Reliable Multicast tool using UDP/IP

David Gong, Stephen Hamilton

Abstract—Our goal is to develop a multicast system capable of reliably sending packets in agreed order utilizing the UDP/IP protocol. In order to accomplish this task, we will implement a ring protocol where a token is passed in order to establish the control of sending packets. A node will only be able to send packets when the node is in possession of the token.

1 Introduction

THIS multicast ring protocol works by utilizing a token to control the flow of packets multicasted to all processes. There are three stages to this protocol. First is the startup which involves initializing the processes and determining the ring-path for the token.

The second stage is data transmission. Once the processes are initiated, the first process will generate the first token. If that machine has data to send, it will send data up to the maximum flow control value (which is pre-defined but also adapts as the program progresses). Once it has sent the data, it will send an updated token to the next machine. The next process then writes from its received data packets up to the min(token_aru, previous_token_aru). It resends any requests contained in the rtr and it sends the packets it needs to send, updates the token sequence number, rtr, an finally sends the token. And so the cycle repeats. Due to the potential for loss over UDP, this protocol has an additional field on the token called loss_level which determines the number of times a token is sent per attempt from one process to another. This protocol will attemp to adapt the loss_level by detecting when a token is lost. The intent is to increase the reliability of the token by sending it multiple times.

Finally, because properly ending is provably unsolvable, we adopt the method of termination that relaxes the requirement for theoretical, finite termination, but in practical settings will work by drastically reducing the chance of failure after every attempt further.

2 Design

2.1 Assumptions

Before delving into the specifics of our protocol, there are a few assumptions on which the success of our protocol depends.

- Machine ids will be in sequential and continuous order up to the maximum number of machines (10).

With these assumptions, we can describe a successful multicast protocol for agreed data ordering over UDP.

2.2 Token Design

```
// Structure of token.
int type;//Type of packet
int sequence;//Sequence of last message
int aru;//Sequence of all rcv up to
int fcc;//Flow control/Max send size
int rtr[MAX_RTR];//Array of Retransmit
    requests
int rtrcount;//Number of entries in rtr
    array
int loss_level;//Increases on token
    regeneration
int nodata[10];//Process completion flag
int round;//Number of ring traversals
int aru_lowered_by;//Flag for lowering
    the aru
```

The token contains all the necessary data needed to maintain the flow of the protocol. It holds the global maximum sequence number so that senders know what the sequence of the most recently sent packet is. The aru on the token helps with necessary protocol semantics. The fcc aids in controlling data flow. The rtr is used by machines to make requests for missing packets. And nodata is the field used to determine the protocol's termination. Other fields are kept in order to optimize the protocol in the way we thought best.

2.3 Data Packet Design

The data structure we plan to implement is a linked list. This list will contain all the sent packets that are greater than min(token.aru, previous token.aru). It will be implemented in a struct as follows:

```
struct packet_structure {
int type;//Type of packet
int sequence;//Sequence of packet
int received;//Flag for when globally
    received
int machine_index;//Index of packet's
    sender
int packet_index;//Index of packet from
    the sender
int random_number;//Data
char data[packet_size];//Contents
}
```

The packet structure contains the overall token sequence, whether or not the packet was received, the machine_index of the sender, the packet_index of the sender's sequence, the random number, the 1200 additional payload bytes, along with the pointer to the next packet.

2.4 Data Structure

This protocol uses one main data structure per machine in order to control the information flow for each machine.

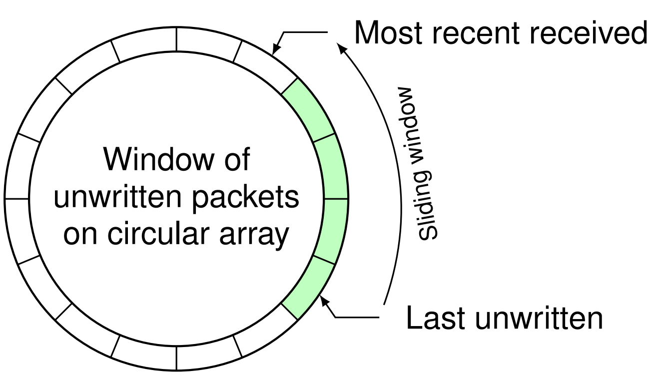

Fig. 1. Representation of the circular array used to store unwritten packets.

Each machine has a circular array (as in Figure 1). The array holds all of the received packets that haven't been written. The packets can only be written if their sequence value is lower than the minimum of the previous token aru and the current token aru. After the packet is written, the slot in the array is freed and the window can progress. We need not worry about filling and writing over the maximum window size because we have the fcc to control data flow.

2.5 State Machine

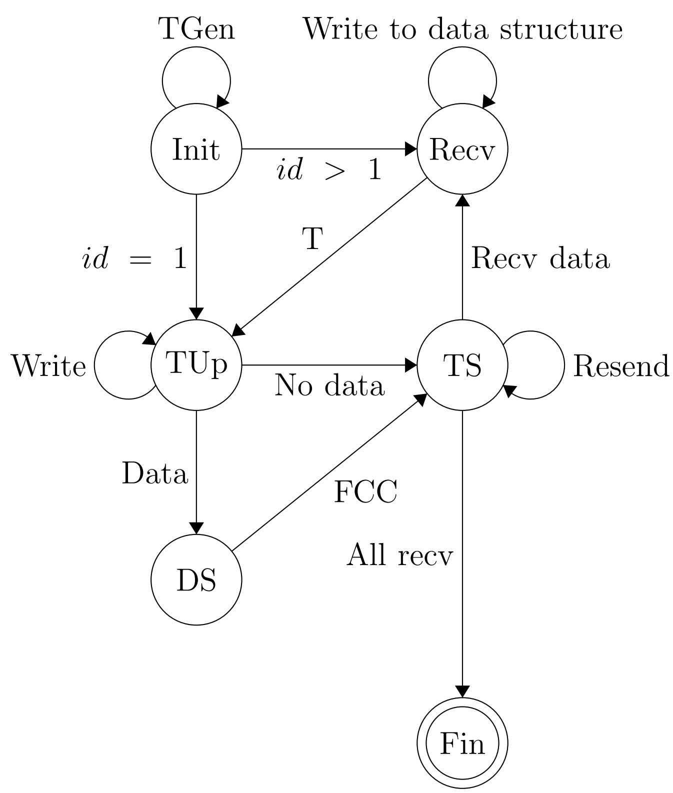

Fig. 2. A general state machine for the program from the perspective of a machine. T is for token, D is for data, S is for send, Up is for update.

2.6 Algorithm

We dedicate a section to explaining the details of the protocol. As stated before, the protocal is split into three main parts: startup, message transmission, termination. The general flow of our protocol is shown in Figure 1.

The startup is difficult because we must find a way to successfully establish the ring through which the token is passed (using unicast). Our approach to this is a simple ack routine. Every machine starts out by multicasting its own information so that everyone (ncluding the target) receives the information. In the meantime, every machine also filters through the received packets, looking for its target (the machine with the next sequential ID). Once it successfully receives the information, it sends a message to the target so the target knows to stop. If the machine receives conformation, then it sends a message to machine 1 (who is essentially the controller). Until all machines are done, machine 1 will not create the token therefore the transmission process will not start. Once all acks are received, the process starts with machine 1 sending data (note, once the other machines start receiving data packets, they know for sure that they're in the messaging stage).

The transmission proceeds as discussed in the introduction essentially covers everything. A note is that flow control is determined by how often various limits are reached. For example, if the rtr ever reaches a state of being completely full, then we lower the fcc meaning that people must send less (giving more opportunitites time for the machines to catch up).

Finally, termination is done by sending a mass of tokens before finaly closing. That way the probability of none of them being received is 0 in a practical setting.

3 Results

Trial \ Loss	0%	1%	2%	5%	10%	20%
1	8.00	8.63	8.60	10.24	11.98	17.79
2	7.99	8.58	9.06	10.22	11.92	18.70
3	7.96	8.50	8.95	10.21	11.97	17.89
4	7.97	8.64	9.08	10.19	11.99	16.49
5	8.08	8.54	8.98	10.18	11.98	15.75

TABLE 1
5 time trials in seconds, each of varying % packet loss.

Table 1 and Table 2 are the results of the benchmarks we ran on our protocol. We ran our protocol with simulations of loss at 0%, 1%, 2%, 5%, 10%, and 20%, 5 times each. The average of those times is shown in Table 2.

% Loss	Average time (s)
0	8.00
1	8.58
2	8.93
5	10.21
10	11.97
20	17.32

TABLE 2
Average times for each level of packet loss.

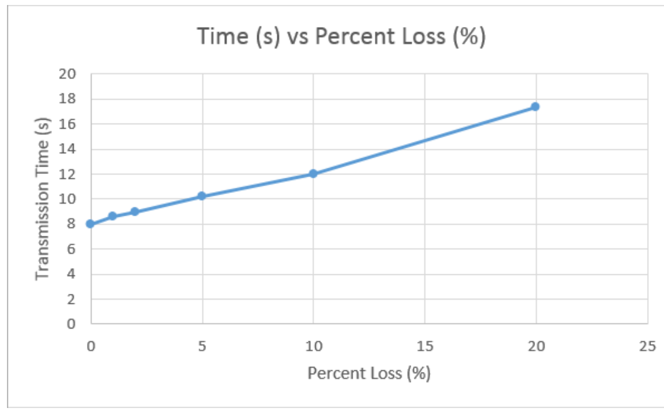

Fig. 3. Graphical interpretation of results

This is the data of Table 2 displayed as a graph.

4 Conclusion

Our results are rather pleasing. At first glance, the graph looks rather respectable. As loss grows, the program's degredation seems close to linear, which would be ideal. Of course, the goal was to get the slope as flat as possible. And especially for the ring token protocol, this was a big worry. Losing the token takes a huge toll on running time if not properly dealt with because without a token, no data can be sent at all. But this data is evidence that our protocol can fairly gracefully recover from token loss. There is obviously still room for improvement as our transmission time slightly more than doubles when going from 0% loss to 20% loss.

Doing some simple math, we can roughly calculate the Mibps of data transmission for our protocol. We send a total of 600,000 packets which is about 5760 Mib. Under 0% loss it takes around 8 seconds so the transfer rate is around 720 Mib/s which is fairly decent. Clearly this degrades as loss grows – at 20%, the transfer rate is around 332 Mib/s which is not great but not abhorrent.

5 Discussion

The token ring protocol we implemented transfers its messages respectfully well. Even under pretty big loss, we manage to get the messages across in a respectful amount of time. Some improvements we could make to our protocol would be to take the next step and do the data transmission after updating/sending the token (advanced token ring protocol). But in general, the main place we can make improvements in is tweaking the data structure size, modifying the retransmission request size allowance, and playing with the method of evolution of the fcc while the protocol runs. These small changes could potentially have a large affect on preventing the protocol from hitting periods of lag, for example, when one process gets too far behind the others.