Multicasting Library Project Description

Luke Hindman and Amit Jain
Department of Computer Science
Boise State University

04/15/2005

The goal of this project is to provide a reliable multicast library for use with Beowulf Clusters. The idea is that programmers will be able to use this library to quickly and efficiently distribute a large amount of data to a subset of nodes within the cluster. Multicasting the data is easy, making it reliable is a bit more tricky.

Background:

From the research I have done in preparing for this project, I learned that multicasting is most commonly used in environments where a small amount of packet loss is acceptable. These include audio and video streaming, slide show presentations, and some types of online gaming. These applications are typically run over routed networks, and so their approach to packet reliability is often hierarchical in nature. If a node misses a packet, it will simply request a retransmission from the next upstream neighbor. This will continue up the network tree until a node is found that has the requested data.

For my library, any amount of packet loss is unacceptable. In addition, this type of hierarchical design doesn't translate well to a high-speed, flat network like the switched gigabit interconnect of our cluster. So I had to keep digging. What I found was that there are primarily two different techniques for adding reliability to projects similar to my multicasting library.

First there is the redundant packet approach called Forward Error Correction (FEC). This technique uses an algorithm to stripe the data packets into slices similar in concept to RAID striping. If a packet is lost, there is typically enough redundant data to recreate the missed packets. I found several software packages that use this technique, including the udpcast library from the SystemImager cluster building suite. The disadvantage of this technique is that nodes that are unable to recover packets just exit rather than send a resend request. To prevent this, the FEC algorithm must be tuned to the specific network and network conditions.

The Second approach to providing reliable multicast on a flat switched network is to use acknowledgments. A block of packets is sent, and each receiving node sends an acknowledgment stating that it received all the packets. If a packets is missed, the receiver asks for the packet to be resent. This is not as fast as using FEC, but it does have the benefit of guaranteeing that all of the nodes will receive all of the data. The downside to this approach is

that if there are a large number of receiving nodes, the sender can easily become overwhelmed with ACKS.
To work around this issue, the receivers can be configured to only send negative acknowledgments (NACKS). This means that they will only send an ACK when they missed a packet. This complicates matters because the sender will either have to send a block of data and wait a specific timeout period for NACKS, or have a separate thread listening for NACKS and notifying the sending thread when one arrives. In addition, the receiving nodes would need to have some means of detecting which packets are duplicates and ignoring them.

As you can see, using NACKS quickly complicates things, and still doesn't prevent the sender from being overrun with ACKS. Just imagine what would happen if 100 nodes all missed the same packet. The sender would receive 100 requests to resend the same packets. A solution to this is to have each node wait a random amount of time and multicast a NACK not just to the sender, but to all nodes within the same mcast group. This way if any of the other nodes missed the same packet, they will know that it has already been requested, so they don't need to send a NACK themselves.

For this library, I initially attempted to code it entirely from scratch. My first version was fast but completely unreliable. The data was multicast whether the receivers were ready or not, and there was no way to recover a lost packet. After much time and effort, my second version was 99% reliable. However, it suffered from SIGNIFICANT performance issues. Then it hit me. I had had the answer for months, but simply didn't have the key to unlock it.

The UDP Cast software package provides support for both acknowledgments and FEC. I've been using it for over a year to deploy node images on my cluster, and for the last few months it has served as a reference to compare functionality and performance. However, since it was designed to transfer data files, I had never considered using it in my library. Then in Operating Systems, we covered a nice little OS feature called PIPEs. This was the key that lead to the creation of the library as you see it today.

Implementation:
The basic concept is simple. The user calls a send function in my library and passes in a void * pointer with n bytes of data. The send function writes the data to a pipe connected to stdin of the udp-sender program. Some black magic occurs, and then udp-receiver writes the data to the pipe connected to it's stdout. The user then calls the receive function in my library which reads from this pipe and writes to the supplied void * pointer.

The benefit of using the Udpcast application in my library is that it drastically simplified my

code, added considerable flexibility to the library, as well as providing a fast and reliable data transport. I'll be giving you the performance details shortly.

Code Layout:

mcast_sender.c

mcast_initsend()

- Creates a pipe, forks a new process, and then connects the pipe to stdin.
- forked process execs udp-sender while the parent returns the pipe file descriptor.

*mcast_send(int sendfd, void *data, size_t size)*

- Writes the size variable to sendfd pipe.
- Writes size bytes from data into sendfd pipe.

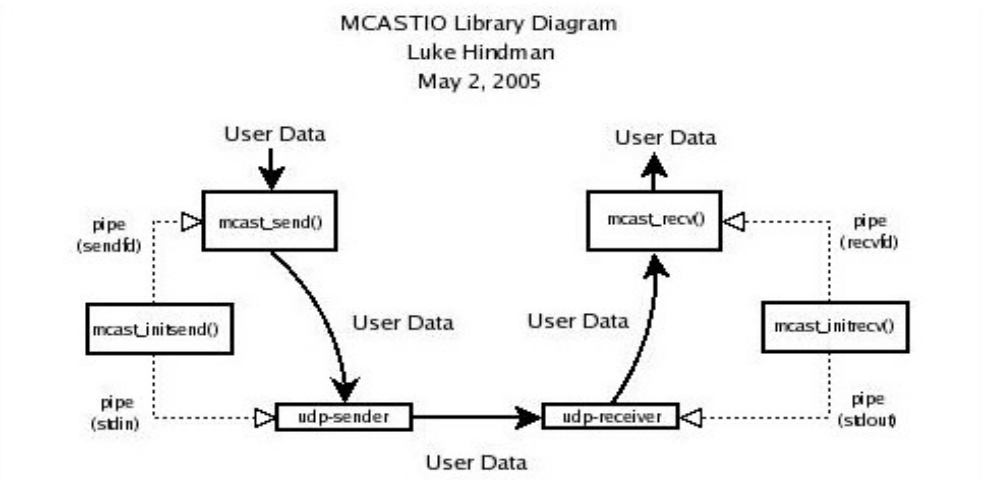
mcast_receiver.c

mcast_initrecv()

- Creates a pipe, forks a new process, and then connects the pipe to stdout.
- forked process execs udp-receiver while the parent returns the pipe file descriptor.

*mcast_recv(int recvfd, void *data, size_t read_size, size_t read_size)*

- Reads the first four bytes of data from recvfd pipe into size variable.
- Reads size bytes from recvfd pipe into data.
- read_size specifies the number of bytes read() should attempt to read from the pipe at one time.



Utilization:
If you've continued reading to this point, you must be interested in this using this library. So you are probably wondering... “What do I need to do to get this multicasting library working with my code?” Well, here you go...

First you must compile and install the udpcast software package, available from http://udpcast.linux.lu/. If installing is not an option, simply compile and copy the udp-sender and udp-receiver binary files from the udpcast src folder to somewhere in your path like ~/bin.

Next you need to compile the multicasting library and ensure that the libmcastio.so and libmcastio.a libraries are in you library path. For me I copy these files to ~/lib and then issue the following command to add ~/lib to my library path:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:~/lib
```

At this point you are pretty well set. Simply include mcastio.h in your header file and link to mcastio in your Makefile. You'll be multicasting in no time.

Just one more note, the mcastio sender and receiver will have to be reinitialized before each use. This is because the associated udp-sender/udp-receiver processes exit after each successful send/receive.

Performance:
Now for the fun part, performance testing. This library began from the idea that it would be nice to utilize the multicast capabilities of the gigabit ethernet interconnect on our Linux Beowulf cluster. The pvm_mcast() function provided with pvm provides a functional way of sending data to multiple nodes at once, however it only sends data to a single node at a time. Therefore, for n nodes, the data would be sent across the network n times.

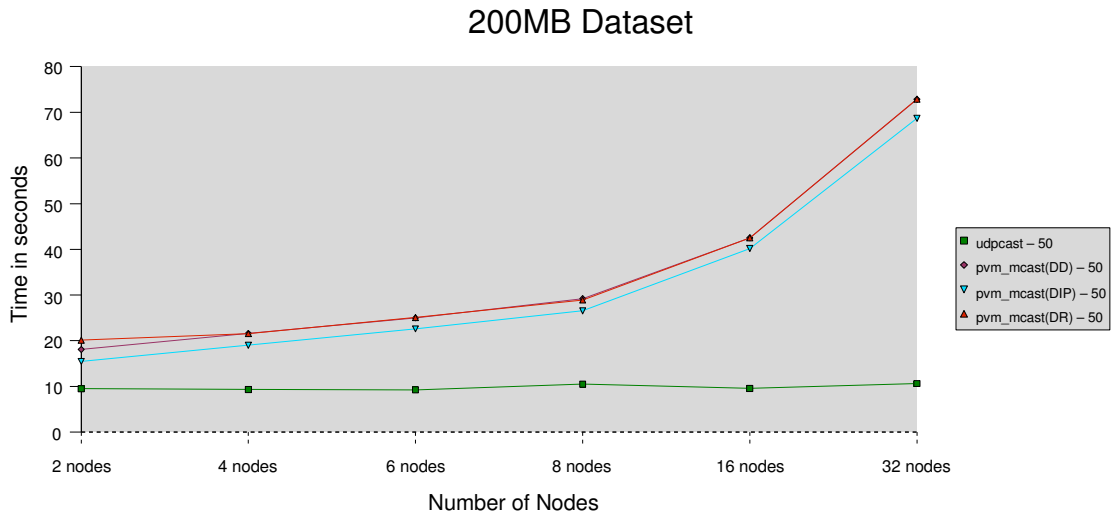
Our concept was that if true IP multicasting were used, the data would only need to be placed upon the network once. This would mean that for n nodes the data would be sent across the network only one time, which in turn should lead to dramatic performance improvements for code currently utilizing the pvm_mcast functionality. This library is the proof of that concept.

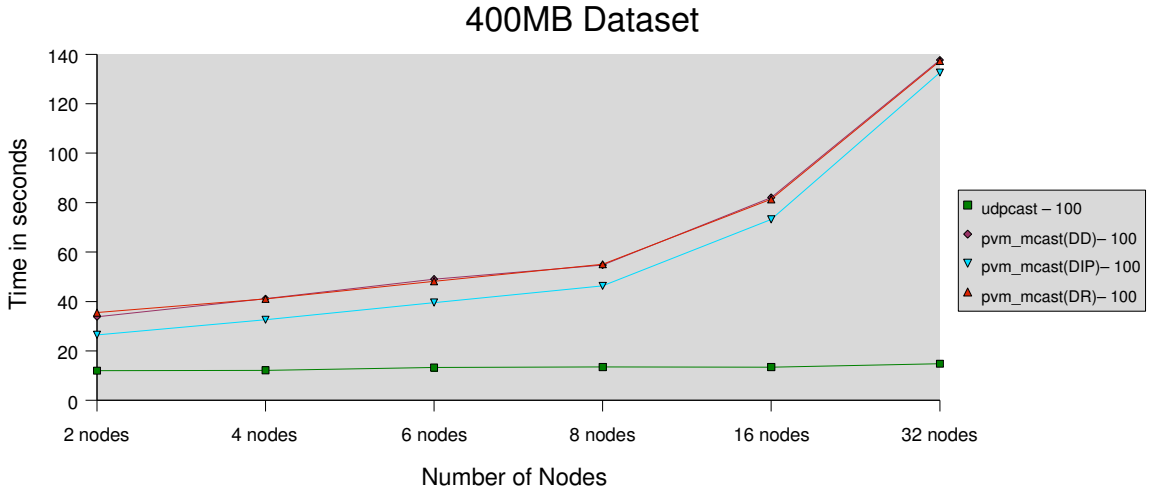
Our test case is a pvm program written in the spmd style. It generates a block of integers and uses pvm_mast() to send this information to all nodes in the cluster. Each node performs calculations on a subset of this data and sends their result to the master, who then combines the results and returns them to the user.

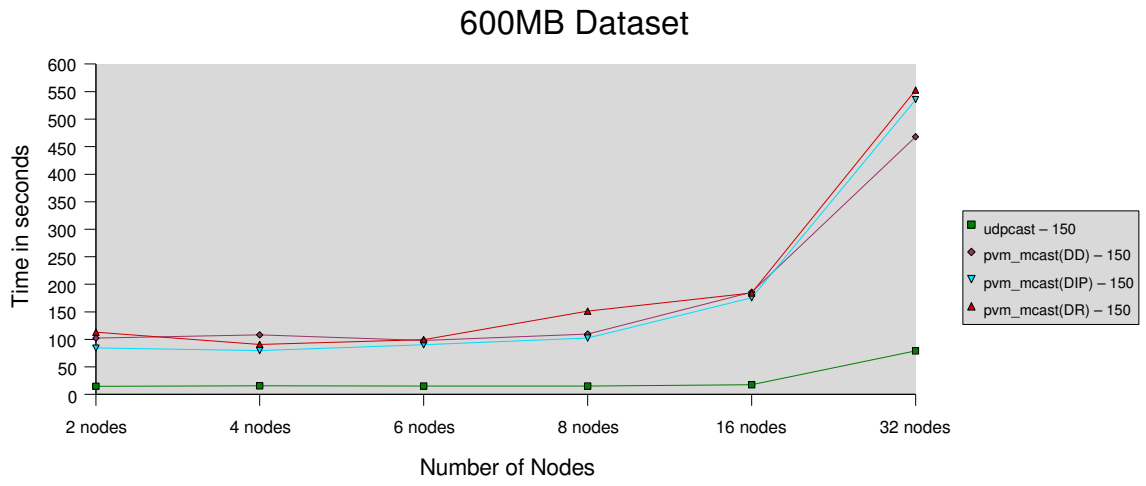
For my testing, I measured the wall time while varying the number of nodes and the size of the dataset. To be fair, I also tested PVM with a couple different configuration options in an attempt to improve performance. These included DataInPlace, DataDefault, and DirectRoute.

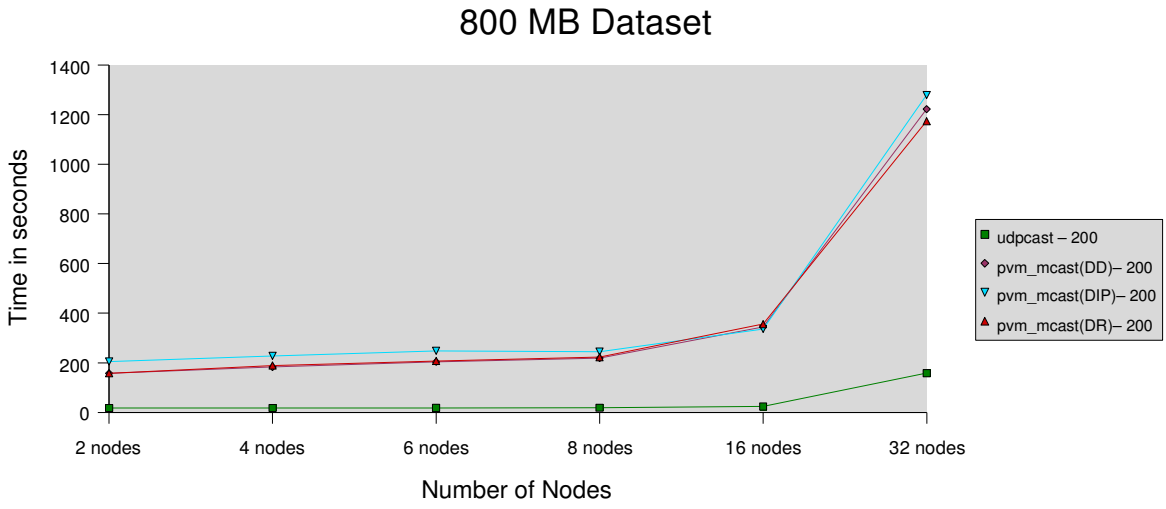
After running the pvm_mcast tests, I replaced the pvm_mcast functions with those from my library and ran the tests again. The following are the results from those tests.

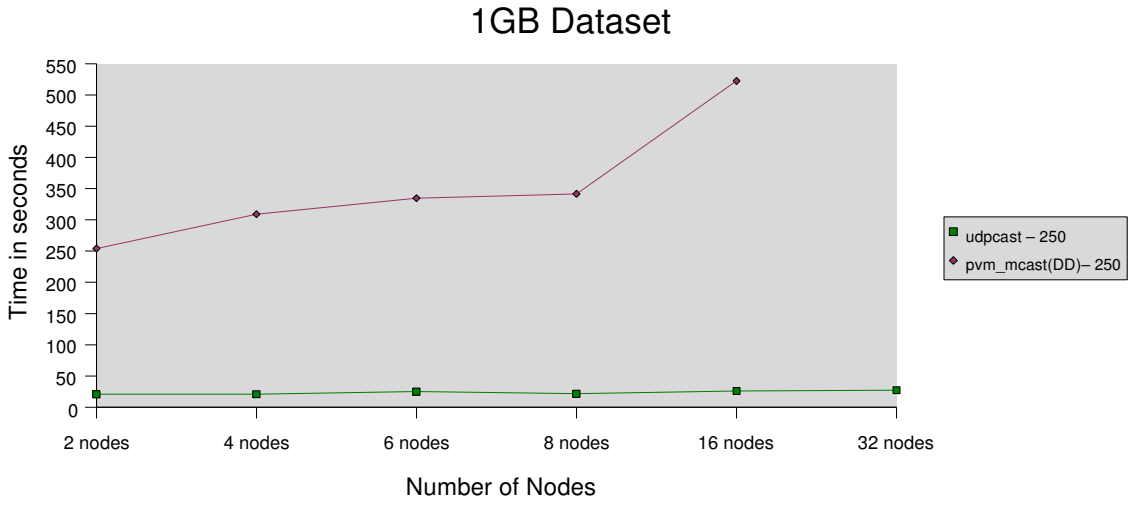
Note: Lower is MUCH better!






600MB Dataset
Time in seconds
600
550
500
450
400
350
300
250
200
150
100
50
0
2 nodes
4 nodes
6 nodes
8 nodes
16 nodes
32 nodes
Number of Nodes
udpcast – 150
pvm_mcast(DD) – 150
pvm_mcast(DIP) – 150
pvm_mcast(DR) – 150


800 MB Dataset
Time in seconds
1400
1200
1000
800
600
400
200
0
2 nodes
4 nodes
6 nodes
8 nodes
16 nodes
32 nodes
Number of Nodes
udpcast – 200
pvm_mcast(DD)– 200
pvm_mcast(DIP)– 200
pvm_mcast(DR)– 200


1GB Dataset
Time in seconds
550
500
450
400
350
300
250
200
150
100
50
0
2 nodes
4 nodes
6 nodes
8 nodes
16 nodes
32 nodes
Number of Nodes
udpcast – 250
pvm_mcast(DD)– 250

Evaluation:
My primary goal in writing this library was to increase performance by only putting the data on the network once. This would in theory allow it to scale to a relatively large number of nodes without significant performance loss. From this graph it is easy to see that as more nodes are added, the runtime for my library remains nearly constant.

What I wasn't expecting was how poorly the pvm_mcast functions performed. From my testing it appears that the pvm_mcast function makes a duplicate copy of the data being sent. This is apparent as the nodes begin swapping with a dataset of only 600MB (each node has 1 GB of Ram). This problem/feature prevented me from completing my 1GB Dataset tests, as the test program would hang and/or die as nodes ran out of RAM and Swap space. (Note: We have since increased the swap space as a result of my testing!) :)

There were a couple instances where one or two nodes would begin swapping when using my library. You can see this on the 600MB and 800MB datasets. I have not been able to determine the exact reason why, but occasionally one or two nodes will began swapping earlier than expected. This did not occur every time, but often enough that I thought I should include it in my results. My library is fast, but it is not perfect.

Conclusion:
The basis for this project was to take advantage of the multicast capabilities of the Ethernet interconnect, providing a faster method than pvm_mcast for distributing large datasets to cluster nodes. From our test results, it appears that this concept is quite promising. However the library will require further testing and development before it will be ready for a production environment.

So, where do we go from here? Well first off is more testing. I am going to test a wider range of dataset sizes, focusing more on the smaller datasets (Up to 200MB). While these aren't necessarily time consuming data transfers, it will be providing a better overall picture of how my library stacks up against pvm_mcast. Is it as efficient for smaller datasets?

I am also making arrangements to have exclusive access to all 60 nodes on the Beowulf cluster for additional scalability testing. The cluster was in use by several other groups while I was running the above tests. This caused a small amount of fluctuation in the results and may have been responsible for some of the swapping that I was seeing.

Finally, I will be compiling a list of needed features. I'll begin with the ability to run multiple instances on a single node. Next will be the ability to define a subset of the nodes as a group and limit the multicast traffic to those nodes. From there, we'll just have to wait and see!