Burghers Design Document

John Bent Brandon Schwartz Trey Cain

January 23, 2001

1 The Big Picture

This paper is the design specification for our implementation of a reliable multicast network and transport layer over a datagram network. It will discuss the various protocols needed, their relationships to each other and to the network as well as a discussion of the human factor involved (i.e. division of labor and timetable).

1.1 Protocol Threads

The five protocols are implemented via threads and blocking functions. Each protocol is described more fully in sections 2–6. An overview of the use of threads by various protocols is given in the following table:

Protocol	Thread Name	Usage
DNP	DNP Receiver	Handles the arrival of packets, forwards them to other threads.
GMP	GMP Receiver	Handles the lists of local and downstream clients for each multicast group
DRP	DRP Receiver	Handles incoming packets passed by DNP.
	Hello Handler	Periodically sends hello messages to neighbors and prunes dead nodes.
	LSP Handler	Reliably floods link state packets.
RAP	RAP server	One per RAP server; implements RAP state machine.
RMTP	RMTP Middle layer	one per node; receives packets from DNP and forwards them to the appropriate RMTP client thread. Also handles pruning of RMTP control packets.
	RMTP client	one per RMTP client; implements RMTP client protocol.
	RMTP server	one per RMTP server; implements RMTP server protocol.

1.2 Memory Usage and Inter-Thread Communication

Communication between these threads is channeled through a producer-consumer buffer. The DNP consists of a receiver thread at each node which takes packets from the network via a call to `recvfrom()`. After comparing the packet's computed and delivered checksums, the DNP receiver thread reads the header field to determine the protocol for which the packet is intended and adds a pointer to that packet to a buffer between DNP and the receiving protocol. Each thread waits for packets to enter this buffer and removes the packet references from this buffer in FIFO order.

A buffer between these protocol threads helps avoid packet loss during burst periods of network traffic in which a receiver thread may be busy yet need to store incoming packets. Sustained bursts may still result in packet loss but in the absence of an infinitely large buffer, this will always be the case. By setting the buffer size well, an acceptable tradeoff between probability of packet loss and memory requirements can be achieved. Currently, the majority of our protocols use buffers that can hold one hundred packet references.

Because the DNP plays such a vital and immediate role in the network, it is important that the DNP not be forced to wait on the other protocols when the buffer between the two protocol threads becomes full. For this reason, we have modified the typical producer-consumer relationship. In our implementation the producer (the DNP) will never wait on a full buffer; instead, it will remove the oldest item in the buffer.

To avoid the expense of redundant copying, the DNP allocates memory space for the incoming packet, and a pointer to the packet is stored in the buffer. The receiving protocol then becomes responsible for freeing the memory after it has processed the packet. Reference counting is used when multiple threads share references to the same buffer, and the last user is responsible for freeing the packet's memory space.

1.3 Optimizations

We have discussed further ways (currently unimplemented) in which to optimize this system. One is to use `realloc()` to truncate packets to their minimum size, which would reduce memory usage for those packets which do not use the full length of the DNP packet. Another optimization would be to move the DNP header information to the rear of the packet. The reallocation optimization could then further reduce memory usage by truncating the portion of the DNP header which is not of interest to the receiving protocol.

1.4 Division of Labor

The division of labor for this project has been done as follows: John has been completely responsible for the DRP protocol, Trey for the GMP protocol, and Brandon for the RAP and DNP protocols. The implementations of these protocols are in a fairly complete state. Due to the complexity of the RMTP protocol, we are jointly working on its implementation.

1.5 Timetable

We hope to have completely finished this project (including the ability to handle the garbler and link/node failure scenarios) by the date of the initial demonstration (November 23). We will then be able to focus our remaining time on some of the performance enhancements described in this document.

2 Datagram Network Protocol (DNP)

The Datagram Network Protocol (DNP) is responsible for the actual receipt and transfer of incoming messages for the other protocols. A decision was made by the BURGHERS to keep DNP as simple and efficient as possible due to its important position in the critical path of all protocols. Therefore, DNP is responsible for a limited number of items.

DNP's functionality is divided into two parts. A detached thread is responsible for the distribution of incoming packets to other protocols. DNP also provides a service interface (function) to the other protocols for disseminating outgoing packets.

DNP reserves some fields in the packet for its own use. Specifically, DNP uses the first ten bytes of each packet for special fields:

Name	Byte offset	Data type	Comment
`checksum`	bytes 0-1	short	
`destination address`	bytes 2-3	short	This is the 16 bit node address of the destination SAP or node.
`sender address`	bytes 4-5	short	This is the 16 bit node address of the source SAP or node.
`flag`	bytes 6-7	short	This field distinguishes the packet protocol type. A listing of the values of FLAG is provided below.
`length`	bytes 8-9	short	This field holds the length in bytes of the packet. Packets are limited to 1024 bytes

When the DNP thread is instantiated, it sets up the network socket for the local node based on the local machine IP address and port, and then executes the following infinite loop:

1. A buffer of size 1024 is allocated with a call to `malloc()`.
2. Block in recvfrom() until a message has been received.
3. Check the message for transmission errors by comparing the received length with the actual (described-in-packet) length, and the computed checksum with the actual (described-in-packet) checksum. If a packet fails either of these tests, it is dropped with a warning and the buffer is freed.
4. If the packet is valid, then a handler function for the appropriate protocol is called based on a flag in the packet. Possible values for the flags field of the DNP header are:
   (a) RMTP_DATA_PACKET — indicates the first sending of an RMTP data packet. (i.e. not a retransmission)
   (b) RMTP_RETRANSMIT — indicates any subsequent RMTP data packet. (i.e. the response to a NACK)
   (c) RMTP_NACK — indicates a client request (bound for the server) for a retransmission.
   (d) RMTP_MAB — indicates a client or node advertisement of the Maximum Acceptable Byte the server may send.

(e) RAP_PACKET — indicates a packet to be handled by the RAP, either request or response.

(f) DRP_PACKET — indicates a packet to be handled by the DRP. (e.g. link state packets, HELLO packets)

(g) GMP_PACKET — indicates a packet to be handled by the GMP. (e.g. add_group(), leave_group())

(h) If the packet has an invalid flag, it is dropped with a warning and the buffer is freed.

5. GOTO step 1.

The DNP provides the function `int send_DNP_packet()` for other protocols to use to send packets. This function:

1. Extracts the length input by the calling protocol (which could be any protocol,)
2. Corrects the length to the appropriate word boundary, (0-padding if necessary,)
3. Inserts the checksum,
4. Figures out the destination socket_address by calling a lookup() service provided by the DRP (Dynamic Routing Protocol) based on the destination node address.This step may be short-circuited in the implementation if the calling thread already has knowledge of the destination socket_address (e.g. DRP),
5. Calls `sendto()`,
6. And finally, checks to see whether the packet was actually sent.

`int send_DNP_packet()` returns TRUE (non-zero) if the packet appears to have been sent, and FALSE (zero) if there was an error.

3 Dynamic Routing Protocol (DRP)

The Dynamic Routing Protocol maintains an accurate representation of the network topology. When a node is booted, the DRP uses the information contained in the configuration file to construct its data structures. The `network_node` structure contains all of the information that the routing protocol will need to make routing decisions and is shown here:

```
struct
network_node {
/* Identification and link state variables */
    unsigned short addr_sap;             // SAP address of this node
    struct sockaddr_in addr;             // socket address of this node
    unsigned long  last_lsp_recv;        // seq. num of last lsp from this node
    struct timespec last_hello_recv;     // time last hello recv from this node
    unsigned short neighbors [MAX_LINKS];// list of neighbor's SAP's
    struct network_node * neighborPtrs [MAX_LINKS];
    unsigned short neighborCount;        // how many neighbors
    struct reader_writer_lock node_lock;
/* Info relative to local node */
    unsigned short isNeighbor;           // is it one of our neighbors
```

```
    unsigned short last_lsp_acked;       // has this node acked our last lsp
    unsigned short forward_addr_sap;     // SAP address of how to forward
    struct sockaddr_in * forward_addr;   // socket address for forwarding
/* Shortest path variables */
    unsigned short known;                // is shortest path yet determined
    struct network_node * predecessor;   // who comes before
    int distance;                        // how many hops to get there
/* Linked list variables */
    struct network_node * next;          // pointer to next node
    struct network_node * prev;          // pointer to prev node
} * network_topology;
```

3.1 DRP Routing Table

After the network topology is read from the configuration file, the DRP will create its routing table. The routing table is built using two parallel arrays:

```
struct network_node ** nodePtrs;        // easily find each node

unsigned short * nodePtrIndex;      // index into array of node ptrs
```

The nodePtrIndex array contains a sorted list of every node's virtual address. The index of the node's virtual address corresponds to the index within the nodePtrs array. A routing table lookup entails a binary search of the nodePtrIndex array and using the index provided a lookup into the nodePtr array. [1]

3.2 DRP Threads and Packets

After the initialization of the network topology and the routing table, a DRP thread is launched at each node which is responsible for maintaining this topology. Currently, the DRP splits its responsibility into three separate logical divisions each of which is handled by a separate thread: handling hello packets; handling link state packets; and receiving and dispersing all packets. (A possible optimization to avoid thread overhead might be to combine some of these threads.) Each packet contains the DNP header and possibly some additional information. The following table lists the various DRP packet types and their fields:

[1] If we were to know that all virtual addresses would be sequential and would start at some defined constant (e.g. 1), then we could put our routing table into one array only and do an immediate lookup as opposed to doing a binary search in our nodePtrIndex array. However, there is no assurance of this and we use two dynamically allocated arrays of size corresponding to the number of nodes. An example of the situation we are avoiding is the simple case in which there exist only two nodes, one of whose virtual addresses is 1 and the other whose virtual address is 30,000. In this case, the naive implementation would create a routing table array of size 30,000 to hold pointers to only two nodes. Our implementation will instead create two arrays of the minimum size two.

Type	Additional Contents
Hello	Contains DNP header only.
Terminate	Contains DNP header only.
Link state packet	Originator node, sequence number and list of neighbors.
LSP_ack	Originator node and sequence number.

The **DRP Receiver** thread is responsible for incoming packets and is the most straightforward of the three DRP threads. It pulls packets from the buffer between the DRP and the DNP and handles each accordingly.

- If the incoming packet is a hello packet, the receive thread remembers the current time. If the sender was not previously considered a neighboring network node, this change is made to the network topology and the link state packet thread is notified that the topology has been changed.
- If the incoming packet is a link state packet, its sequence number is checked. If it is not a new link state packet, it is acknowledged and then dropped. If it is new, its information is substituted into the network topology and the link state packet thread is notified that the topology has been changed.
- If the incoming packet is a link state packet acknowledgement, the receive thread marks that this packet's sender has acknowledged this particular link state packet.
- If the incoming packet is a terminate connection packet, the receiver thread removes the sender node from its network topology and alerts the link state packet thread that the topology has been changed.

The **Hello Handler** thread cycles through its list of neighbors, sending each a hello message and checking the last corresponding hello received from each. If the last hello message received from node X is outdated, then this node removes X's "neighbor" designation and the link state packet thread is notified of the topology change. The hello handler then enters a pthread conditional wait after which it will repeat this cycle. The hello handler can be awakened by a DRP_emergency conditional variable, but this functionality is added as a contingency only and is not currently used. The timeout frequency is set at 1 second and nodes will be removed as neighbors if a hello message is not received within the last three cycles (these numbers may change depending on experimental efficiency findings).

The **Link State Packet Handler** thread is similar to the hello handler thread in that it cycles through the topology, sends appropriate packets to its neighbors and then waits either to be alerted of a topology change or to be awakened by a timeout and repeat the cycle. Each incoming link state packet is forwarded reliably to each neighbor of the node (excluding the original sender). So, at some regular interval (currently set at one second), the link state packet thread will check whether the last received link state packet from each and every node has yet been acknowledged by each of its neighbors. The link state packet is then forwarded to each neighbor which has not yet acknowledged it.

3.3 DRP Interface

The DRP provides the following interface for use by the other protocols:

```
void pass_packet_from_DNP_to_DRP (char * packet);
struct sockaddr_in * next_hop    (unsigned short destination);
```

The `pass_packet_from_DNP_to_DRP` function is the means by which the DNP passes packet pointers to the DRP. This buffer is implemented as an impatient producer-consumer buffer as described in section **??**.

The `next_hop` function does a routing table lookup and returns the socket address of the node to which packets for the queried destination should be forwarded.

3.4 Routing Loops

As described, this routing protocol should not introduce any routing loops into the topology. As all nodes remain dynamically aware of the topology, the shortest path algorithm in which each hop has equal cost will plot the same path at every node for all combinations of source and destination. It is possible to imagine unusual circumstances in which an unstable link, fluctuating between up and down, can temporarily confuse the shortest path algorithm such that packets do enter a temporary loop. But this is extremely unlikely and would not last long.

3.5 Routing in an Untrustworthy Environment

Our routing protocol as described above can potentially fail under two extreme circumstances. The first of these to consider is when a connecting link between neighbors becomes unusable for a long enough period of time that each node considers the other to be lost. In this case the nodes can never again discover whether the link becomes usable without being rebooted. The other situation is the one in which a node is given an incomplete configuration file. This node will know only the network topology described in its own configuration file and will not dynamically discover the rest of the network until every other node happens to experience its own topology change thereby triggering a flooding of its link state packet.

To prevent these extreme circumstances from adversely affecting our network's performance, we have defined two additional behaviors for the DRP. The first of these is that every node will maintain a list not

only of current neighbors but will additionally maintain a list of former neighbors and will periodically (with less frequency than normal hellos) send hellos to all former neighbors in an attempt to discover if a bad link has been restored.

This second problem of a node being booted with an incomplete or inaccurate configuration file is actually addressed by the above described behavior. Because every node will periodically reforward link state packets to all neighbors which have not yet acknowledged those link state packets, a new node need only register as a neighbor of a "network-aware" node. When this happens the "network-aware" node will then forward all of its previously received link state packets to this new neighbor. A node will also send to its neighbors the link-state that it creates for each node from the configuration file if it has not received a link state update from that node. The sequence number for a configuration file derived link state packet is zero. By sending these zero sequenced number link state packets, new neighbors can learn of any nodes which may not have been included in their own configuration file.

Another consideration is a node's behavior when it dynamically discovers either new neighbors or new nodes all together. Our routing protocol handles situations in which a known node which was not thought to be a neighbor suddenly becomes a neighbor. What is not implemented (and what may not be implemented should it not prove necessary) is DRP's behavior upon discovering dynamically the existence of a new node.

4 Group Management Protocol (GMP)

The Group Management Protocol runs in its own thread. Its job is to manage multicast address subscriber lists for both local clients as well as downstream clients of a multicast. The state machine for the group management protocol is shown in figure **??**.

These are the contents of a GMP packet:

Name	**Byte offset**	**Data type**	**Comment**
`DNP header`	bytes 0-9	N/A	The first 10 bytes is reserved for the DNP header.
`multicast address`	bytes 10-11	`short`	The multicast address to which this join or leave packet corresponds.
`join`	bytes 12-13	`short`	This is a boolean field that designates the GMP packet as a join or leave packet.

The data structure GMP uses to keep track of the local and downstream clients of a multicast is a linked list with one node (`struct Multicast_Table_Entry`) per multicast address to which this node is currently

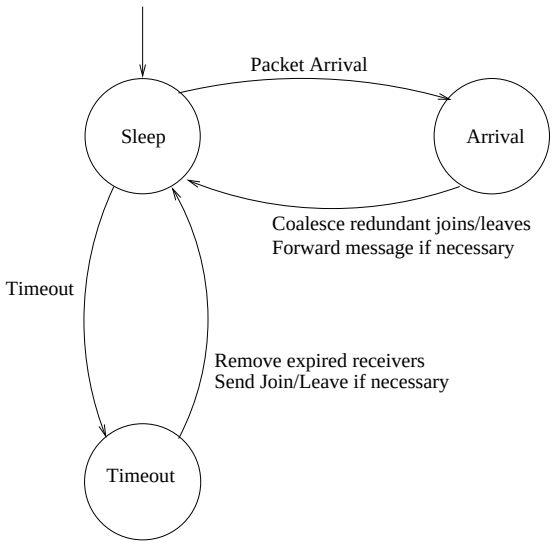

Figure 1: GMP state machine

subscribed. Each node in this list contains two other linked lists: the first is a list of downstream neighbors (`struct addr_list`) that subscribe to this multicast address, the second is a list of local clients (`struct client_list`) that are subscribers to this multicast address. Here is the code that defines this data structure:

```
struct addr_list {           //Nodes in list of downstream subscribers
  unsigned short address;    //address of downstream node
  time_t expiration_time;    //timestamp at which this client will be pruned
  struct addr_list *next;
};

struct client_list {         //Nodes in list of local clients (SAPs)
  unsigned short service_id; //service id of local client
  struct client_list *next;
};

pthread_mutex_t multicast_table_mutex;
struct Multicast_Table_Entry {
  unsigned short multicast_address; //Multicast address associates with subscribers
  unsigned short sender_address;    //Address of multicast sender

  pthread_mutex_t addr_list_mutex;
  struct addr_list * forward_list;  //list of downstream clients

  pthread_mutex_t client_mutex;
  short client_count;
  struct client_list * client_list; //list of local clients

  struct Multicast_Table_Entry *next;
} *multicast_table_head;
```

The GMP thread begins its life in the sleep state. It waits on a condition variable that is used to monitor the buffer between the DNP thread and the GMP thread. When the DNP thread adds a packet to this

buffer, the GMP thread is awoken.

Upon the arrival of a packet, the GMP thread looks at the multicast address specified in the packet. Using the multicast address table, we look for the addresses of our downstream neighbors for this multicast.

If the GMP packet is a join packet, and we already have a downstream neighbor for this multicast, we add the sender's address and a current timestamp to the multicast address table. If the GMP packet is a join packet, and we do not already have a downstream neighbor, we add this address to the list and send a join packet to the upstream neighbor for this multicast.

If the GMP packet is a leave packet, and there is more than one downstream neighbor in the list, then we simply remove the sender's address from the list of downstream neighbors for this multicast address.

If the GMP packet is a leave packet, and there is only one downstream neighbor in the list, we remove the sender's address from the list, and also send a leave packet to our upstream neighbor.

When a timeout occurs, the GMP thread wakes up. The first thing that it does is look at the list of multicast addresses for which this node has downstream clients. For each address in this list, we examine the timestamp of each downstream neighbor for this multicast. If the timestamp has expired, the downstream client is removed from the list. If the list is now empty, we send a leave message to our upstream neighbor for this multicast address.

Currently, we are setting the time duration after which GMP subscribers are pruned to ten seconds. The GMP join refresh packets are being sent every second. This means that at least ten consecutive join packets must be lost before a multicast subscriber will be erroneously pruned from the multicast tree.

Once we have pruned all of the stale nodes for each multicast address, we send a refresh join message to the upstream neighbor for that multicast address. The GMP protocol provides the following interface for use by other protocols:

```
void pass_buffer_from_DNP_to_GMPthread(char *buffer);

int local_client_subscribe_to_multicast(unsigned short multicast_address,
                                        unsigned short sender_address,
                                        unsigned short service_id);

int local_client_unsubscribe_from_multicast(unsigned short multicast_address,
                                            unsigned short sender_address,
                                            unsigned short local_service_id);
```

The `pass_buffer_from_DNP_to_GMPthread` function is used by the DNP thread to place incoming packets in the queue between the DNP and GMP threads, and wake up the GMP thread if it is sleeping.

The `local_client_subscribe_to_multicast` function is used to notify the GMP thread that a new local client will be subscribing to a multicast address. It is called by the RMTP protocol. If the join request is successful, 0 is returned. If the join request is redundant (the local SAP is already a subscriber), a -1 is returned.

Conversely, the `local_client_unsubscribe_from_multicast` function is used to notify the GMP thread that a local client is leaving a multicast. If the leave request is successful, 0 is returned. If the leave request is for a multicast client that is not currently a member of the multicast group, -1 is returned.

5 Reliable Announcement Protocol (RAP)

Information about available multicasts is gathered by the Reliable Announcement Protocol. The RAP is the means by which clients can obtain a list of available services from any server. The contents of the RAP packet are:

Name	**Byte offset**	**Data type**	**Comment**
`DNP Header`	bytes 0-9	N/A	The first ten bytes are reserved for the DNP header.
`destination SID`	bytes 10-11	`short`	Stores the service id of the RAP packet destination
`source SAP SID`	bytes 12-13	`short`	Stores the service id of the RAP packet sender
`RAP_ID`	bytes 14-15	`short`	Session id for this RAP session (described below).
`payload`	bytes 16-1023	`char[]`	Optional payload field used in RAP announcements. It is an ASCII null terminated string.

When a RAP client wishes to know the services offered by a RAP server (whose service_id and address it already knows), the client sends an RAP_Request_Info packet to the server. When the server receives this packet, it assigns an ID number to the request, and replies with a RAP_Provide_Info packet containing the list of its services. The client receives this response and ID and sends an ACK containing the same ID back to the server. When the server receives the ACK it deletes the state associated with the pending client request.

The guarantee of reliability is wholly implemented on the client's side. After a client sends the RAP_Request_Info packet, it sets a timeout and resends the packet if it has not received the corresponding RAP_Provide_Info packet. Note that the client will do this as long as the routing protocol, DRP, provides a path to the server. The protocol state machines depicted in Figures 2 and 3 demonstrate these client-server interactions.

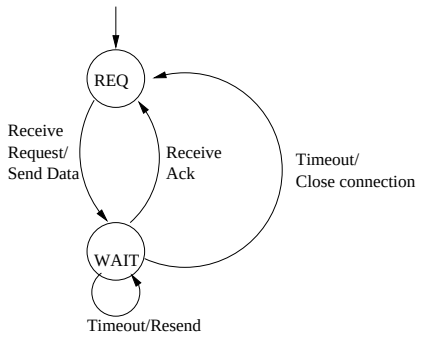

Figure 2: RAP server state machine

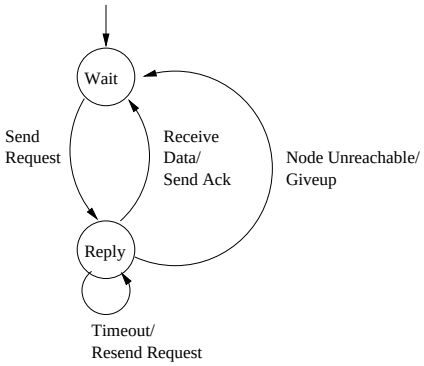

Figure 3: RAP client state machine

6 Reliable Multicast Transport Protocol (RMTP)

The functionality provided by the RMTP is divided into two abstract parts. We will refer to the part of the RMTP which is invoked at every node[2] we will refer to as the RMTP Middle Layer. The functionality of RMTP at the client and server SAPs will be treated separately.

This separation of concerns leads to the distinction between local *client* members of a multicast group and remote *node* members of a multicast group. Each node with one or more clients subscribed to multicast group itself becomes a member of the multicast group. Each node needs to remember both the local clients to whom it needs to pass incoming packets, and the downstream nodes to whom it needs to forward the incoming packets.

[2]Each node must be capable of handling aggregation of NACKs and MABs, and distributing RMTP data and control packets.

6.1 RMTP Middle Layer

When the DNP at any node receives a packet with type RMTP_NACK, RMTP_MAB, RMTP_RETRANSMIT, or RMTP_DATA_PACKET, that packet is passed (by reference) via a buffer mechanism to the RMTP middle layer thread. The RMTP middle layer thread waits in an infinite loop for these packets to arrive. This "middle layer" functionality avoids forcing the DNP thread to spend time dealing with RMTP distribution issues.

6.1.1 Data Packets

If the packet is of type RMTP_DATA_PACKET, the thread looks up the list of nodes whom are subscribed to this multicast address in the multicast table (maintained by the GMP thread). The packet is first distributed to the downstream nodes via a list traversal and repeated calls to DNP_send_packet().

Once the packet has been distributed to the downstream nodes the packet is passed (via reference) to each client at the local node who is waiting for a copy. A reference count is kept so that the space allocated to store the packet is freed only after each client has commited the contents to disk.

The same process occurs for RMTP_RETRANSMIT packets except that the lookup table is not the multicast table (maintained by GMP) but a NACK table (maintained by RMTP middle layer). Additionally, once a retransmission has passed through a node, the table entry associated with the NACK of that packet is deleted. If a new NACK for the same packet arrives after this time, it is treated as a new NACK for any packet.

6.1.2 Control Packets

If the RMTP middle layer receives a RMTP_NACK packet, it adds the sending client or node to its list of clients or nodes waiting on this packet for this multicast. If there is no entry (either for this multicast because it's the first dropped packet of this multicast, or for this sequence number because it's the first client or node to realize that a retransmission is required) then an entry is created, a timestamp noted, and the request is forwarded toward the server via a call to DNP_send_packet. The packet will be intercepted at the next upstream node and the process is repeated wherever necessary. It is the clients' responsibility to retransmit a NACK request if an outstanding NACK has not been fulfilled after some time. If the RMTP middle layer receives an RMTP_NACK after the timestamp on the existing table entry is stale, then the

RMTP middle layer will reset the timestamp and again send the request up the tree (RMTP_NACKs which arrive before the timestamp is stale are simply pruned and their senders' addresses put in the NACK table).

Each client also maintains a sliding window of packet sequence numbers it is willing to accept, up to a maximum acceptable byte (MAB). This MAB is sent periodically by each client to its local node RMTP middle layer which periodically forwards the least of these (considering also the least of the MABs it knows about from its downstream nodes) toward the server. Again, the process is repeated at each intermediate node. If the minimum MAB from a node has not been updated for a certain amount of time, the RMTP middle layer will check to see if that node is still reachable according to DRP. If not, then that client is pruned from the multicast tree.

6.2 RMTP client

RMTP clients each live on a particular local node. Per client state is kept at each node for the clients at that node. Each client runs in a separate thread, and each node has one RMTP middle layer thread running to service all RMTP traffic at that node.

The joining and leaving of groups is handled by calls to `local_client_subscribe_to_multicast` and `local_client_unsubscribe_from_multicast`, provided by GMP and described in section **??**.

Once a client has joined a group, it is responsible for periodically announcing to the local RMTP middle layer its MAB, as well as detecting dropped packets and sending NACKs.

In the standard case, an RMTP client will wake up on a `pthread_cond_timedwait`. If a timeout has occurred, the MAB is sent and/or NACKs are sent if appropriate timeouts have expired. Otherwise, the thread has a packet reference in its buffer. Once a contiguous set of packets of a certain size has arrived in the client's buffer, the client writes those packets to disk, decrements the reference count on those packets, updates its MAB and finally informs the RMTP middle layer of the new MAB.

6.3 RMTP server

Like all other threads, the RMTP server thread has a buffer to receive packets containing information it should receive. In the ordinary case, the server will loop, sending a new RMTP_DATA_PACKET if that packet's sequence number is below the MAB the server is aware of. Every iteration the server will also check its receive

buffer, removing informational packets which might contain information about new MABs or retransmission requests. In response to retransmission requests, the server will send an RMTP_RETRANSMIT packet, which will be distributed by the RMTP middle layer to the appropriate clients.