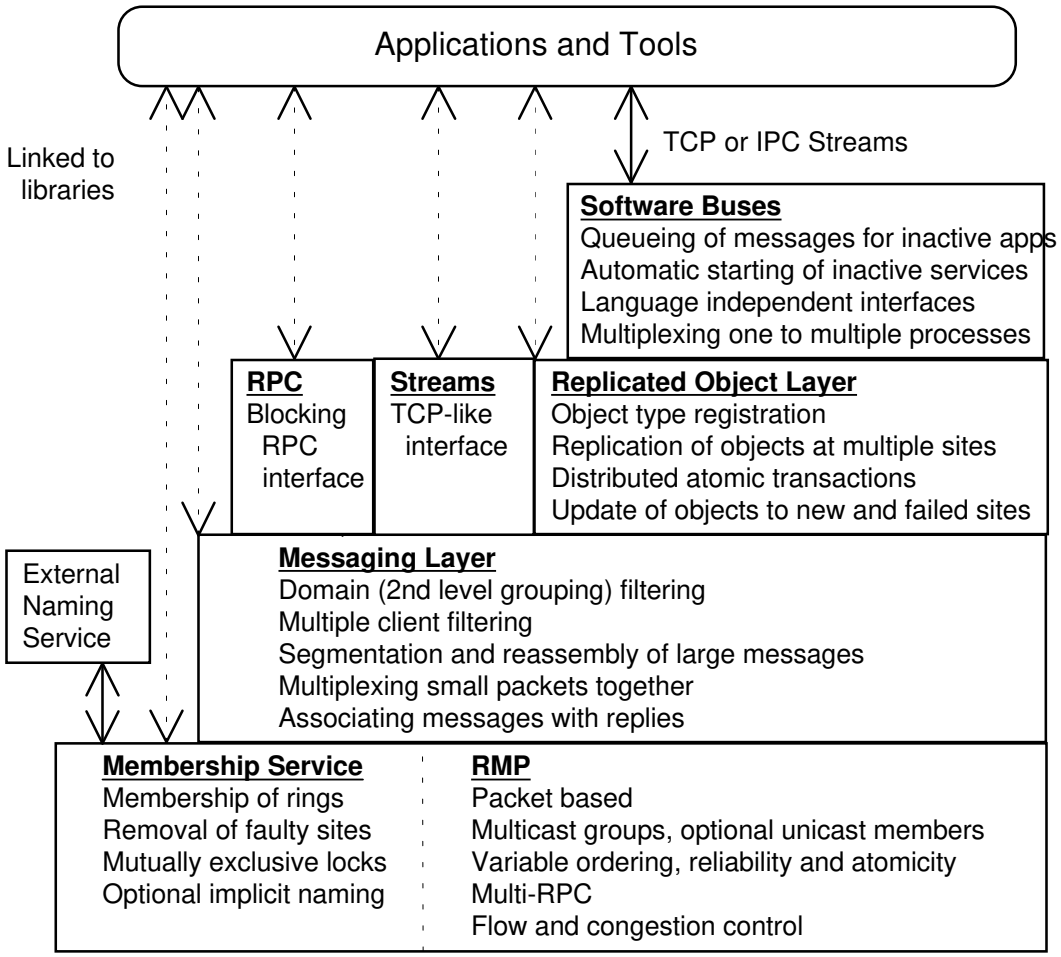
RMP Architecture



The extended RMP architecture is a plan for the division of a wide range of functionality and interfaces into a series of layers. Higher layers build on top of lower layers, although in some cases one or more lower layers will not be needed. In this case, those lower layers would not have to be linked in to the application or tool in question. Almost all communication will be done through procedure calls to linked in libraries, except at the software bus level. At this level, communication will usually occur through TCP/IP or IPC streams. An external group naming service, if one is used, would also communicate with an inter-process socket of some sort.

RMP and Membership Service Layer

The basic transport services are provided by the RMP library. This library provides both membership services as well as a wide range of packet delivery options. These features have been well detailed in other RMP documents.

Messaging Layer

The messaging layer provides a set of functions that we expect will be needed by many, if not most, of the applications and tools that use RMP. This layer is organized around messages, instead of packets. The message layer is required by all higher layers, because it provides destination layer typing. This is a field that is included with a message that indicates what layer it was sent from. This is critical for error checking because higher layers will add in extra header fields at the beginning of the data packets. At the destinations, this field is used to determine which layer to deliver the packet up to.

In addition to destination layer typing, this layer provides segmentation and reassembly of messages, domain and client filtering, keeping track of messages that are being replied to, and compacting of small messages into larger packets. Each message can be of arbitrary length, subject primarily to the available memory of the group members. Messages are delivered as an indivisible unit to the destinations, even though they may be broken up into multiple packets by RMP. Along with this segmentation and reassembly of large messages, this layer also provides two types of filtering. It provides domain filtering, where packets delivered to the destinations of a token ring are the delivered only to the clients that have requested them. It also provides multiple client filtering, which allows multiple clients to use the same copy of RMP, although no support for these clients being in separate processes is made at this layer. This task is left to higher layers, such as software buses. For people familiar with the first version of RMP, it did these three functions directly in the transport protocol. By including them in this layer, this layer provides a superset of the functionality supported in version one.

The messaging layer will also keep track of which messages are being replied to, and will match up these replies with the messages when the replies are received. This is used for messages that request a handler. This makes it very easy to implement a blocking RPC layer, for example. Finally, this layer will compact multiple small messages into larger packets. This is done by delaying the sending of messages to a token ring until RMP reports that there are no packets waiting to be sent to this token ring or until a set number of bytes have been collected. When either of these conditions occur, the waiting messages are all wrapped up into a single larger message and sent. Upon arrival, the larger message is decomposed in to the smaller messages. This policy minimizes the latency of small messages by sending them through as soon as possible while maximizing throughput when the sending queue backs up.

While there are a good number of extra fields needed for messages sent with this set of functions (up to 24 bytes per message in the worst case) they are only included in the message headers when needed.

RPC Layer

This layer provides a blocking RPC API. Multiple different versions of this layer can be implemented. The most basic version will primarily be a wrapper to the messaging layer, while more advanced implementations could do resource location and binding, argument marshalling, and stub generation. RMP provides a multi-RPC service instead of a unicast RPC service, and so the actual process that replies to a request can easily and

frequently change, while still guaranteeing that at most one site will respond. The RPC layer will use all of the functions of the messaging layer except the multiple client filtering. Please note that non-blocking RPC functionality could also be included in this layer.

Streams Layer

The streams layer provides a virtual circuit interface. It will allow multiple streams to be opened up, with each typically corresponding to a token ring:domain number pair. This layer will have very little functionality, being primarily an alternate API to the messaging layer. The name resolution for a stream and the unicast vs. multicast semantics of the streams still have to be decided upon. In addition to allowing multiple streams, this layer would also allow for out of band data.

Replicated Object Layer

This layer provides replicated objects. Each object has an integer type field associated with it, which primarily defines its size. Internal typing of objects is left to higher layers, and is facilitated by allowing applications to create a typing object which is replicated among the members and contains the typing information. Objects are grouped into object pools, and each pool is replicated across a set of processes. To allow an object pool to be replicated at multiple sites, this layer can copy objects to new sites, and can send updated copies to sites which have failed or partitioned away and then rejoined. Updates to objects are done with totally ordered messages, so consistency is preserved. For more complicated updates, distributed transactions are provided. Because they use totally ordered multicast, these transactions will be extremely efficient. Applications that use this layer can register callback functions which notify the application when a given object changes. Applications can also receive notifications of when all sites in the object pool have received an update, and when membership in an object pool changes.

At this layer, objects are very rudimentary. Because of this, they can correspond to any replicated piece of information that will be mutated over time. For example, each page in a white board could correspond to an object in an object pool. When a new site joined the object pool, it would automatically get a current copy of the pages. Any changes that are made to a page at one site would then get consistently updated at all other sites, and the callback functions could be used to notify the application when a given page needs to be redrawn. Another possible application that could be built on top of this layer is a replicated OODB.

Software Buses

Software buses provide a tool that allows multiple processes to communicate through the same copy of RMP. In addition to this multiplexing of multiple processes, software buses can also provide language independent message interfaces, dynamic invocation of processes that have been registered but are not currently active, and queuing of messages to processes that aren't currently active. Communication between the buses and the client processes is usually done with TCP/IP or IPC sockets. Examples of software buses include the MessageBus, ToolTalk, and the MultiBus.

Other Possible Layers

Many other layers are possible--these are just a few of the ones that we feel are most important. Most notably, we envision a possible security layer which sits between the messaging layer and other layers on top of it. Object systems such as CORBA would fit on top of the replicated object layer. A distributed shared memory layer could also be provided on top of the replicated object layer, along with replicated data bases and replicated OODBs.