Implementation of Remote Procedure Call and TCP Stream Interfaces to the Reliable Multicast Protocol

by Li Jiang, Chyr-Chong (Joseph) Ting, Donald K. Wong

Abstract

This paper discusses the TCP interface and the Remote Procedure Call interface to the Reliable Multicast Protocol (RMP). This paper begins by introducing RMP. After the discussion of various concepts of the Reliable Multicast Protocol architecture pertinent to our project, we will stress on the design, motivation, and implementation issues, including advantages and disadvantages to our implementation of the TCP and RPC layers. The conclusion will discuss what we learned from the project.

1.0 Reliable Multicast Protocol

The Reliable Multicast Protocol (RMP) is a protocol layer that runs on top of an unreliable multicast datagram service such as IP Multicasting. It provides a totally ordered, reliable, and atomic multicast guaranteed with options to change the quality of service (QoS) on a per packet basis. RMP also provides many communication options, including virtual synchrony, a publisher/subscriber model of message delivery, a client/server model of delivery, an implicit naming service, mutually exclusive handlers for messages, and mutually exclusive locks. [W1]

1.1 Introduction

The basic RMP protocol provides group communication between sets of processes connected by a multicast medium. RMP uses NACKs for error detection and retransmission. It provides a wide range of reliability and ordering guarantees on packet delivery, selectable on a per packet basis. On top of unreliable and reliable but unordered QoS levels, RMP can also provide atomic, reliable delivery of packets ordered with respect to each source. It provides both total and casual order delivery of packets. Totally ordered delivery provides virtual synchrony which guarantees that new members atomically joins or leaves a group such that the sets of messages delivered before and after each membership change are consistent across all sites. [W1]

The publisher/subscriber model of communication is based on textual group names. When network partitions are absent, any member of a group (a subscriber) will receive all packets sent (published) to the group associated with the group name. RMP also provides a client/server model of communication, where the servers are members of a group and clients are non-members, but are able to communicate with each other. [W1]

1.2 System Usage and Architecture

The basic RMP entities are RMP processes, groups, and membership views. The basic entity of communication is the RMP process. Essentially, the RMP process corresponds to a single UNIX process or application. There may be multiple RMP processes on the same host. Each RMP process is uniquely identified by the IP address of the host the process concatenated with a UDP port number. This identification is unique across all RMP processes in the internetwork and is constant over the life of the RMP process. [W1]

A RMP group is the basic unit of group communication consisting of RMP processes that are receiving packets sent to a given IP Multicast address and port. Each RMP group has a group name, which is a text string similar to Internet host names. Unlike host names, however, a group name identifies a group of members instead of a single host. The mapping of group names to IP Multicast tuples may either by handled externally by a multicast authority or implicitly by RMP. Given a text string, RMP applies a hash function that converts the string to a subset of class D IP address space (24 bits) and a port number (15 bits). Each group member also specifies a TTL when it joins. The TTL for packets sent to a group is set to the maximum of all the TTL's in specified by the group members. [W1]

Since membership of a group changes over time, a given list of members of a group is called a membership view, identified by a membership view ID. A membership view ID consists of two parts: an ID that is unique across the internetwork, and a counter that is unique across the max TTL for that membership view. The first part is the RMP process ID, while the second part is a counter maintained by the RMP process. After the RMP process is instantiated, the counter is set to the current time in milliseconds. After each subsequent creation of a membership ID, the counter is incremented. By limiting the lifetime of a membership view to less than 25 days, membership IDs are guaranteed to be unique as long as a RMP process does not generate more than one membership view per millisecond. By keeping membership IDs unique, we can filter RMP packets in case of collisions to the same IP Multicast addresses and port numbers. [W1]

1.3 Communication Model

RMP supports the peer group model and the client-server model of communication. In the publisher/subscriber model, implicit naming is used. With implicit naming, RMP allows processes to join a group by specifying the name of the group to join. Other processes send messages to this group by using the group name or a membership view ID associated with the name. Messages sent to the group name are delivered to all RMP processes that are members of that group. No explicit knowledge of the membership of a group is needed. RMP does this by mapping group names into {multicast address, port, TTL} tuples that are used to send to other members of the group.[W1]

RMP also allows RMP processes to name another RMP process (specified by an IP address and a UDP port) that is a member of the group and request that this member forward the packets to the other members. Because members are notified when the current membership of a group changes, processes can explicitly control group naming and membership when desired. [W1]

In the client/server model of communication RMP uses a multi-RPC mechanism. It allows non-members to send to a ring and to optionally get replies from a group member. When the packet is delivered, the current token site normally sends back an acknowledgment to the non-member. Multi-RPC packets are asynchronous and have integrated flow and congestion control. Mutli-RPC can receive replies through RMP handlers. RMP provides a set of mutually exclusive locks to each group, that members can efficiently request and release. The first six of these handlers are called handler locks. Each data or multi-RPC packet can specify a handler for the packet. The current member who holds the corresponding handler lock is expected to reply to this message. If a request is from a group member, the reply is also sent to the group. If it is from a non-member, it is sent only to that member. [W1]

1.4 Ordering Guarantees

[W1]

Qos	**Name and Service Guarantees**	**Delivery Time**
Unreliable	Packets are delivered 0, 1 or more times, in any order.	Immediately upon receipt of a data packet.
Unordered	Packets are delivered at least once, in any order.	Immediately upon receipt of a data packet, with missing packets detected and rerequested.
Source Ordered	Packets are delivered exactly once, in the order they were sent from each source.	After all of the data packets from the same source and with smaller sequence numbers have been delivered.
Totally Ordered	Source ordered, plus all totally ordered packets are delivered in the same order at all sites.	After all of the data packets with smaller timestamps have been delivered.
K Resilient	Totally ordered, plus delivery is atomic at all sites that do not fail or partition, provided that no more than K sites fail or partition at once.	After all of the data packets with smaller timestamps have been delivered and the token has been transferred K-1 times.
Majority resilient	K resilient, with K set to (MaxN+1)/2, where MaxN is the highest number of sites in the group for any membership view in the OrderingQ.	Same as K-Resilient, but also requires that only a majority partition can continue functioning.
Totally resilient	K resilient, with K set to N.	Same as Majority Resilient

Each QoS level builds upon the previous level, providing a bit more reliability. The basic RMP QoS levels are unreliable, reliable, source ordered, and totally ordered. In general, higher levels of QoS implies an increase in the latency of packet delivery. One issue that the basic RMP QoS does not address is the event when there are failures and partitions within a group. For example, in the totally ordered delivery, if a group member is malfunctioning, the group connection will terminate. In many situations, however, it is desirable to terminate a connection. On the other hand, it is desirable to have atomic delivery. Therefore, RMP provides three extra services: K-Resilient, majority resilient, and totally resilient. {W1]

K-Resilient is a guarantee about delivery and ordering of packets between partitions. K is the minimum number of sites that must fail in order to violate atomicity guarantees. Each RMP member M checks that at least K other members have received a packet before M can deliver it. In this case, each partition will always have at least one member which has received all of the packets that have been delivered at any site. [W1]

The next level of atomicity, majority resilience, guarantees that no matter how many partitions or failures occur, any members of a group that deliver any two messages will agree on the same ordering of the messages. This level of atomicity is achieved by not allowing minimum partitions to continue and by making sure the majority of the members of a group have a message before any members delivers it. A conservative estimate is made on the total number of members in a group, called MaxN. This is the maximum number of members that are in any membership view for any packet which is not received by all the members. [W1]

Total resilient delivery requires that a packet be received by all the members before it can be delivered. This is done by making sure that each member knows that all other members have received it.

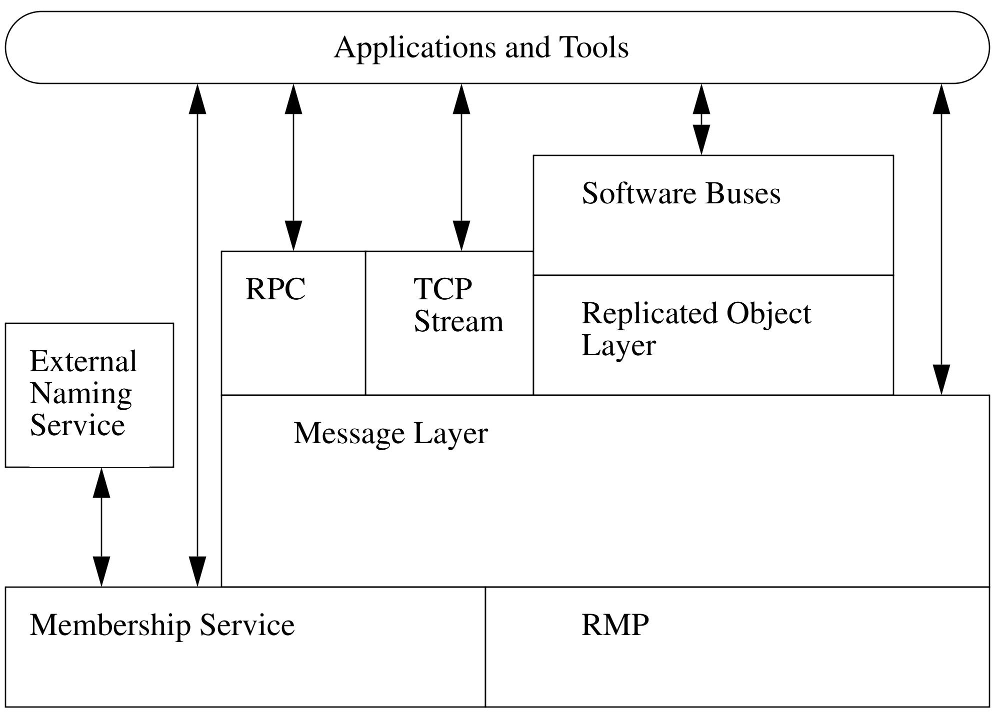

FIGURE 1. Layered RMP architecture [W3]

2.0 RMP Layer Architecture

The RMP architecture is a division of a wide range of functionality and interfaces into a series of layers. Intuitively, higher layers are built upon lower layers. However, applications may bypass the upper layers and directly access the lower layers. While the highest layers are interfaces to applications, the lower layers provides a reliable multicast service on top of unreliable datagram services such as IP and IP Multicasting.

2.1 Software Buses

The Software Bus Layer provides queueing of messages for inactive applications, automatic starting of inactive services, language independent interfaces, and multiplexing one to multiple processes. Examples of software buses include the MessageBus, ToolTalk, and the MutiBus.

2.2 TCP Streams Layer

The TCP Streams Layer is one of the two layers we implemented. The streams layer provides a TCP-like virtual circuit interface. Essentially it will allow multiple streams to be opened up, with each corresponding to a {token ring::domain number} pair. Design and implementation issues will be discussed in the next section.

2.3 RPC (Remote Procedure Call)

The RPC layer we implemented supports the client/server model enabling the non-member RMP processes to make remote procedure calls to a multicast group using a Mutli-RPC algorithm. It is written on top of the Messaging Layer. The design issues will also be discussed later.

2.4 Replicated Object Layer

It provides object type registration, replication of objects at multiple sites, distributed atomic transactions, and update of objects to new and failed sites.

2.5 Messaging layer

The Messaging Layer is the most common interface to application and tools that use RMP. It provides destination layer typing for all the higher layers. This is a field that is included with a message that indicates what layer it was sent from. In addition, this layer provides segmentation and reassembly of messages, domain and client filtering, and compacting of small messages into larger packets. The Messaging Layer assists the RPC layer by keeping track of which messages are being replied to, and will match up these replies with the messages when the replies are received.

2.6 Membership Service layer and RMP Layer

The membership Service Layer records membership of rings, removes faulty sites, and provides mutually exclusive locks as well as optional implicit naming services. The RMP layer is packet based. It controls flow and congestion, and provides the QoS service mentioned earlier.

3.0 RMP Application Programming Interface

3.1 TCP Multicast Programming Interface

3.1.1 Introduction

The streams interface is a layer built on top of the existing messaging layer of the Reliable Multicast Protocol. It allows for the sending and receiving of messages of unlimited length to and from multiple hosts and provides a familiar interface to a high performance, totally ordered multicast protocol. The interface is similar to traditional stream socket interfaces and the functionality mirrors those of TCP sockets.

3.1.2 Motivation

The fundamental idea behind mirroring TCP sockets is to provide users a with a familiar interface to RMP and to utilize the functions of RMP. Since most network applications use either UDP or TCP sockets, adding multicast support will be a major undertaking. Even if IP Multicasting is used, it does not provide the added functionality that RMP provides, including reliability of messages, total and source ordering of messages, and fault tolerance handling. The implementation of the TCP Streams Layer will facilitate the production of multicast applications because the RMP socket interface closely resembles the TCP socket interface.

3.1.3 Implementation

As mentioned earlier, our goal is to implement a socket-like stream interface with as much similarity to the TCP socket interface as possible. Below is a table which outlines the similarity between the two interfaces. Since the RMP naming mechanism uses group names to refer to multicast groups instead of a hostname/port number pair, the bind() system call is not needed.

TABLE 1. TCP socket interface versus RMP socket interface.

TCP Socket Interface	RMP Socket Interface
socket()	RMPSocket()
connect()	RMPConnect()
read()	RMPRead()
write()	RMPWrite()
select()	RMPSelect()
setsockopt()	RMPSetSockOpt()
getsockopt()	RMPGetSockOpt()
close()	RMPClose()

- ***int RMPSocket()***

This call, if successful, will create and return a socket descriptor that the application can use with subsequent RMP socket calls. On failure, it will return -1. The socket descriptors returned are guaranteed not to clash with regular TCP socket descriptors or file descriptors so that the RMPSelect() call will work with all types of descriptors. There are no parameters to this call since the protocol is assumed to be RMP.

- ***int RMPConnect(int sockId, char *groupName)***

RMPConnect() will connect the specified socket to the specified multicast group. sockId must be a socket descriptor returned from a previous call to RMPSocket() and the socket must not already be connected to a group. If the multicast group does not already exist, this group will be created. Otherwise, this socket is joined to the existing group. This call returns 0 on success and -1 on failure.

This call differs from the connect() system call in that the name of the group is specified rather than a hostname/port pair. This is because RMP uses an implicit naming model, ie. group names are used to identify multicast groups. Thus, the struct sockaddr type used in the connect() system call is not used here. The name of the group is used instead.

- ***int RMPRead(int sockId, char *buf, int nbytes)***

This call will try to read nbytes of data from the multicast group connected via the socket specified by sockId into the buffer buf. The socket can be set to blocking or non-blocking mode. If it is set to blocking mode, then this call will not return until something is read from the socket, at which time it will return the number of bytes actually read. If an error occurs during this time, it will return -1. If the socket is set to non-blocking mode, then this call will return immediately with -1 if there is nothing to be read or read the data and return the number of bytes read if there is something available to be read. This is equivalent to the read() system call.

- ***int RMPWrite(int sockId, char *buf, int nbytes)***

This call will try to write nbytes of data from the buffer buf to the multicast group connected to the specified socket. If the socket is set to blocking mode, then the call will not return until the flow control of this socket enables the data to be written, at which time the number of bytes written will be returned. If an error occurs, then -1 is returned. If the socket is set to non-blocking mode, then the call returns immediately with -1 if the flow control does not allow data to be sent to the group at this time or it sends the data to the group and then returns the number of bytes sent if the flow control is fine. This call is equivalent to the *write()* system call.

- ***int RMPSelect(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)***

This call operates similarly to the *select()* system call. Both TCP and RMP socket descriptors as well as regular file descriptors can be passed to *RMPSelect()*. TCP socket descriptors and file descriptors are passed to the *select()* system call to process while the RMP socket descriptors are handled by *RMPSelect()*. Read socket descriptors are checked by determining if that socket has messages waiting to be read. Write socket descriptors are checked by examining the flow control status of that particular group. The number of file descriptors in the descriptor sets is returned on success and -1 is returned on failure.

- ***int RMPSetSockOpt(int sockId, int option, void *value, int len)***
- ***int RMPGetSockOpt(int sockId, int option, void *value, int *len)***

These functions operate exactly like the *setsockopt()* and *getsockopt()* system calls, except that different options are available. The options available to RMP sockets are the quality of service desired, the time to live for the message, and the blocking/non-blocking mode of the socket. 0 is returned on success and -1 is returned on failure. The table below lists the options available and the legal values for each.

TABLE 2.

Option Name	**Values**
Quality of Service	QOS_UNRELIABLE QOS_UNORDERED QOS_SOURCE_ORDERED QOS_CAUSAL_ORDERED QOS_TOTALLY_ORDERED QOS_K_RESILIENT QOS_MAJORITY_RESILIENT QOS_TOTALLY_RESILIENT
Time to Live	an integer > 0
Blocking Mode	NON_BLOCKING BLOCKING

- ***int RMPClose(int sockId)***

This call will deallocate the socket created with the *RMPSocket()* call. If the socket is connected to a multicast group, then it is disconnected from the group before deallocation. This returns 0 on success and -1 on failure.

3.1.4 Differences Between Traditional Stream Interface

Although the RMP socket interface closely resembles the TCP socket interface, there are several inevitable differences.

The primary difference is the naming convention used. RMP uses an implicit naming model where groups are specified by name rather than a multicast address. This changes the semantics in the startup of connections. Instead of specifying a hostname and a port number or a multicast address, a group name is specified. Similar to domain names, it is more desirable to refer to names of groups as opposed to cryptic IP addresses. This provides the user with a level of abstraction that hides address mapping.

Another difference is the functions that set and retrieve socket options. Many of the options available to regular sockets are not applicable to RMP sockets. For example, the options to reuse addresses and to permit sending broadcast messages are irrelevant in the context of RMP sockets. On the other hand, there are numerous options that are useful but have not been implemented. Such options include send buffer size and receive buffer size. However, there are extra options available to RMP sockets that are not available to regular sockets: namely, quality of service and message time to live (TTL).

3.1.5 Not Yet Implemented

Support for non-RMP socket descriptors as well as RMP socket descriptors should be included in all of the above procedure calls. The procedure calls would then operate differently based on the type of descriptor. Since both RMP socket descriptors and non-RMP descriptors are allocated from the same name space, these are guaranteed not to conflict. For example, if a non-RMP socket descriptor is passed to *RMPRead()*, the *read()* system call would be called on the descriptor. Otherwise, it would operate as usual. This functionality already exists in the *RMPSelect()* call, but needs to be added to the others.

Joining to a domain of a multicast group should be supported as well. This allows the application to become a member of part of the group, rather than the entire group.

3.2 Multicast RPC Programming Interface

The Remote Procedure Call (RPC) programming interface provides application with a blocking RPC capability. Instead of providing a unicast RPC service, the actual RMP process that replies to a request can easily and frequently change, while still guaranteeing that at most one site will respond. The process of selecting one RMP site to execute the RPC call is totally transparent to the user. Essentially, RMP RPC follows the conventions of the regular RPC model (see diagram below). The client procedure sends a request to a group. The group elects a least busy site that handles the request. The client and the server communicate across the network through stubs. These stubs are responsible for translating data into a universal data representation using Sun's External Data Representation (XDR).

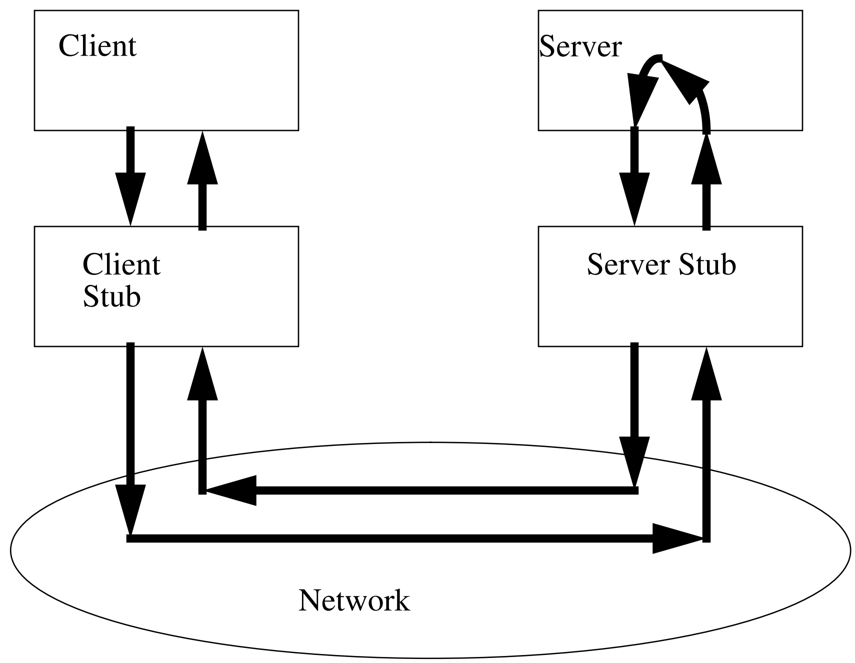

FIGURE 2. Local and remote procedure call communication [B1]

3.2.1 Motivation

The client/server model of computing is a popular model for distributed processing. It applies to any processing environment where one set of entities requests work to be done from another set that actually performs the work.

The primary goal behind multiple servers handling RPC is load sharing. By allowing a group of servers to handle RPC requests that can be handled by any of them, it is possible to delegate the task to a machine that is least busy or even idle or even distribute work among a group of servers. The result is an increase in performance and utilization. In addition, more services will be available to the client. An example of applications that can take advantage of this capability is data replication in a distributed database system where copies of a database are maintained at all the major sites. With Quality of Service(Qos) of total ordering, data at each site is consistently maintained with each other. For example, an update to data at one site will propagate to all the other sites before any more updates can be performed. As a result, any site can serve a request by the user without worrying about consistency. Users have the complete freedom of specifying the QOS level for the service which in return decides whether the RPC request should be blocking or non-blocking.

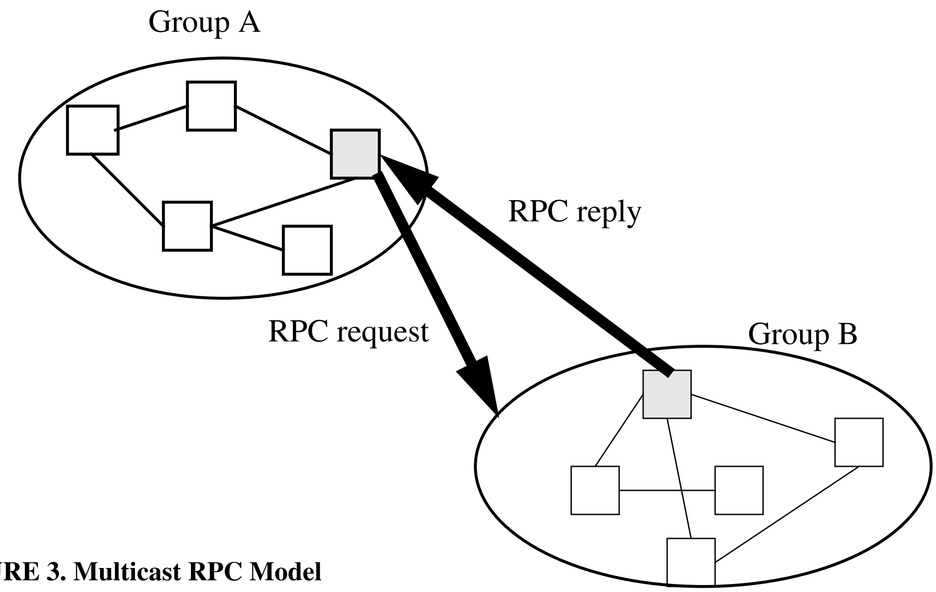

FIGURE 3. Multicast RPC Model

3.2.2 Implementation

To serve an RPC request, each RMP site provides an RPC server stub that listens to message events coming from the RMP process as a result of a RPC requests from a non-member of the group. When a particular site is chosen to be the host of the event, it will then receive the RPC request messages with the procedure name, the version number, and the arguments represented in Sun's XDR type format. The remote site will perform a translation from XDR format to the normal C data type. The portmapper will subsequently dispatch the arguments to the targeted local procedure or program. Service dispatching and network I/O details are hidden from both sides of an application. Portmapper advertises the program address information to open a channel with the server.

The data structure of our portmap is implemented as follows:

```
struct _PMap{
        char *(*progname)();
        u_long prognum;
        u_long versnum;
        u_long procnum;
        void (*proc(void *));
        int port;
        char *(*XDRInProc(char *));
        char *(*XDROutProc(char *));
        void (*dispatch)();
        struct PMap *next;
};
```

It contains the program name, program number, procedure number, procedure name, as well as the version number of the program that registered with the RMP RPC server. Many procedures may be associated with one program, and none of the procedures are explicitly advertised in the portmap. The portmap also records the XDR procedures that decodes arguments from the client and encodes replies to send back to the client. A dispatch procedure is also saved so that a user has control over how the procedure can be dispatch. A default dispatch routine is also available for normal procedures. The member port in the port map stores the port number of the program.

• *Server*

To register a procedure. the *RMPregister()* routine is called. It tells the server's RMP portmappper that a procedure is ready to provide a service and should be registered. It also alerts the RPC server to make ready a dispatch routine to watch for incoming requests.

```
RMPRPC_Register(u_long prognum, u_long versnum,
                u_long procname,
                void (*inproc(XDR *, char *)),
                void (*outproc(XDR *, char *)));
```

Once the server starts, it registers itself via a call to RMPRPC_Register(). It then goes into an RMP event loop, listening for message events to the member, with a server routine RMPRPC_Run(). It basically puts the server in an infinite loop and effectively issues a blocking select(). When the least busy server is chosen, a connection is established between the client and the server. As soon as the RMP event occurs, a service request in the following format is generated. It then launches an associated service procedure using the arguments received.

```
struct svc_req {
        u_long rq_prog;
        u_long rq_vers;
        u_long rq_proc;
        struct opaque_auth rq_cred;
        caddr_t rq_clntcred;
        SVCXPRT *rq_xprt;
};
```

• *Client*

On the client side the, one RPC call is all that is needed. The RMPRPC_Call() function is called by the client to find and execute an RMP service on a host, given that the service is registered and responsive.

```
int RMPRPCcallprc(char *groupName, u_long prognum,
                  u_long versnum,  u_long procname,
                  char *in, int (*inproc)(XDR *, char *),
                  char *out, int (*outproc)(XDR *, char *),
                  struct timeval *timeout);
```

It first establishes a connection with a chosen member of the group groupName which is chosen through a special mechanism implementedby the RMP layer. It then checks with the host's portmap and sets up the appropriate I/O channel to run a remotely-registered procedure. The *in* and *out* are pointers to the remote procedure argument and *inproc*() and *outproc*() are XDR routines that encode the data to provide the server with arguments and decode the data from the server reply.

4.0 Performance

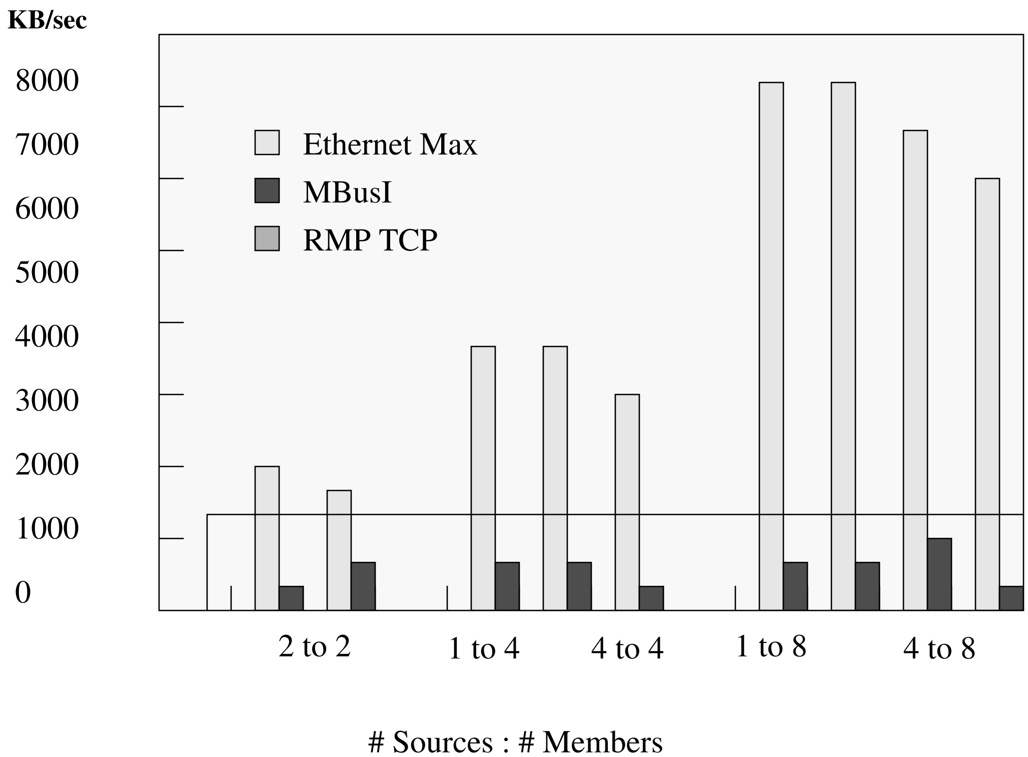

FIGURE 4. Aggregate Throughput(KB/sec) [W1]

Figure4 shows the aggregate throughput of the network as a function of the number of sources and number of group members. This graph represents previous simulation of RMP. Aggregate throughput is the throughput the user sees. It is computed by taking the amount of data sent from all of the senders and multiplying it by the number of destinations. In these tests, the sender is also a destination. This is done so that it can see its own messages totally ordered with the others.

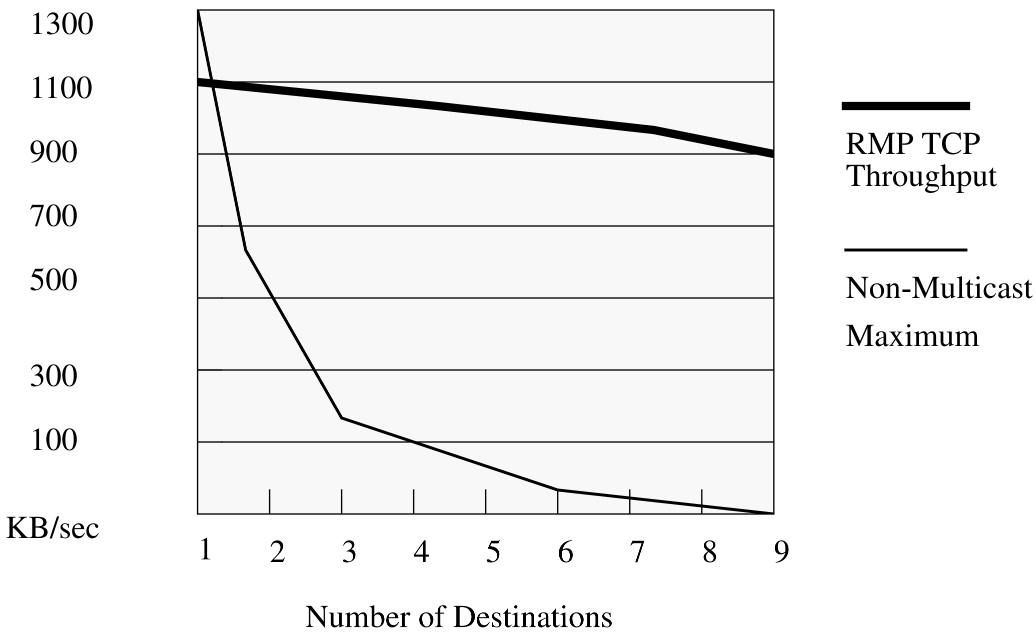

FIGURE 5. Single Sender Throughput [W1]

Figure 5 shows the single sender throughput plotted against the number of destinations. The single sender throughput is equal to the aggregate throughput divided by the number of destinations. RMP breaks the fundamental unicast limit for two destinations, and so has higher throughput in this environment for all groups of more than one destination. As you can see the performance of the RMP stays almost constant.

Unfortunately, we did not have the resources to preform our own tests using the TCP and RPC API, but we expect that the results would be similar. Some performance might be lost due to extra queueing and overhead of maintaining socket numbers and RPC servers. However, this extra overhead should be minimal and should not have any significant effect on the overall performance.

5.0 Conclusion

This paper discusses our implementation of the TCP socket layer and the RPC layer to RMP. Most of our work involved reading through the RMP material, including the various versions of source code and API's. Although there were documentation, most of the code was undocumented, and there were numerous bugs. For example, the message layer did not even compile. However, designing and actually implementing the layers enabled us to realize the gap between theory and actual implementation. As a whole, we believe that the TCP interface and the RPC interface are useful additions to RMP because they are transparent and easy to use. If the application programmer is familiar with TCP and RPC, he/she will be able to utilize the functionalities of RMP.

6.0 References

[A1] S.Armstrong, A. Freier, K. Marzullo. “Multicast Transport Protocol”, February, 1992.

[B1] Bloomer, J., Power Programming with RPC, O’Reilly & Associates, Inc., 1992.

[T1] Tanenbaum, A. S., Computer Networks, PTR Prentice Hall, Englewood Cliffs, New jersey, 1989.

[W1] B.Whetten, T. Montgomery, S. Kaplan, “A High Performance Totally Ordered Multicast Protocol”, U.C. Berkeley, 1992.

[W2] B. Whetten, “The Message Bus II and the Reliable Multicast Protocol”, Champaign-Urbana, Illinois, February, 1995.

[W3] B. Whetten, T. Montgomery, “The Reliable Multicast Protocol API”, November 9, 1994.

[W4] B. Whetten, T. Montgomery, “The Reliable Multicast Protocol Messaging Layer API”, November 9, 1994.