This is currently done for testing purposes and to allow for quick testing on the Virtual Area Network being used for testing the ESN project. The client is also capable of simply

reading a unique id from a file. This would definitely be preferable once the client is deployed across a larger number of Tablet PC's for actual testing in the classroom.

As data is transmitted it is also cached locally; this ensures that at least one node in the network always has a completed packet that other nodes might not have completely received.

3.3 Receiving Data

Data received is first checked to ensure that the receiving client is a participant for the multicast address specified by the packet. Data corresponding to the client's address is added to the local cache.

The local cache consists of a mapping from node id's to packets. Packets consist of Packet Fragments. In addition to maintaining this mapping, the local cache also notes the range of sequence number received from each host.

As incoming data completes pending packets, these packets are inserted onto a list of completed packets that are ready to be sent up to the application.

When packets are completed a thread is awakened to process the completed packets. Since the cache is aware of the range of sequence numbers received from each node and since it is also aware of what packets have been delivered to the application, it is able to determine whether a particular packet is ready to be passed to the application layer. Since in-order delivery is guaranteed, a packet is not ready if packets with a lower sequence number, from the originating node, have not yet been sent to the application layer.

There are scenarios in which the local cache believes some node x has a range of

sequence numbers $b...z$, while there exists some packet a that has not yet been received. It is possible that b is delivered to the user and then a is later received. In this particular scenario the thread processing incoming data marks a as being out of order and ignores this packet, thus preserving in-order delivery. Below we specify what steps the client takes to avoid this particular case.

3.4 Gossip

Gossiping is handled by a thread, which periodically walks through the local cache extracting information on missing fragments. This data is gathered by seeing the range of sequence numbers that are known for a particular node and seeing which packets have been received. For packets in the known range of sequence numbers that have no fragments at all, the gossip thread only requests the first fragment. Any packet that has at least one fragment received is time stamped with the time at which it was received. Given the number of fragments, and thus overall data length, each packet only requests missing fragments once sufficient time has passed since the first fragment was received. This waiting allows the data to be transmitted uninterrupted from the sending node. A constant `WAIT_PER_FRAGMENT` is currently defined to determine how long to wait for each fragment. Thus a packet with 5 fragments would wait, $5 * \text{WAIT_PER_FRAGMENT}$ milliseconds before requesting missing fragments.

The gossip thread requests pending packets in batches, therefore if a node x is missing fragments x,y,z for packet I then the gossip thread would request these fragments in a single request along with other missing fragments. Currently, the size of the entire gossip request is trimmed to a constant size. Batches of gossip requests that surpass this constant are partitioned accordingly.

Although uniquely identified, gossip requests are only transmitted once. Nodes never send gossip requests for missing gossip packets, nor do nodes enforce in-order delivery of gossip packets.

As gossip requests are received, the client probabilistically processes the incoming request. The client chooses to process the incoming request with probability $1/(\text{membership size})$. The motivation behind this choice for probabilistic replies is to force a node to reply to gossip requests with lower probability as the number of nodes transmitting data in its vicinity increases. The process of maintaining membership is specified below.

Gossip replies are broadcasted by nodes once they determine that they can fulfill the request. Any given reply is no larger than the fragmentation constant. Replies are sent one fragment at a time to increase the likelihood of successful delivery.

As nodes receive gossip replies, they process the replies, determining if the data received is need for a pending packet fragment. This allows for all nodes that hear a reply to take advantage of the delivered response, despite the fact that the reply might not have resulted from a request issued by that particular node.

The Ad-Hoc network thus acts as a data store of sorts, where all nodes are able to benefit from potentially overlapping gossip requests.

3.5 Membership

Each node maintains a membership list, which is used to determine whether or not to respond to gossip requests. This list is constructed by maintaining information on which nodes have successfully transmitted

data (User data or gossip data) directly to a particular node. Membership is therefore solely determined by what nodes are currently transmitting data. The list is dynamic, since members are removed once data has not been received for some amount of time. This method of maintaining membership is a direct result of the overarching goal of minimizing the number of control messages. In this case control messages are avoided by extracting the membership information from actual data being transmitted.

3.6 Sequence Numbers

As stated above, it is immensely important to ensure that nodes are aware of the range of sequence numbers being transmitted. To ensure that this information is kept up to date, nodes transmitting data keep track of the most recent range of sequence numbers transmitted. A thread then periodically transmits this range if the node has transmitted data recently. This is one of very few control messages beyond gossip requests and replies.

The only drawback to this approach is the fact that it is possible for a node joining a multicast address to receive data that was transmitted slightly before it joined (Currently ~20 seconds, this can be adjusted by simply changing a few constants). We do not believe that this is a significant problem.

Since the sequence number range is the only data needed for a node to become aware of the existence of missing packets, the sequence numbers could also be used for very limited routing. Currently, nodes probabilistically rebroadcast advertisements of sequence number ranges. This allows for adequate one hop routing. (*The sequence number advertisements are marked with a TTL to avoid flooding the network.*)

3.7 Canceling Gossip Requests

It can often be the case that a node broadcasts a gossip request that is satisfied by one node before another node has had an opportunity to process the request. In this particular case, it is possible that the request is satisfied by the first node, yet the other node will also broadcast a reply. It is easy to see how this problem is amplified when several nodes probabilistically choose to reply after a node has already had its request satisfied.

To avoid this problem, we have chosen to broadcast cancellations as packets are delivered to the application. Nodes maintain a list of cancellations received from other nodes which specify the highest sequence number processed for a particular node. Using this data, nodes are then able to ignore subsequent gossip requests from these nodes for the corresponding hosts that they have sent cancellation requests for. Since these cancellations are transmitted unreliably, initially it might seem like a better mechanism is needed to counter this problem. However, this method sufficiently suppresses unnecessary data transmission in conjunction with the probabilistic replies based on dynamic group membership.

3.7.1 Caching Recent Requests

In addition to avoiding superfluous responses it is also essential to avoid unneeded requests. As stated above, a node is able to process responses to gossip request that it might not have originally requested. Therefore, it is not necessary for several nodes to request the same fragments.

Imagine the case where several nodes require a set of fragments and each repeatedly requests these fragments. This is clearly sub-optimal behavior.

To avoid flooding the system, each node caches recent requests that it has received. When any node is about to transmit a request it first checks that no other nodes have recently requested the same fragment.

The cache of recent requests is reaped appropriately to allow for nodes to eventually issue their own requests in the event that other nodes are unable to obtain the missing fragment.

This component of the RMC is crucial for the scenarios where the system is experiencing a high loss rate.

This particular method of achieving coordination amongst the requesting nodes is directly inline with the overarching strategy of decentralized coordination used throughout the Reliable Multicast Client.

3.8 Cache Eviction

An important factor in the implementation of the reliable client is cache eviction. It is important to free up unnecessary memory use. The eviction of packets from the cache is handled by a reaper thread. Essentially, packets are maintained in the local cache of a node as long as they have received some sort of use recently, by either receiving a fragment or fulfilling a gossip request for another node. Once the packet is no longer being used, then the packet is removed in a bottom up fashion. The reaper thread periodically sees what packets are not being used, testing from the lowest sequence number until it encounters a packet that is still in use. As unneeded packets are removed the corresponding node sequence number range is adjusted.

3.9 Departing Nodes / Unreachable Nodes

Clearly the algorithm should perform well in the event that a node stops participating in the multicast session. The cache eviction

policy is useful in this respect by allowing pending packets for these departing nodes to eventually be removed from this system.

Although cache eviction allows the departing node to gracefully be removed from the system, this alone will not provide good performance. It is easy to see that there are cases where nodes leaving the system are the only nodes caching the data that they have transmitted. This scenario occurs when data has been recently transmitted and the other nodes in the system only receive a small percentage of the transmitted data or perhaps only the sequence numbers identifying the missing packets. In this case, all of the nodes would then continue to request the missing data, thus slowing down the entire system. To avoid this problem, packets use a linear back-off, which is used when a packet has not received a fragment for a significant amount of time (defined by a constant, currently 35 seconds). As more time passes, since the packet has received any responses, the back-off is increased. A packet therefore requests missing fragments less frequently once a substantial amount of time has passed. This back-off is immediately stopped once the packet starts receiving fragments.

4 Design Choices

By choosing to minimize the number of control messages, the client dedicates bandwidth to either transmitting data or gossiping. This choice was crucial in the current performance of the system. Initially a large design choice in the development of the RMC was whether to have two distinct modes of use: Anonymous and Non-Anonymous. The Anonymous mode was initially proposed for networks with a large number of nodes. Essentially, the Anonymous client is very similar in design to the final RMC client produced.

The Non-Anonymous mode was initially suggested for use in small ad hoc networks. Based on the small group size, we thought that we would benefit from transmitting data and having the members of the group *ack* the data. The Non-Anonymous client would have required precise group membership to be maintained. The Non-Anonymous client would have also involved several control messages, not only to maintain accurate membership, but also to allow each client to acknowledge receipt of data being transmitted. The need for a Non-Anonymous client was avoided by simply noting that the local cache can be forced to maintain entries for an extended period of time. The cache eviction policy stated above ensures that all data is delivered reliably and that unnecessary memory usage is kept to a minimum.

5 Results

The Reliable Multicast Client has been fully implemented and is currently being used in the Electronic Student Notebook project.

The client has performed well in our current test runs. Our tests were run on networks of 5 Tablet PC's, with: 1GHz Transmeta processors with 20 GB disks and 256 MB RAM.

For the test runs we ran the Electronic Student Notebook on each of the Tablets. We then participated in a session in which we distributed documents and annotations. The documents ranged in sizes from 25kb (a *one page Word document*) to 1.2mb (an *83 page PowerPoint document*). The annotations being shared ranged from 2-8kb. Our criterion for a successful test run was that all documents and annotations be delivered to nodes participating in the session.

For all test runs, all data was delivered reliably and in-order at acceptable transmission speeds. Twenty five kilobyte documents were delivered in less than 2 or 3 seconds. Larger documents (1.2mb) were delivered in 13-16 seconds.

The delay¹ in delivery is largely attributed to the fragmentation that the ESN performs on documents in order to provide the transfer status to the user.

These times are taken from the moment the document is added to the moment the document appears as being successfully delivered at the receiving nodes. When a document is added for sharing, ESN also caches the document as well as Meta-Data. Additionally, the multicast client is heavily used to send coordination messages to participants in a session. The caching, processing and messaging by the application adds to the perceived time required for a document to be delivered.

Annotations alone were delivered very quickly. We noted that we were able to write text on one client and the text would rapidly appear at the other clients. Therefore, from a users perspective annotations are delivered rapidly.

We did not encounter any cases of out-of-order delivery. All test runs were failure free, no data was lost.

5.1 Stand-Alone Tester

To further test the performance of the Reliable Multicast Client, we implemented a stand-alone tester. (Figure 2) The tester allows us to specify the amount of data to transmit. The tester also prints out data that is received, along with the corresponding

¹ The delay beyond the benchmark times listed in the following section

data length and transmitting host IP. The test client also provides 4 predefined test cases:

1. Session Test- *specified below*
2. Annotation Test- *tests sending 6kb of data 1000 times*
3. Benchmark- *used for data from Table 1*
4. Climbing Test- *first sends 5kb of data and then increases the data transmitted by 250kb increments until 1.5 mb is reached*

Using this tester the RMC was able to achieve the transmission rates specified in Table 1.

Data Size	Time(Seconds)	Transmission Delay(Seconds)
6kb	.25	1
250kb	3.2	5
500kb	6.8	10
750kb	10.3	15
1mb	12.3	20

Table 1

These times were gathered by running 25 simulations in which the 5 nodes in our

experiment transmitted 100 packets of each data size specified above via the RMC. The third column in the table indicates the delay between each transmission. The indicated times are the averages across the 5 nodes throughout the simulations.

Beyond testing data transmission rates for arbitrary data, we also ran a simulation of an ESN session on the stand alone tester. This simulation consisted of each node transmitting the following:

- Four 1mb byte arrays
- Two Hundred 6kb byte arrays
- Four 1mb byte arrays

This particular sequence of data transmissions emulates the typical multicasting pattern in an ESN session, where a user joins, shares several documents, then shares annotations and then finally shares some final documents before leaving the session. For this particular sequence of data transmissions we once again ran several simulations and then averaged out the results across the 5 participating nodes, Figure 1 shows the results.

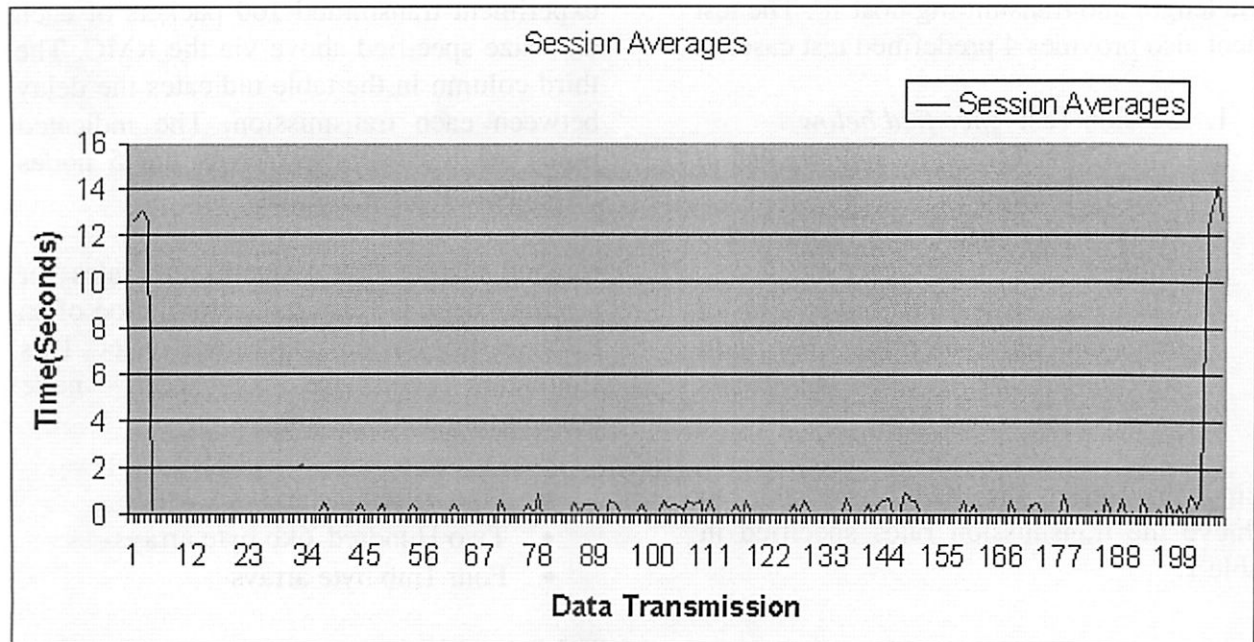


Figure 1

The graph clearly shows that the client performs well for the initial sharing of documents, delivering the 1mb data arrays quickly. (12.2-13.4 seconds)

Once the session begins the various annotation sized (6kb) data arrays are also transmitted quickly. (.01-.55 Seconds)

At the end of the session, we once again see that 1mb data arrays are transferred in reasonable times, ranging from 12.5 to 14 seconds. The slight increase in the time it takes for the 1mb byte arrays to be transmitted in the end is likely caused by the control messages that exist in greater abundance after the system has been running: Gossip Request, Gossip Reply's, Sequence Number Advertisements and Gossip Cancellations.

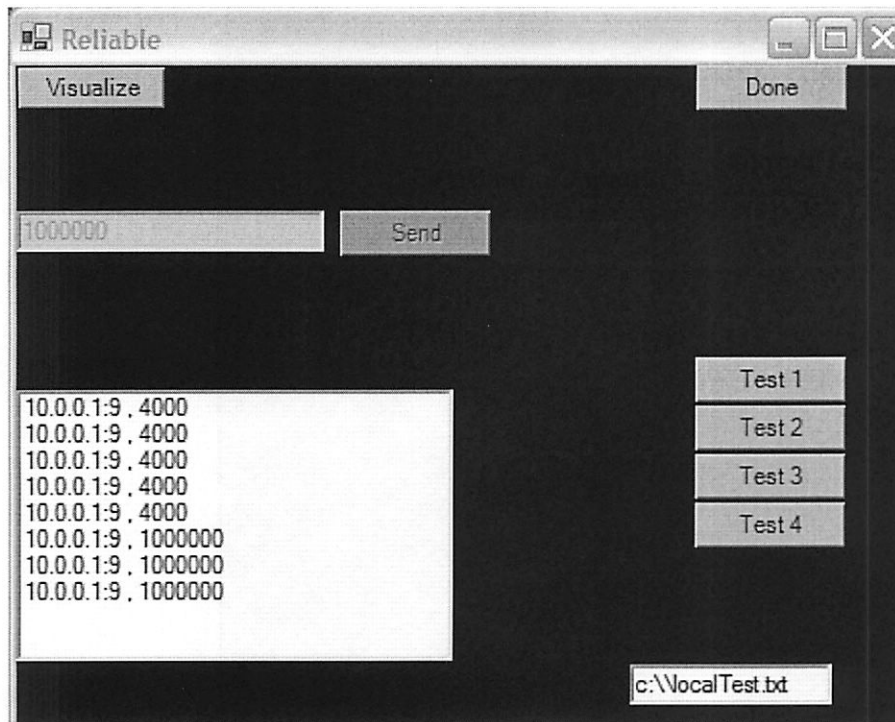


Figure 2: a sample run of the stand alone test client

5.1 Remarks

It is important to note that we tested a modified client that provided faster data transfer using the ESN client, however the client then consumed too many resources for the ESN project to still function properly. This particular modification was the motivation for the fifth goal stated at the beginning of this paper: The RMC should be *Resource Efficient*.

Finally, when using the Electronic Student Notebook it is more useful to use the provided visualizer (Figure 3) to view the performance of the Reliable Multicast Client instead of the file transfer status reported by ESN. The file transfer status only provides an estimate of the amount of data transferred, based on application layer information. The visualizer shows the cache size, member list, probabilistic response information, gossip requests sent, gossip replies sent, user bytes sent and the total bytes sent.

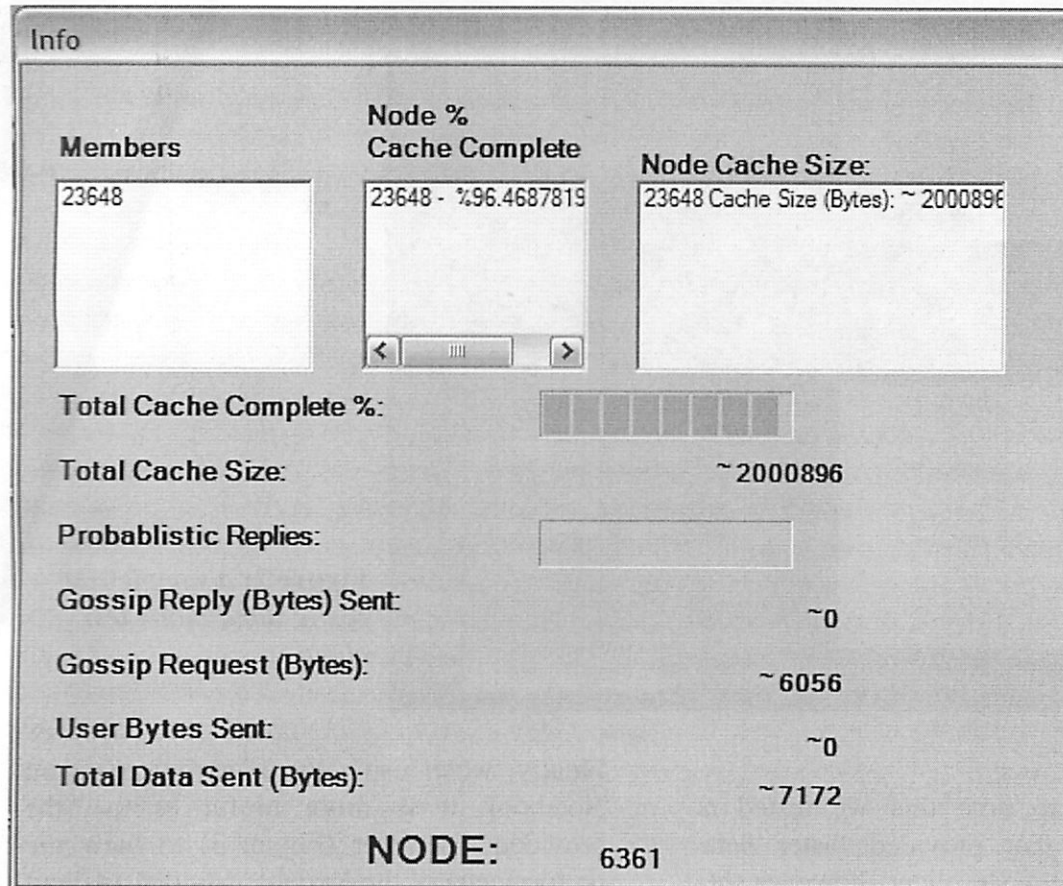


Figure 3: *The RMC Visualizer*

6 Future Work

It would be interesting to derive a more precise mechanism for determining how long one should wait before requesting missing fragments for a particular packet. The current approach is strictly based on the size of the data and when the first packet was received. This is somewhat limiting since a single constant can not easily capture the precise amount of time that one should wait for delivery. A more elaborate approach would definitely require one to take several variables into account, including: traffic, load at the sending site and load at the receiving site.

As the ESN project is deployed for initial testing, it will be beneficial to note the performance of the client, using the visualizer we have provided to determine how to modify some of the constants currently being used.

7 Conclusion

The Reliable Multicast Client is an essential part of the Electronic Student Notebook. The RMC allows users to collaborate through the Electronic Student Notebook without being restricted to areas with a fixed network infrastructure. The Reliable Multicast Client was built to be scalable. The client currently performs well in small networks, and we are sure that it will also perform well for larger networks, based on the design choices we have made.