Robust Communications for High Bandwidth Real-Time Systems

Jorgen David Pedersen

CMU-RI-TR-98-13

Submitted in partial fulfillment of
the requirements for the degree of
Master of Science in Robotics

The Robotics Institute
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, Pennsylvania 15213

May 1998

This work was sponsored in part by NASA. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of NASA or the U.S. government.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Foremost I would like to thank Tony Stentz for giving me the freedom to independently explore and create, while simultaneously providing the perfect level of guidance. Likewise, I want to thank John Bares for pushing me to go that extra mile throughout.

Thanks to all my friends, especially those at the National Robotics Engineering Consortium. I believe that all those football games, dart games, racquetball games, and the frisbee- and football-throwing sessions both outside and *inside* the building helped this workaholic stay *slightly* sane!

A special thanks goes to my friend Chris Leger who helped put the insanity *back* into me during the week-ends by dragging me (sometimes literally) to the top of some mountain. Rock climbing helped me see what is really important in life.

Finally, thanks to my family for supporting me in all my endeavors.

Abstract

Robotic systems today have such high computational requirements that it is necessary to distribute the workload across many processes and processors. Because of this distribution, a means for transferring data between these processes is required. Many low level protocols exist today for handling this communication task, each with its own advantages and disadvantages. This thesis work strives to create a higher level communication protocol built on top of these lower level protocols, geared specifically toward meeting the system requirements of real-time robotic systems.

This thesis work attempts to encapsulate all the beneficial features from each of the underlying protocols, providing a programmer with an efficient toolkit for interconnecting a system of processes. This new high level protocol provides several advantages over other protocols. These advantages include extremely high bandwidths in real-time environments, low memory overhead, stability, data transmission reliability, diagnostic capabilities, and ease of use. Other communication protocols exists, but none currently provide all of these features.

This thesis work presents new philosophies on how data should be transferred between processes. Moreover, this work not only presents a new communication paradigm, but it also provides a real-world solution to real-time interprocess communications. This protocol, termed Real-Time Communications (RTC), has been rigorously tested in several real-world robotic applications.

Chapter 1

Introduction

1.1 Background

Algorithms employed by robotic systems today are becoming more and more complex. Consequently, these systems have higher computational requirements. Methods are being applied to these algorithms in an attempt to reduce the computational costs, but there remain some systems that cannot be optimized to run as a single real-time process. For these systems, processing needs to be done in parallel, distributing the workload over many processes and processors. This parallel, or concurrent, processing can be achieved by either multithreading or multitasking.

Multithreading, as its name suggests, distributes the workload into several "threads" of execution. New threads are generated from previous threads, stemming from the original thread of execution. All threads of execution share the same memory space, meaning that no explicit communication is required between threads. Therefore, threads must use semaphore techniques in order to access all shared memory. However, the workload distribution is limited. A new thread cannot be spawned on a different processor running in some remote location. This is a major disadvantage, especially if the system is to be robust, as many of today's system try to be.

The other technique of concurrent processing is called multitasking, or multiprocessing. Just as multithreading has multiple threads, multiprocessing has multiple processes. These processes run in parallel just as threads do, except they do not share the same memory space. This has an advantage in that implementing multiple processes is far simpler, but it also introduces the problem that now these processes have no direct means of communication. This problem generated the need for interprocess communication (IPC). IPC is the mechanism which allows these processes running in separate memory spaces to talk to one another. Unlike threads, processes may be distributed over the network to remote processors, providing the tools needed for creating a robust communication software package.

The development of IPC had to trade-off speed and reliability. An ideal protocol of communication would be one that could optimize both of these aspects. However, no such protocol exists, therefore many protocols emerged, each with its own relative balance of speed versus reliability. Some protocols were built on top of other protocols to enhance reliability. For example, the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP) were both built on top of the Internet Protocol (IP). In fact, TCP is also referred to as TCP/IP. TCP traded off speed for reliability, whereas UDP traded off reliability for speed. There are many other protocols, but TCP and UDP, also known as the Internet protocols, are two of

the most common and popular protocols. TCP has grown in popularity over UDP due to its reliability.

1.2 Motivation and Goals of this Thesis

This thesis presents a robust interprocess communication design, *real-time communications*, RTC [rtc], which provides real-time capabilities as well as flexibility and reliability. RTC merges existing low level communication protocols into one protocol, providing the tools for developing and modifying real-time multiprocess systems. RTC was designed with robotic systems in mind, especially embedded robotic systems.

Robotic control and planning systems today often require concurrent processing to meet their computational needs. In general, the system architecture of robotic systems is constantly undergoing change since robotics is such a cutting edge research field. Therefore, for robotic systems, multitasking has a much greater appeal than multithreading. As mentioned earlier, multitasking requires IPC, but most of the IPC protocols such as TCP and UDP are still fairly rigid. That is, specific knowledge about each process, such as its address, must be predefined. This means that anytime the system architecture changes, all the code most likely would need to be recompiled. Moreover, effectively coding with TCP or UDP requires a large amount of background knowledge about socket communications. Therefore, one of the primary goals of the RTC protocol is to form another layer on top of existing protocols. This next layer of protocol will allow a programmer the ability to implement multitasking without having to be an expert in networking and IPC. RTC will also give the flexibility needed to meet the changing needs of robotic systems.

Most robots today also have the ability to perceive the world in some manner. Mobile robots need to perceive their environment so that they are able to safely maneuver through their surroundings, avoiding any obstacles. There are many sensor technologies used to map a robot's surroundings. Of these, one particular sensor technology, called "ladar", was a major driving force for the creation of the RTC protocol.

Ladar sensors transmit and receive a burst of laser light. This light travels as a narrow beam away from the sensor until it collides with a part of the environment. When the light contacts part of the environment, the light reflects, and part of the light reflection is received by the sensor. The ladar sensor is able to measure the time interval between the instant the laser beam was transmitted and received.

Since the speed of light is constant, this time interval provides a distance measurement from the sensor to the part of the environment where the contact occurred.

For mobile robotics, these ladar sensors are usually mounted onto scanning and panning mechanisms. These mechanisms redirect the laser as needed, providing absolute distance from the laser sensor to many points in the region of interest of the environment. Since light travels so quickly, it is possible to receive thousands of data points from the sensor in just one second, with the distance to each of these data points being known.

A large amount of data is being generated from these ladar sensors. The data being provided is usually in a raw form of just range and angle information Several calculations are required to transform the raw sensor data into a form acceptable to processes that use this data. Generally, one processor cannot perform all the required calculations itself due to the high bandwidth of data being output by these sensors. A pipeline of processors, and consequently processes, is required. The goal of this pipeline is to have each process perform the maximum amount calculations possible, while keeping up with the data rate. That is, many CPU cycles of each processor will be used just to receive and retransmit the large amounts of sensor data, so the amount of calculations that a process can perform are limited by the amount of remaining CPU cycles for that processor.

To implement a sensor pipeline as discussed above, a fast protocol of communication is required, possibly on the order of magnitude of one megabyte of data per second. The problem is that IPC protocols have too much overhead to keep up with the high bandwidth of data flowing through one of these sensor pipelines. Therefore, the other primary goal of the RTC protocol is to use the speed of shared memory to deliver this high bandwidth sensor data from one process to another.

Essentially, RTC is combining the desirable aspects of multithreading and multitasking into one protocol. We will be able to employ shared memory for high bandwidth systems, such as a sensor pipeline, while retaining the freedom of process distribution across a network.

A secondary goal of this thesis work is to provide a protocol that enhances the development and debugging of concurrent systems. The low level networking functionality will be hidden from the user. Debugging functionality will be introduced in order to tap into the low level functionality as needed. This debugging functionality should also speed up the development of multiprocess systems. Other goals of the RTC protocol include reliability, portability, and robustness.

1.3 Related Work

Several software packages have been created which are similar to this research, some academic, some commercial. Each of these packages has its strong points, but all seem to have their deficiencies as well. Several of these packages emerged from Carnegie Mellon University (CMU). The first of these packages was TCA [tca], Task Control Architecture [simmons94], which not only handled interprocess communications, but also served as a toolkit for building system architectures. TCA uses TCP for communicating between processes.

The initial versions of TCA followed what is known as the “client-server” paradigm of networking (see Figure 1.1). With this paradigm, every process in the system would connect to the server. When a process would like to send data to another process, it would send the data to the server, and the server would in turn send the data to the appropriate process. This method, although conceptually much easier to implement, is inefficient because the data must make two hops, as opposed to only one hop if the data were sent directly to the receiving process. Later versions of TCA allowed for what is known as “point-to-point” connections, which is another networking paradigm (see Figure 1.2). This paradigm requires only one hop when sending data from one process to another process.

FIGURE 1.1. Client-Server Paradigm

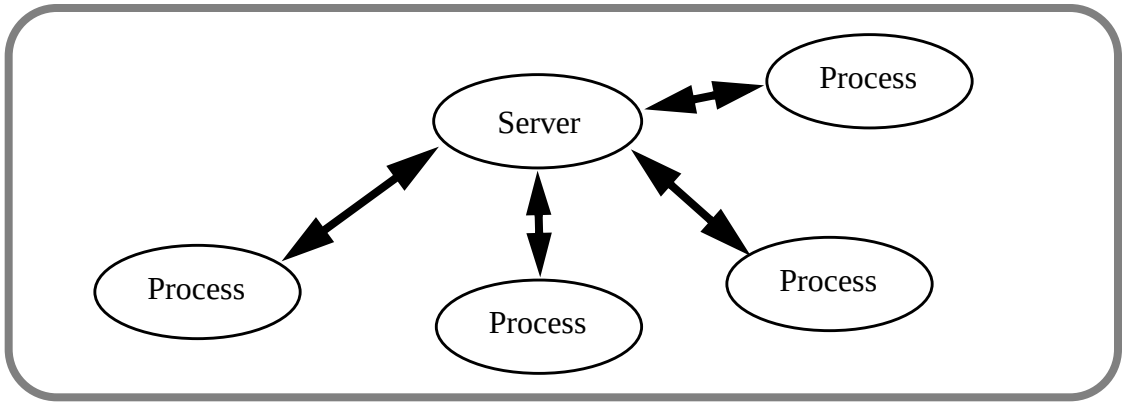
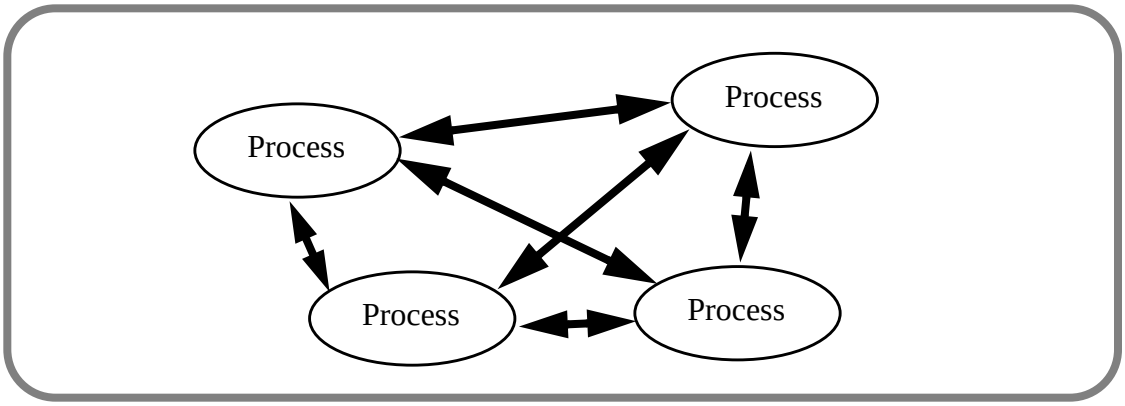


FIGURE 1.2. Point-to-Point Paradigm



TCX [fong95] originated at CMU as well. TCX stripped away all the control architecture code and was merely the communications part of TCA. The third communications package emerging from CMU was InterProcess communications Toolkit, IPT [gowdy96]. IPT was based on TCX, which was based on TCA. IPT brought with it the approach of object-oriented programming, among other aspects.

TCA, TCX, and IPT are all very similar. All of these packages now support the quicker point-to-point paradigm, use a “message handler” technique for handling control flow, and use the TCP internet protocol for transferring data between processes. This thesis uses some of the ideas founded in these packages.

The Common Object Request Broker Architecture, CORBA [corba], like IPT, is an object-oriented means of distributing processes over the Internet. Parallel Virtual Machine, PVM [pvm], is another way of distributing work across many processes. PVM offers “load-balancing”, where the amount of work done by each process is handled by weighting each process with a priority based on the physical hardware on which each of the processes is running. Remote Procedure Call, RPC [birrell78], was introduced for client-server applications.

Another related software package called Network Data Delivery System, NDDS [ndds], is commercially available. This Application Program Interface (API) provides a fast, yet unreliable message delivery method, as it was built on top of UDP. With NDDS, the burden is put upon the user to break up large messages into smaller messages due to the message size limitations of UDP.

Neutral Management System, NML [nml], is very similar to RTC. Just as RTC was designed specifically for robotic applications, NML was developed specifically for manufacturing applications. NML provides an interface to many protocols such as shared memory, semaphores, and TCP.

There are many IPC protocols that have emerged since the concept of multitasking emerged. However, none of these communication protocols provide all the features that are desirable or necessary in multiprocess robotic systems.

1.4 Research Issues

This thesis is attempting to address many issues that are prevalent mainly in real-time robotic systems. RTC was designed to provide the following features:

- **Real-time capabilities for extremely high bandwidths.** RTC should allow processes to send large amounts of data to each other, typically at a rate greater than 1 megabyte per second.
- **Interprocess reliability.** When data is transmitted between processes, RTC should be able to guarantee that the data sent is received. That is, there will be no data lost during transmission.
- **Portability across many architectures and operating systems.** RTC should be able to be used under many different computer platforms (e.g. x86, 68k, MIPS, SPARC, etc.).
- **Flexibility for constantly changing or evolving systems.** With minimal effort, a user should be able to introduce new processes into a system which is interconnected by RTC.
- **Robustness during run-time.** RTC should allow processes to form or destroy connections with each other asynchronously. Processes interconnected using RTC should be able to start or stop without hanging or crashing the system.
- **Ease of use.** A programmer who is unfamiliar with networking should be able to quickly learn how to use RTC.
- **Stability.** RTC should never crash or directly cause an executable to stop running. RTC should be designed such that even if the user induces an error, RTC should merely report the error without crashing or intentionally exiting. Control should be left in the user's hands as to how errors should be handled.
- **Diagnostic capabilities.** A mechanism for tapping into the internals of RTC should be provided if a user-induced problem arises, helping the user to find the problem.
- **Low memory overhead.** RTC should require as little memory as possible, while still providing all the required functionality.
- **Variety of low-level protocols.** The user should be able to select from many underlying protocols, choosing the protocol best suited for each application.

The communication protocols discussed in the previous section fail to meet all of the above criteria.

Chapter 2

Design And Implementation

2.1 The Communication Protocol

TCP is a connection-based protocol, that provides a reliable byte stream for processes to talk to one another. The other internet protocol, UDP, is a connectionless protocol that provides an unreliable datagram service. UDP primarily provides speed, whereas TCP primarily provides reliability. In robotic systems, reliability is important. For example, suppose a robot lost control and was endangering the lives of nearby humans. In this situation, you would want to ensure that the an emergency stop message sent to the controller was actually received. Therefore, TCP would be a better choice for reliable message delivery.

UDP is reliable most of the time, but it is impossible to predict when a message, or part of a message, called a “packet”, might get lost during transmission. This does not appeal to systems that require reliable data transmission between processes. Also, UDP has the limitation of having message boundaries, whereas TCP does not. Therefore, sending images, which are usually large messages, over a network using UDP could be disastrous. Due to these message boundaries, an image message would need to be broken up into several smaller messages and transmitted separately. If one of these separate messages were not delivered, then the image being received would be invalid. To overcome this, a retransmission technique could be employed, but then the speed of UDP would be compromised. So TCP proves to be the best choice for reliable data transmission between processes, which is why the TCP protocol was selected in this work as one of the two protocols of communication between processes.

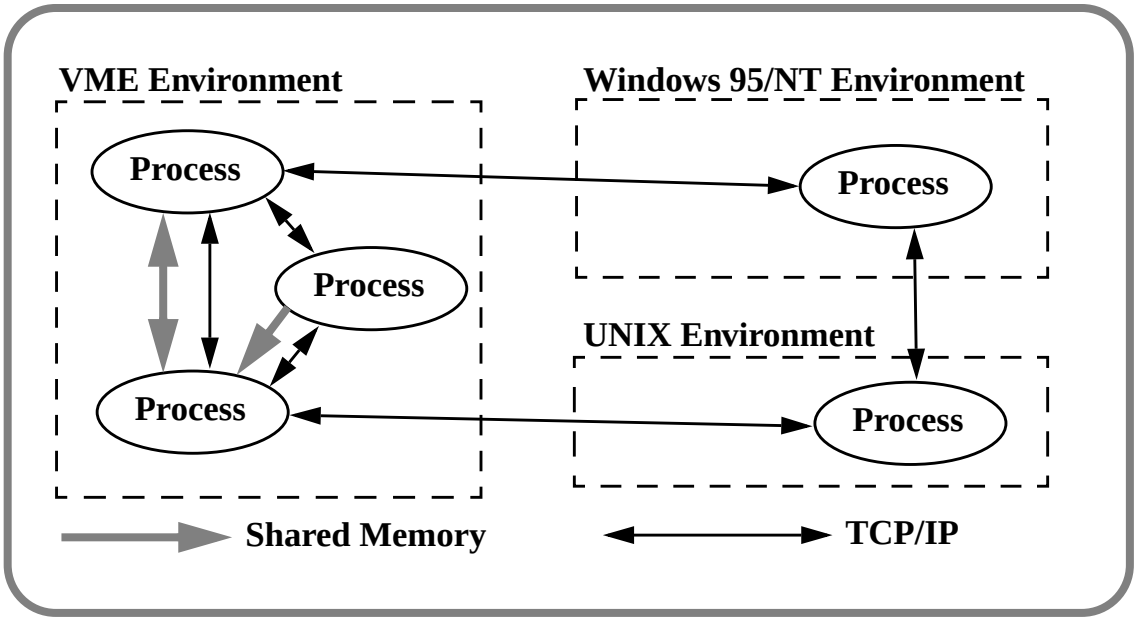
Remember that one of the primary goals of RTC is to combine the advantages of both multithreading and multitasking. Recall that the major advantage of multithreading is its ability to transfer large amounts data quickly between threads of execution. Shared memory is the communication protocol behind multithreading. IPC protocols such as TCP, cannot supply the bandwidth of shared memory. Therefore, for RTC to incorporate the advantage of multithreading, RTC must utilize shared memory as another protocol for transferring data between processes.

Most of today’s robotic systems are embedded in real-time environments such as VME cages. The backplane of VME cages is very conducive to shared memory implementations since the backplane is a high bandwidth bus. This bus is able to handle both TCP and shared memory bandwidth easily. Furthermore, VME cages are one of the most commonly used embedded system cages used today. Therefore, RTC was designed specifically with this technology in mind. Also, RTC was

originally designed for VxWorks, because it is currently one of the most prevalent real-time operating systems.

Although RTC was designed specifically for VME environments, it is not limited to this type of hardware. In fact, RTC was designed to be as flexible as possible, such that it can be ported to multiple hardware platforms. TCP makes portability simpler since most operating systems support TCP. Figure 2.1 shows an example of how TCP and shared memory can be used together to form a reliable real-time multiprocess system.

FIGURE 2.1. Example of a System Interconnected using RTC



RTC is capable of transmitting data at a rate of approximately half of a megabyte per second using TCP as the underlying protocol. Using shared memory, RTC is able to transmit data on the order of magnitude of 10 megabytes per second. Some specific results are provided in Chapter 3.

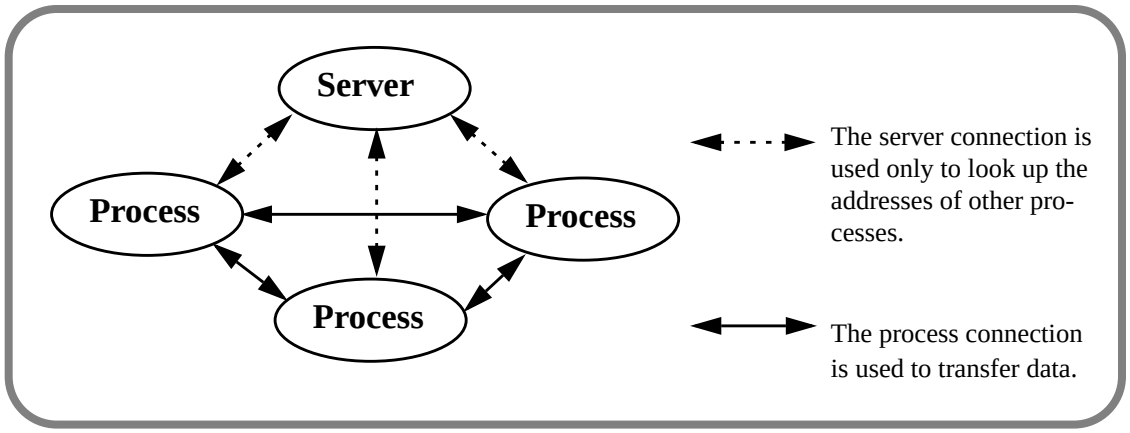
2.2 The Communication Architecture

The point-to-point architecture (see Figure 1.2) is the fastest way to transfer data between processes that are connected with each other over a network. A direct connection will always be faster than going through a server, as in a client-server model (see Figure 1.1). In the client-server model, the server process is really just

acting as a relay station, which could become a system bottleneck if overloaded. For this reason, this thesis project was designed to follow the point-to-point communication architecture.

If a process needs to transfer data to another process, it first needs to form a connection to the other process. To do this with TCP, the process would need to know exactly how to find the other process. Each process essentially has an address at which it can be found by other processes. This address consists of an IP address and a port number. The IP address is a numeric address which uniquely identifies a processor in a network. The port number is an identifier for talking to a specific process on a given processor. To hide low level information, like addresses and port numbers, a server may be employed. Note that this server is not used in the same manner as the server in a client-server model. Rather, the server is used merely for creating a flexible communication system. Figure 2.2 depicts the server relationship with all the system processes.

FIGURE 2.2. The Server-Process Relationship

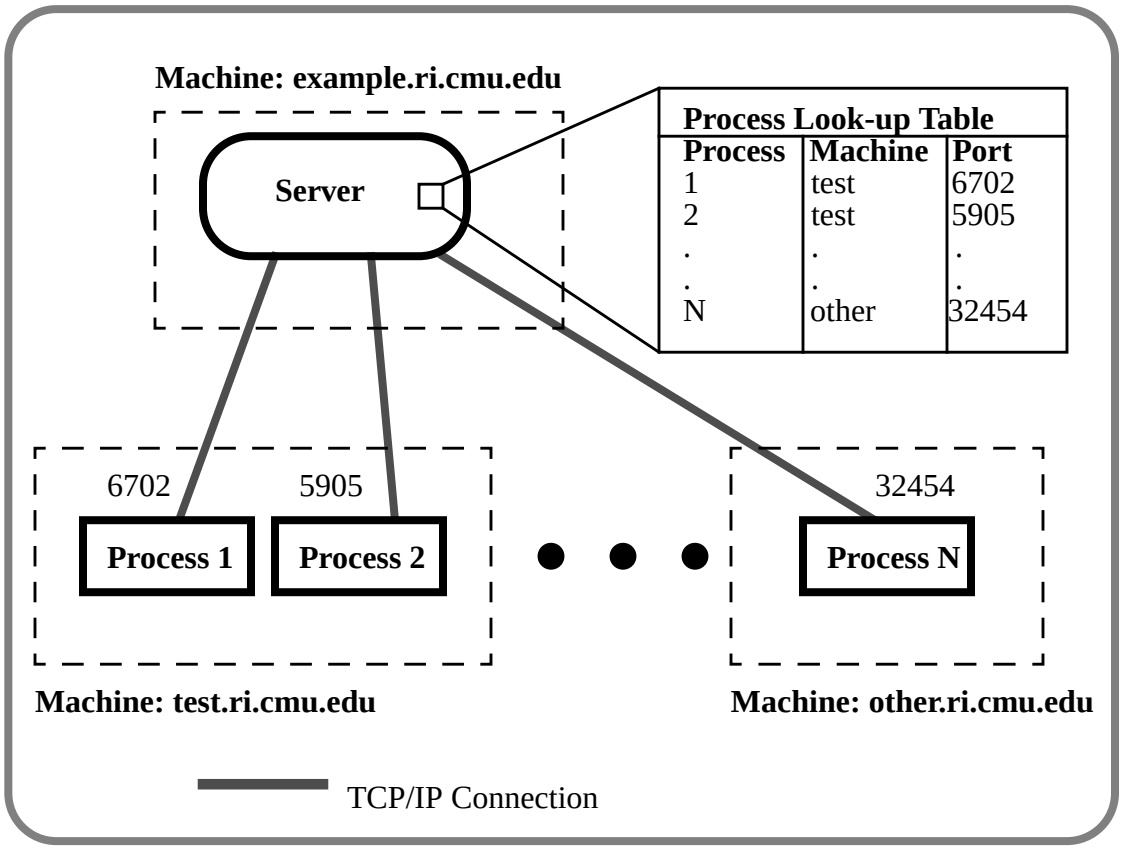


The server saves a process from having to know the exact address of every other process. Each process in a multiprocess system only needs to know the location of the server. So each process only has to modify one parameter, rather than many parameters. With the server, a process only needs to be supplied the IP address of where the server is running.

The server, in effect, is merely a look-up table for processes. When a new process is spawned, it first connects to the server. The process then registers all information about itself with the server. This information includes the platform on which the process is running, which port by which the process may be accessed, and the IP address of the process. The server stores this information in a table, and provides the connection information to any process that requests it. So, when a process

would like to form a connection to another process, it queries the server for the appropriate information, and when received, forms a direct connection to the other process. Figure 2.3 shows the server's role in a system using RTC.

FIGURE 2.3. The Role of the Server



The server was designed such that if a process of the same name tries to register with the server, the server will deny the second process connection rights. This way, the user can never accidently launch the same process simultaneously, which could cause system-wide confusion.

2.3 The Connection Model

Since RTC follows the point-to-point connection model, some issues need to be addressed. In a client-server model, no special communication multiplexing needs to occur within the processes, only in the server. The server in the client-server model must constantly be accepting new connection requests, whereas the pro-

cesses need not worry about multiplexing because they only connect and respond to the server. This is not so in the point-to-point model. In effect, each of the processes in a point-to-point model need to be servers themselves, handling multiple connection requests from other processes. How processes perform this task is discussed in Section 2.6.

One of the secondary goals of RTC is to create a flexible, robust communication toolkit. One aspect of a robust multiprocess system is the ability for processes to start and stop asynchronously. Processes should be able to be spawned and killed at random times without crashing or hanging the system. Furthermore, a robust process should posses the ability to request connections to other processes without waiting for those processes to register with the server.

For example, consider a newly spawned process that displays some status information in a graphical window. Suppose that this process wants to form a connection to another process which allows a user to manually change the display values in the graphical window. Further suppose that the first process, once spawned, must regularly be printing data to the graphical window, otherwise the process will fill up a buffer and eventually cause the process to hang or crash. In this case, the first process would not want to be waiting for the second process to be spawned. The first process would want to merely request a connection to the second process, and continue on its way, regardless of whether a connection to the second process was formed initially. Furthermore, the connection should be automatically formed whenever the second process registered with the server. Generally, this "non-blocking" capability helps avoid deadlocks during the connection phase of processes. RTC allows this non-blocking connection formation feature.

Not only does RTC allow for non-blocking connections, but it also allows for "blocking" connections. Blocking connections signify that a process *will* wait for the other processes to register with the server before continuing execution. Blocking connections are especially useful for synchronizing an event-based system. Non-blocking connections are especially useful for asynchronous systems, where processes cannot be dependent on other processes. RTC also allows for a hybrid of blocking and non-blocking connections.

All connections, whether blocking or non-blocking, are formed using TCP, primarily because it is a connection-oriented protocol. That is, explicit connections are required for the TCP protocol to transfer data between processes. Shared memory, on the other hand, does not require explicit connections to transfer data between processes. However, RTC models the shared memory protocol as explicit connec-

tions. These shared memory connections can really be thought of as “links” into each other’s memory.

RTC was designed such that every process must form a TCP connection to every other process with which it wishes to communicate. This is required because even with shared memory, low level information about other processes needs to be acquired. Since the mechanism for acquiring this data (through the TCP connection to the server) already exists, it is called upon again when making shared memory links between processes. The second reason for requiring every process to form a TCP connection is that if shared memory is not available, data which was to be sent through shared memory link can still be sent through the TCP connection. To demonstrate the usefulness of this feature, consider the case where code written for an embedded system requires debugging. Also suppose that a non-real-time development processor without shared memory capabilities is available for debugging the code. This feature would allow the real-time code, which normally communicates through shared memory, to be able to be run on the non-real-time processor without any code changes, provided the shared memory protocol will default to the TCP protocol if shared memory is not available. RTC was designed to provide this functionality. Note that even if a process primarily deals with shared memory, the TCP connection would present minimal overhead in terms of memory and, after the connection phase, no overhead in terms of performance.

2.4 Messages

2.4.1 Defining Messages

RTC uses a message-oriented paradigm to transfer data between processes. That is, data is sent from one process to another process as a “message”. This message representation is independent of computer architecture and the underlying protocol being used (TCP or shared memory). For a process to send a message to another process using RTC, a message must be defined. These definitions represent the C or C++ data structures being transmitted over the connection.

2.4.1.1 Message Format Grammar

Character strings are used to easily represent data structures. By following a simple grammar, a character string can be formed to represent any C data structure.

The grammar for defining messages in RTC is listed in Figure 2.4, where <format> represents a complete message format.

FIGURE 2.4. Grammar Rules For RTC Message Formats

<digit> := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<num> := <digit> <num> | <digit>

<primitive> := NULL | char | short | int | enum | float | long | double

<field> := <type>,<field> | <type>

<structure> := {<field>}

<array> := [<num>:<primitive] | [<num> <array>] | [<num>:<structure>]

<type> := <primitive> | <structure> | <array>

<format> := "<type>"

The grammar listed in Figure 2.4 adheres to the same style that many compiler grammars follow. This grammar can be used to completely describe the syntax, or form, of any RTC message. The := symbol can be read as "is defined to be". On the left of this symbol, surrounded by < and >, is the language construct being defined. On the right of the := symbol is the description of the syntax being defined for the language construct. The | symbol can be read as "or is defined to be" For example, the second to last rule in the grammar of Figure 2.4 can be read as "*type* is defined to be a *primitive* or is defined to be a *structure* or is defined to be an *array*".

Also note that this grammar has recursive properties. For example, the <num> language construct has a potential recursive element to it. To represent the number 1, the first and second rule are able to do this without recursion (see Figure 2.5). To represent the number 25, however, the recursive properties of the grammar are called upon. Figure 2.6 shows how the grammar can be used recursively to generate the number 25.

FIGURE 2.5. Nonrecursive Properties of the Grammar

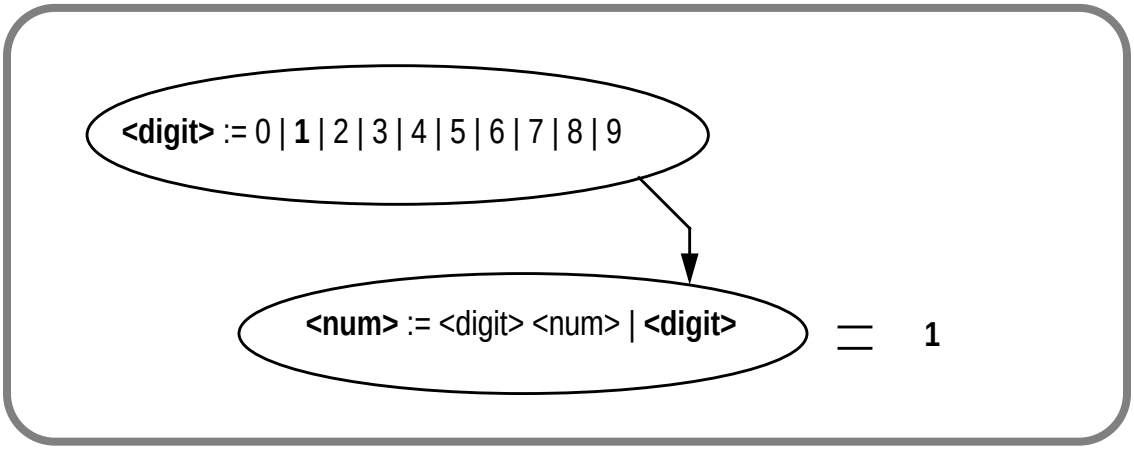
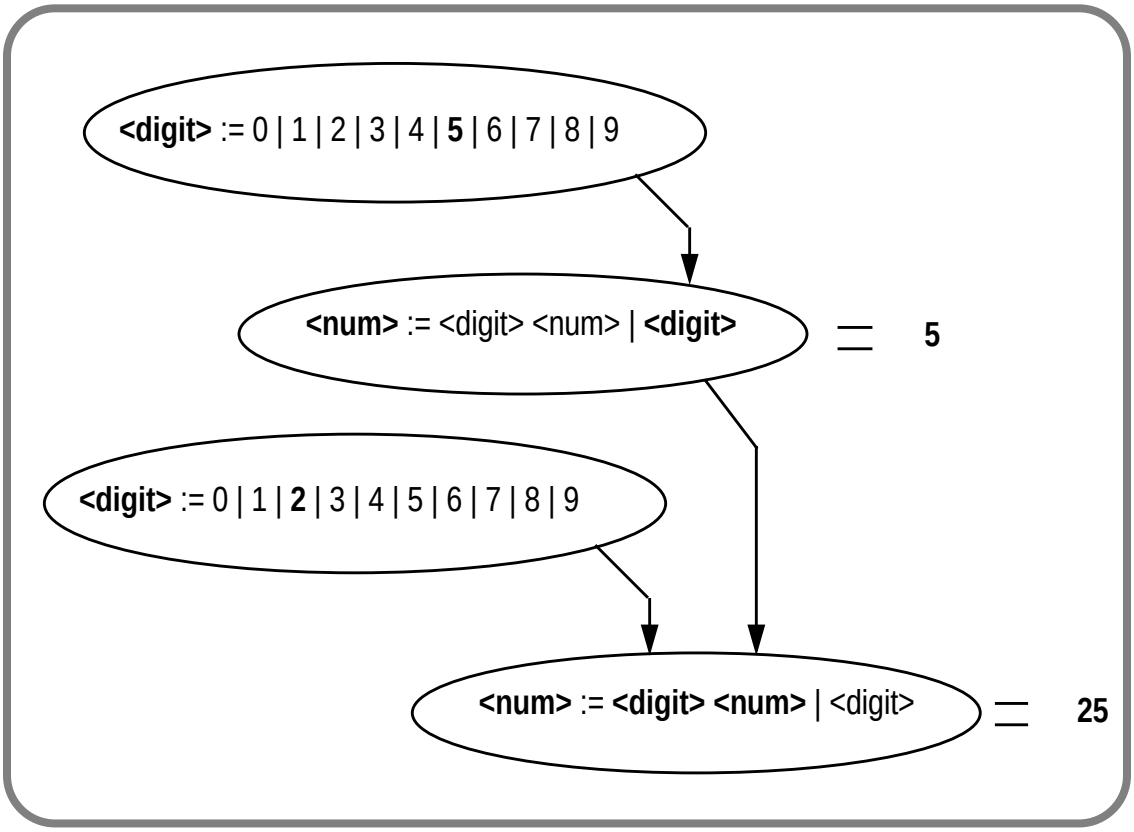


FIGURE 2.6. Recursive Properties of the Grammar



The grammar of Figure 2.4 is used to form a character string that accurately represents a C data structure that is to be sent using the RTC protocol. This character string representation will be referred to as a “message format string”. How these message format strings are generated is best shown through examples. Sections

2.4.1.2 through 2.4.1.4 illustrate how to form these message format strings through examples.

2.4.1.2 Primitive Data Types

The C programming language provides a set of simple data types. These types are: void types, characters, short integers, enumerated types, floating-point numbers, long integers, and double-precision floating-point numbers. The corresponding symbols as listed in the <primitive> rule respectively are: NULL, char, short, int, enum, float, long, and double. To form a message format string for a primitive data type, only three of the rules are required: the <format> rule, the <type> rule, and the <primitive> rule. Generally, primative data types are represented by the C primitive type name, surrounded by quotes. The exception to this is the NULL data type of course. NULL is a macro in C, not a data type. Figure 2.7 shows a few examples.

FIGURE 2.7. Message Format Strings for Primitive Data Types

The message format string for a message containing an integer would be:

"int"

The message format string for a message containing a floating-point number would be:

"float"

The message format string for a message containing no data would be:

"NULL"

2.4.1.3 Composite Data Types

The C programming language allows the user to define new data types. These new data types combine primitive data types together to form the desired data type. RTC supports most of these composite data types, which include any combination of structures, fixed length arrays, and primitive data types. Note that variable length arrays and linked lists are not included directly in the RTC grammar. However, RTC can be employed to transfer such structures as well. This is explained in Section 2.4.1.4.

Composite data structures are also easy to represent with the grammar presented in Figure 2.4. As with primitive message format strings, composite message format strings are surrounded by quotes. Structures are represented by a pair of braces surrounding a list of data types, composite or primitive, which are separated by commas. Fixed length arrays are represented by a pair of square brackets surrounding the dimension field and data type which are separated by a colon. Matrices are represented as nested arrays. The examples in Figure 2.8 demonstrate how to form message format strings for composite data types.

FIGURE 2.8. Message Format Strings for Composite Data Types

Consider the following composite data type:

```
typedef struct {
  int x;
  double y;
} TYPE1;
```

The message format string for a message containing TYPE1 data would be:

"{int, double}"

Consider the following composite data type:

```
typedef char STRING[80];
```

The message format string for a message containing STRING would be:

"[80: char]"

Consider the following composite data structure:

```
typedef struct {
  long time;
  int grid[640][480];
} TYPE2;
```

The message format string for a message containing TYPE2 would be:

"{long, [640[480: int]]}"

2.4.1.4 Dynamic Data Types

Dynamic data types are those which involve pointers to memory that was dynamically allocated. Dynamic memory allocation can be detrimental to a process if not handled properly. That is, every section of memory that is allocated by a process must be properly restored. If a process were to continually allocate memory without ever freeing it, the process would eventually experience a segmentation fault

and crash. Memory leaks such as this are even more drastic on real-time processors, possibly causing processors to hang or crash.

Most real-time operating systems do not provide the memory protection and clean-up ability of most non-real-time operating systems. Therefore RTC was designed such that dynamic memory allocation is kept to a minimum. During the design phase, whenever there was a choice between allocating static memory versus dynamic memory, the former prevailed.

With this knowledge in mind, RTC was designed for real-time embedded systems. Real-time systems rarely perform dynamic memory allocations. Static memory is always preferred in a real-time system, mainly for deterministic purposes. Given this, a design choice was made not to explicitly support dynamic data types. However, that does not mean that dynamic data types cannot be transferred as messages via RTC.

Dynamic data types can be sent as messages using RTC, they just need to be converted into a static structure which RTC can interpret. The user must pack and unpack dynamic data types. This packing and unpacking is trivial for the user to perform, but would be fairly complex for RTC to interpret. Moreover, the amount of time wasted by having RTC decipher, pack, and unpack messages would be far greater than having the user pack and unpack dynamic data types into a RTC-supported data type. Memory is cheap these days, so saving transmission time at the cost of memory is worth it. Finally, realize that a user would need to pack and unpack linked lists even if TCP or shared memory were being utilized directly. Therefore, RTC expects that all linked lists and variable length arrays to be converted into a RTC-supported data type.

Converting variable length arrays into a static data type is trivial. Figure 2.9 demonstrates how this conversion could be accomplished. Likewise, Figure 2.10 demonstrates how a linked list could be converted into a static structure that RTC could interpret. Once a static representation of the dynamic data structure has been created, then the data in the dynamic structure can be copied into the static structure, which can then be transmitted via RTC. Further note that a programmer could design all variable length arrays in the format presented in Figure 2.9 so that the array would require no special treatment. Linked lists, on the other hand, are extremely time consuming when represented as static structures, therefore conversion of linked lists is most likely inevitable.

FIGURE 2.9. Representing a Variable Length Array

A variable length array of doubles could be statically represented by the following data structure:

```
typedef struct {
  int length;
  double elements[5000];
} VAR_TYPE;
```

The “elements” field is the array of doubles. The “length” field indicates how many elements of “elements” are valid. Note that the size of the array is limited to 5000 doubles in this example, but it could be set to any length.

FIGURE 2.10. Representing a Linked List

Consider the following data type and variable declaration:

```
typedef struct _LINK_TYPE {
  int value;
  struct _LINK_TYPE* nextP;
} LINK_TYPE, *LINK_PTR;
LINK_PTR list;
```

Suppose that “list” is grown into a linked list. To represent “list” in a format acceptable to RTC, the following data structure could be used:

```
typedef struct {
  int length;
  int values[250];
} LIST_TYPE;
```

The “values” field contains the values in the linked list. The “length” field indicates how many of the values in “values” are valid. Note that the length of this static linked list structure can be any length, but is set to 250 in this example.

2.4.1.5 Lexical Analysis

Once message format strings are generated, the built-in parser of the RTC protocol can parse these strings in order to know exactly how to pack and unpack the data associated with each message. Parsing of messages only needs to be done once, during the initialization phase of a process. This speeds up transmission time because RTC looks up the pertinent information, rather than recalculating it each time a message is to be sent between processes.

2.4.2 Message Queues

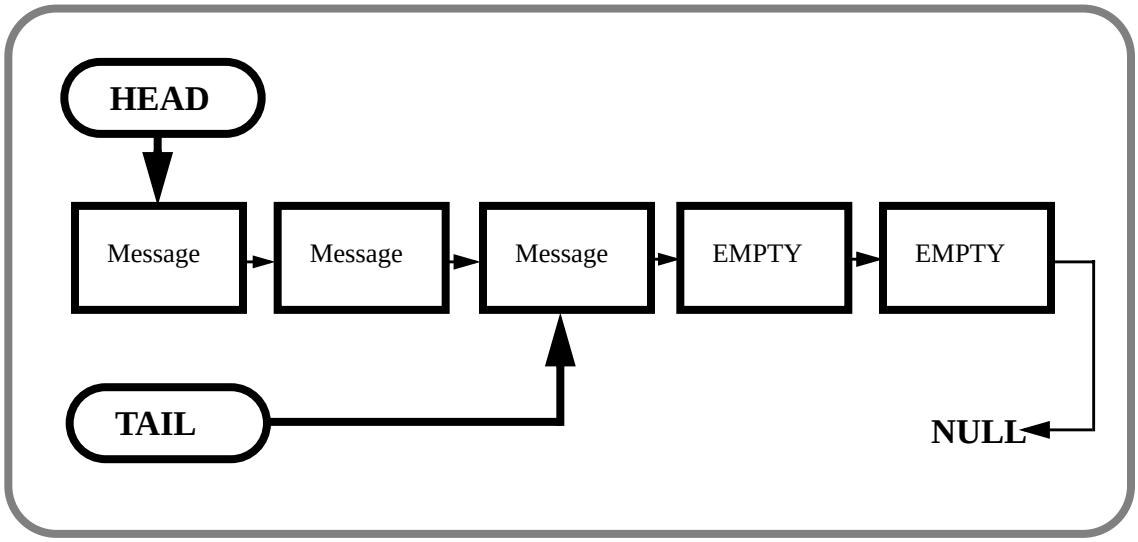
2.4.2.1 TCP Message Queues

Data is sent over a TCP connection as a data stream of bytes. The receiving end of this stream buffers any unread data. This buffer could become full, thus blocking the sending end from transferring any more data until bytes are read out of the receiving buffer. To prevent the buffer from becoming saturated, RTC maintains its own buffer which has no boundaries, unlike the TCP buffer size. RTC reads any available data from all the TCP connections as quickly as possible. This data is then copied directly into the RTC buffer.

The RTC data buffer takes the form of message queues. There is a separate message queue for each type of message. By having a dedicated queue for each type of message, RTC is able to minimize the number of dynamic memory allocations required to maintain each message queue, thus increasing throughput.

For each message type, the initial queue length allocated is five messages long. Note that efficient systems would rarely allow messages to back up in the queue to this length. But it is conceivable that a process may get so bogged down sometimes that messages may begin to accumulate in their respective queues. If this happens, RTC will grow the message queues to the required length until the process is able to handle messages again. In the case where a new message is received which is intended for a queue that has become saturated, RTC will extend the queue length by another five elements. This operation will be repeated as necessary. Figure 2.11 depicts a typical RTC message queue.

FIGURE 2.11. A Message Queue



The memory allocated for message queues is never restored until just before the process stops execution. This is possible by always keeping a pointer to the head and tail of each queue. If the queue shrinks in size, no memory restoration is required, only redirection of the tail pointer is needed. By never needing to restore memory at run-time, message transmission time is reduced.

Note that this message queuing model in unconventional for two reasons. Firstly, most queues grow and shrink continuously, allocating and restoring memory constantly, which affects performance. With the RTC message queues, queues are grown to an initial length of five during the initialization phase of a process. Only in rare occasions will the queue need to be grown during run-time. So in the best case, memory allocation and restorations only occur during the initialization and shutdown phases of a process, and in the worst case, memory allocations may occasionally occur during run-time. Secondly, this queuing model differs from conventional communication queues in that there are multiple queues, one for each type of message. Since the same type of message always is stored in the same queue, retrieval of messages in faster than the conventual model of having only one queue for every type of message. With the RTC queuing model, a memory trade-off was made so that the queue does not need to be searched for the type of message being sought after.

When a message is copied from the TCP buffer into the appropriate RTC message queue, the name of the process which sent the message is tagged to the message. Therefore, if multiple processes are sending the same type of message to the same process, a means exists for extracting a message of that type from a particular pro-

cess. This case dictates that the queue must be searched for the first instance of the message with the desired process name tag. Since queues follow the “first in, first out”, or FIFO, concept, the search time is minimal to find the message which arrived first. Messages read out of an RTC message queue are always provided in the order in which they were received.

There are two additional features built into the RTC message queuing model. Firstly, a process can limit the length of any message queue to one message. This capability guarantees that the message in that queue is always the most recent message received. That is, with a queue length of one message, that message will continually be overwritten as new messages are read off the TCP buffer. Secondly, to save memory, a process can eliminate any message queue. This is useful if the process is merely a sender of a message. Since the process only sends the message, never receiving it, there is no need for a queue of that message type.

2.4.2.2 Shared Memory Queues

Messages sent over a shared memory link are stored in message queues separate from the TCP message queues. Some real-time operating systems implement their own shared memory message queues. Since shared memory messages are supposed to be fast, a design choice was made to separate the TCP message queues from the shared memory message queues. This way, when running under real-time operating systems which already support shared memory message queues, RTC can work directly with those queues, rather than making an extra copy from their message queue into another RTC message queue. However, for real-time operating systems which do not provide built-in shared memory queues, this extra copy is needed.

2.4.3 Registering Messages

Before a process can send or receive a message, the message must first be registered. A message has four attributes which need to be registered. Firstly, each message is given a name, which is a character string. This string is not used for transmission. Rather the message name is merely there to provide the user with textual information if an error should occur. Secondly, each message must be assigned a unique integer greater than or equal to 0. The maximum number of messages is currently set to 100, but can be easily changed to meet system requirements. When sending and receiving messages, the message ID is used. The ID is really an index

into a large table containing information about the type of message and the queues for that message type. Thirdly, a message format string, as described in Section 2.4.1 of this chapter, needs to be bound to the message ID. Finally, information about how the message is sent or received needs to be registered. That is, the type of message queue needs to be specified. As discussed in the previous section, the queue can be a normal FIFO queue, a queue one element long which always guarantees the most recent message, or an empty queue (provided the process is only a sender of the message).

Message registration occurs during the initialization phase of a process. This way, all the message queues and information about the messages are generated before the main execution loop of a process is invoked, thus moving potentially time-consuming operations outside the main execution loop.

2.5 Portability

RTC was designed to run simultaneously under multiple computer architectures. Consequently, RTC supports all the different byte-ordering and alignment models used by various architectures. RTC was also designed to support different operating systems as well as the architectural differences.

There are several paradigms for converting data from one architecture into a form understandable by another architecture. One of the more common methods first converts the data into a standard form, which is in turn is converted into the target form. This method is common in many communication protocols. The sending process first converts the data into a “network” form. The receiving process then translates the data from the network form into its own form, both in terms of alignment and byte-ordering. Obviously, if the network form is identical to the local representation, no conversion occurs. But consider the case when both the sending and receiving forms differ from the standard network form. In this case, two conversions are occurring when maybe only one or no conversion needs to happen. This methodology is followed because many protocols do not have the luxury of knowing the architecture with which they are communicating. So with this model, a process would only need to know how to convert data into network form.

RTC does not perform any conversions to a standard network form. Rather RTC follows the rule that it is the responsibility of the sending process to convert the data into the correct representation for the receiving process. This methodology is possible because each process has knowledge about the architectures of the other

processes. Remember from Section 2.2, that this information is gathered whenever a connection request is made to the server.

To convert data into the appropriate form, a complete mapping of the data's structure is needed. Fortunately, this mapping is created during message registration. While RTC parses the message format strings, it builds an efficient data structure that completely describes how the data is structured. Since registration is performed during the initialization phase of a process, the message format string is parsed only once and the internal mapping is formed only once. This mapping serves as a quick reference chart of how to break up the data into the appropriate pieces. These pieces can then be modified or rearranged to conform to the target data form.

Note that if the receiving process has the same byte-ordering and alignment as the sending process, the data is not converted. In this case, the data can be directly transferred.

2.6 Control Flow

RTC gives the user full control of execution. There are two conceptual communication control loops that the user must handle. The first is the TCP control loop. The second is the shared memory control loop. Although these two loops are conceptually independent of each other, they must be consolidated into one synchronized control loop.

The TCP control loop is based on monitoring what are called "sockets". A socket defines a complete route between two processes. Each process has a different socket for every one of its connections. For example, if a process is connected to the server and three other processes, then that process will have four sockets, recording exactly how to communicate with each of its four connections. A process can either listen or talk to any of its sockets, which is how it communicates.

By listening to all of its sockets, a process can determine if any activity happened on those sockets. More specifically, by listening to a socket, a process can easily determine if any data was sent to that socket. A process performs this listening task by looking directly at a set of hardware bits. By watching these bits, a process can sleep and not need to poll, preserving precious CPU cycles, until some activity occurs on one or more of its sockets.

A process may also talk to all of its sockets by sending messages across them. This operation is always non-blocking. Therefore, it is not a large factor for synchronization.

The TCP control loop consists of two primary stages. The first stage is for a process to listen to all of its sockets as described above. The second stage requires a process to handle all of the "events" that occurred on its sockets. An event signifies that some sort of activity happened on a socket. An RTC process handles four types of events which are listed in Table 2.1.

TABLE 2.1. RTC Events

Event	Description
New Connection	Another process is attempting to connect to the process.
New Message	A message was sent from another process.
Disconnection	Another process to which the process was connected died.
Server Request	A request message was sent from the server.

The user must guarantee that these events are continually handled. This handling is all accomplished internally, but the user must ensure that an RTC function which performs this handling task is called on a regular basis. There are several functions in RTC which take care of this internal event handling. Thus, it is fairly easy to guarantee continuous event handling. However, when merging a shared memory control loop into an existing TCP control loop, conflicts can emerge which can cause dead-lock or missed messages.

Shared memory links are not based on sockets. Unfortunately there is no way to watch all shared memory connections simultaneously. This dictates that a polling mechanism must be employed when multiple shared memory connections exist for a process. Therefore, when a process uses both TCP and shared memory, the non-polling TCP functions must be occasionally interrupted so that the shared memory links may be polled for new messages. This interruption is made possible by providing time-outs to all the data transmission functions. With this time-out capability, the shared memory and TCP control loops, although conceptually different, can be easily merged into one synchronized control loop.

2.7 Sending and Receiving

Once messages are registered and connections are formed, transmitting messages between processes is straightforward. Since a point-to-point connection is established between two processes that want to transfer messages, a message can be sent directly from one process to another, without having to pass through a server.

RTC provides two methods for transferring messages between processes. The first method only utilizes the TCP protocol to transfer messages. The second method uses shared memory as its primary protocol, but defers to TCP if shared memory is not available. This second method allows the communications to be viewed as a "black box" to the user. In essence, this second methodology will always try to use the fastest protocol available.

To send a message from one process to another, all that is required is the name of process (as was obtained during the connection phase), the type of message (as was registered during the initialization phase of the process), and a pointer to the data that is to be sent. Realize that this pointer is to the user's static data structure (Section 2.4.1.5 discusses dynamic data structures), so the user does not need to do anything special to the data being transferred. RTC takes care of the packing and unpacking of data.

To receive a message from a sending process, the receiving process needs to know the name of the sending process and the type of message. The receiving process must supply a pointer to the data structure into which the data should be copied. The receiving process may also supply a time-out. That is, if the message is not received within a specified time interval, RTC will stop trying to receive the message, allowing the receiving process to perform other actions. Of course, the receiving process can try receiving that message again at a later time. Time-outs are useful when processes need to perform many calculations and cannot afford to wait for messages which may be delayed temporarily.

Sometimes a receiving process does not care which module sent a particular message to the receiving process. In this case, the receiving process does not want to specify the name of the sending process when receiving a message. RTC allows a receiving process the ability to receive a particular message from *any* sending process. This feature is applicable in a control arbitration scheme as is sometimes found in control systems for robotics. In this scenario, many processes make votes on how to control the robot, and a controller or "arbiter" process compiles these votes and decides which command will best control the robot. In this scheme, the

arbiter process would just read command messages from any planning process that wishes to vote for the way that the robot should be controlled. This way, the arbiter process does not need to know anything about which processes are connecting to it.

Before a process sends an actual message to a receiving process, the sender must inform the receiver about the data that is to be transmitted. To do this, the sender first sends a header message to the receiver. This small header, the size of one integer, contains the ID of the message that is about to be sent. When the receiving process receives the header, it knows exactly which message queue in which to place the actual message and the size of the actual message, meaning that it knows exactly how many bytes to read off the socket or shared memory location. Remember that it is the sender's responsibility to put the data in a form understandable to the receiver, so the receiver never needs to unpack or reformat the incoming data.

2.8 Publishing and Subscribing

There is one minor problem with the point-to-point paradigm. Namely, a process must know about every other process to which it is going to connect. However, this problem is easily bypassed by incorporating a "publisher-subscriber" methodology. This methodology, as will be explained, emulates a connectionless network.

Normally, every sending process needs to explicitly make a connection to every receiving process to which the sending process intends to send a message. Consider the case when a sending process wants to send data to all those receiving processes which require the data, but the sending process does not know how many processes will require the data, or even worse, does not know the name of every process that requires the data. In this case, the sending process cannot form explicit connections to every potential consumer of the data.

To overcome this explicit connection problem, the concept of a publisher and subscriber was introduced into RTC. The publisher maintains a list of all those processes which have subscribed to the publisher's data. This list is initially empty. The publisher has certain data that other processes want. If these other processes need this data, they inherently know the name of the publishing process, so the subscribing processes can make explicit connections to the publisher. This means that the publisher never needs to know the name or number of processes connecting to it.

When a subscriber subscribes to the publisher, it forms a connection. The information about the subscriber is stored in a list. If a subscriber shuts down or unsubscribes, the subscriber is removed from the publisher's list. When a publisher publishes its data, it cycles through its list of subscribers and delivers the data directly to each of the subscribing processes.

This publisher-subscriber or "producer-consumer" model is available over both the TCP and the shared memory protocols. Recall from the previous section that a receiving process can also be configured such that it knows nothing about the processes sending data to it. That methodology coupled with the publisher-subscriber model allows RTC to emulate a connectionless architecture.

2.9 System Shutdown

A system interconnected using RTC can be spread over many processors. Large systems may have fifty or more processes, distributed over many processors. When a system of this magnitude is ready to be shut down, the user must destroy every process. With such a large system, this task could be extremely tedious. To overcome this potential headache, a shutdown process, the "RTC terminator", can be launched to shutdown all system processes.

The RTC terminator can perform one of three operations. Firstly, it can destroy the RTC server only. Secondly, it can destroy a specified process which is connected to the server. Finally, it can destroy every process which is connected with the server. This final option would be useful for shutting down a large system of processes.

Not only is the terminator useful for shutting down large systems, but it is useful for shutting down a system running within a real-time environment. For example, several processes running under a real-time operating system, such as VxWorks, would need to be shut down individually. This would mean that the user would need to remotely access each processor and shut down each process manually. If there are many processors within the real-time environment, this task could be extremely tedious, especially during system development. By using the terminator, all processes could be terminated nearly instantaneously from a remote location, eliminating this otherwise tedious operation.

Chapter 3

Results

3.1 Autonomous Excavation

The Autonomous Loading System (ALS) [stentz98] at the National Robotics Engineering Consortium (NREC) is developing technologies for automating mass excavation. Typically, a loading machine, such as a hydraulic excavator, digs material from a face and dumps the material in a truck. ALS has verified these core technologies in simulation for two loading machines - an excavator and a wheel loader. Moreover, ALS has automated a real excavator, as depicted in figure 3.1.

FIGURE 3.1. An Automated Excavator Loading a Dump Truck

Over the years ALS has developed into a sophisticated robotic system, consisting of 12 processes running on 8 processors. ALS processes communicate with each other using RTC. The TCP protocol is the primary underlying protocol employed, but ALS also relies heavily on the shared memory capabilities of RTC. That is, RTC was necessary to handle the large bandwidth of data being generated from onboard ladar sensors. ALS required a total bandwidth of approximately 3 megabytes per second.

ALS rigorously tested RTC on many different computer platforms. The operating systems and computer architectures that RTC needed to support for ALS are listed

in Table 3.1. ALS still pushes the development of RTC, requiring RTC to be ported to even more architectures. Future ALS components will require RTC to run on HP workstations, the VRTX real-time operating system, DOS, and 32-bit DOS.

TABLE 3.1. Platforms Supported by RTC for the ALS Project

	Sun	**SGI**	**68k**	**MIPS**
SunOS 4.x	X			
Solaris 5.5.x	X			
Irix 5.3		X		
Irix 6.2		X		
VxWorks 5.2			X	X

ALS has used RTC as its primary protocol of communication for years, over which time hundreds of trucks have been loaded autonomously. RTC provides a robust, reliable, real-time protocol, making a complicated robotic system like ALS possible.

3.2 Autonomous Underground Continuous Mining

Technologies for automating a continuous mining machine are being developed at the NREC. This project involves the automation of a real 12CM mining machine (see Figure 3.2), supplied by Joy Mining. This project utilizes simulators for testing new software before trying it on the real hardware. RTC is used to interconnect processes both for the simulated and real system.

FIGURE 3.2. A Continuous Mining Machine Cutting Coal

The platforms which RTC is currently supporting for the Joy project are listed in Table 3.2.

TABLE 3.2. Platforms Supported by RTC for the Joy Project

	Sun	SGI	68k
Solaris 5.5.x	X		
Irix 5.3		X	
Irix 6.x		X	
VxWorks 5.2			X

3.3 Autonomous Harvesting

The Demeter [hoffman96] project, also at the NREC, recently adopted RTC as its primary communication protocol. The Demeter project is developing the next generation of autonomous hay harvesters (see Figure 3.3) for agricultural operations.

FIGURE 3.3. An Autonomous Harvester Cutting Alfalfa

The Demeter project helped RTC expand its portability. Specifically, RTC was ported to Linux and Windows NT to accommodate their system. The computer platforms which RTC supports for the Demeter project are listed in Table 3.3.

TABLE 3.3. Platforms Supported by RTC for the Demeter Project

	Sun	**68k**	**x86**
Solaris 5.5.x	X		
VxWorks 5.2		X	
Linux 2.0.x			X
Windows NT 4.0			X

3.4 Synthesis Tools for Robot Configurations

A doctoral thesis in Robotics at Carnegie Mellon University has employed RTC for its computational needs for the past year. This doctoral thesis is developing synthesis and optimization tools for robot configurations [leger98]. This work is based on genetic programming which by nature is computationally expensive. Consequently, the workload must be distributed across many processes and processors. The system uses RTC for communications between a central server and approxi-

mately 50 identical evaluation processes distributed over a network of workstations. The platforms supported by RTC for this work are listed in Table 3.4.

TABLE 3.4. Platforms Supported by RTC for Genetic Programming

	SGI
Irix 5.3	X
Irix 6.x	X

3.5 Performance Results

The objective of these performance tests was to get a general idea of the speed of RTC both in real-time and non-real-time situations. The first performance test involved two Sun SPARC 20 workstations talking via TCP over a T1 ethernet connection. Messages of various sizes were sent between processes running on the two machines. The results of this test are listed in Table 3.5. Note that the network bandwidth limitation becomes obvious with messages greater than 10,000 bytes in size.

TABLE 3.5. Performance between Sun SPARC 20 workstations

Message Size (bytes/message)	**Bytes/second**	**Messages/sec**
4	24,017	6,004
40	235,111	5,877
100	522,633	5,226
400	930,035	2,325
1,000	971,472	971
4,000	985,942	246
10,000	992,004	99
40,000	992,085	24
100,000	992,126	9
400,000	992,171	2
1,000,000	992,284	0

To demonstrate RTC operation without the network limitation, the same set of messages was sent between two processes running on the same SGI MIPS R10000 processor. In this scenario, transmission occurs internally, avoiding the limitations of ethernet. However, different limitations are imposed on this performance test. Now, two processes are competing for CPU cycles since they reside on the same processor. The results of this performance test are listed in Table 3.6.

Generally, as message size increases, throughput (bytes/second in tables 3.5 - 3.8) increases. There is a great deal of overhead sending a message via TCP. Suppose that we want to transfer 4 megabytes of data from one processes to another. If we were to send that data incrementally using 1 megabyte messages, we would need to send only four messages, whereas if we were to send the data incrementally using 4 byte messages, we would need to send 1 million messages. In the latter case, the size of the message is dwarfed by the overhead of TCP, whereas in the former case, the overhead of TCP is insignificant when compared to the magnitude of a 1 megabyte message. Of course these two messages are extremes, but they help illustrate the point that the natural behavior of RTC favors larger messages for greater throughput.

However, this natural increase in throughput is interrupted in Table 3.6. Note that transmission rates for messages greater than 40,000 bytes begin to decrease. This artifact demonstrates the limitation imposed by the TCP buffer. The TCP buffer specifies how many bytes may be buffered before communications will block. Every architecture has its own allowable TCP buffer size.

If a message is smaller than the TCP buffer size, it can easily fit into the buffer, provided that the buffer is not already saturated. However, if the message is larger than the amount of space currently available in the buffer, the messages must be broken down into smaller messages and sent incrementally. This message fragmentation, which is handled by the IP layer, slows throughput. The more a message must be broken down into smaller pieces, the longer it takes to transfer that message. This effect is clearly seen in Table 3.6. One can determine that the TCP buffer size is somewhere between 40,000 and 100,000 bytes (the size was actually 50,000 bytes).

TABLE 3.6. Performance on a SGI MIPS R10000 Processor

Message Size (bytes/message)	Bytes/second	Messages/sec
4	48,495	12,123
40	456,393	11,409
100	858,785	8,587
400	3,218,296	8,045
1,000	7,984,436	7,984
4,000	15,411,469	3,852
10,000	20,049,633	2,004
40,000	35,583,420	355
100,000	32,133,831	321
400,000	30,238,266	75
1,000,000	19,497,693	19

The next performance test involved two MIPS 4700 processors running inside a VME cage. The processes sent the same set of messages that were sent in the previous performance tests, however, in this test, the shared memory capabilities of RTC were called upon. The two processes communicated by accessing each other's memory via the VME bus. The results of this performance test is listed in Table 3.7.

TABLE 3.7. Performance between MIPS 4700 Processors

Message Size (bytes/message)	Bytes/second	Messages/sec
4	21,978	5,494
40	216,216	5,405
100	528,169	5,281
400	1,643,835	4,109
1,000	2,790,697	2,790
4,000	3,934,426	983
10,000	4,124,420	412
40,000	4,225,352	105
100,000	4,301,175	43
400,000	4,761,905	11
1,000,000	5,625,000	5

The final performance test is similar to the previous test. However, in this test, the two processes reside on the same MIPS 4700 processor. In this scenario, the processes can tap into each other's memory without having to pass through the VME bus. The results of this test are listed in Table 3.8. Just as in the second performance test, the two processes are now in competition for CPU cycles. Each task was given the same priority.

TABLE 3.8. Performance on a MIPS 4700 Processor

Message Size (bytes/message)	Bytes/second	Messages/sec
4	38,504	9,626
40	304,440	7,611
100	738,916	7,389
400	2,727,273	6,818
1,000	6,250,000	6,250
4,000	12,500,000	3125
10,000	13,793,103	1379
40,000	16,243,655	406
100,000	17,241,379	172
400,000	17,628,541	44
1,000,000	18,404,908	18

Benchmarking interprocess communication protocols is rarely performed, mainly because there is no way to predict the behavior of most networks. Transmission times of data across the Internet are nondeterministic for many reasons. Some of the factors contributing to this nondeterministic behavior are:

- **Bandwidth.** The amount of data that can be sent across ethernet during the same amount of time differs depending on the type of ethernet. For example, a system networked with a T1 ethernet cable will not be able to support as much bandwidth as a T3 ethernet cable.
- **Network Traffic.** Transmission times vary based on the current amount of data being sent across the network. For example, if several users decide to FTP large files across the network at the same time, most likely collisions will occur, requiring retransmission of data packets, thus slowing each transmission.
- **Gateways.** Processes communicating across large distances most likely go through gateways. These gateways can sometimes act as bottlenecks reducing transmission times.

- **Various Platforms.** There are many different computer architectures connected to the network. Some architectures are able to handle networking better than others. Transmission of data from one computer to another computer across the network is limited by the slower computer.
- **Workload.** Sometimes a single processor hosts many processes communicating across a network. If this processor becomes so bogged down by handling so many tasks, the transmission times of data to and from these processes will be negatively affected.
- **Application.** Even if a processor only hosts a single process, transmission times are affected by the function of the process. For example, if a process handles an interrupt-driven serial line in addition to talking via the internet, the serial line can interrupt the receipt of data across the internet.

With this in mind, realize that the performance results presented in this section are merely ballpark figures. One must extrapolate how these results would map into another networked system.

3.6 Supported Platforms

RTC currently supports many operating systems and architectures, as listed in Table 3.9. RTC is still broadening its scope to support new platforms. Some examples of platforms that will eventually be supported are listed in Table 3.10.

TABLE 3.9. Currently Supported Platforms

	Sun	**SGI**	**68k**	**MIPS**	**x86**
SunOS 4.x	X				
Solaris 5.5.x	X				
Irix 5.x		X			
Irix 6.x		X			
VxWorks 5.x			X	X	
Linux 2.0.x					X
Windows NT 4.0					X

TABLE 3.10. Additional Platforms to be Supported

	HP	68k	x86
HP-UX	X		
VRTX		X	
VxWorks			X
DOS			X
Extended DOS			X

Chapter 4

Conclusions and Future Work

4.1 Conclusions

RTC has proven to be the reliable, robust protocol that it was designed to be. RTC has undergone extensive use during its growth over the past few years. RTC is able to support the required bandwidth of sensors such as ladar, as well as provide the flexibility needed for developing robotic systems. RTC has proven to be an excellent development tool for large robotic systems, allowing many components of a system to be integrated quickly and easily. RTC provides an easy means to separate system tasks so that system developers can work separately.

By incorporating the features discussed in Section 3.2, RTC will be a complete protocol able to handle most of the interprocess communication issues that exist in robotic systems. Even without the additional features, in its current form, the RTC protocol attained all of its design goals, as discussed in Sections 1.2 and 1.4.

4.2 Future Work

4.2.1 Addition of the UDP Internet Protocol

Currently RTC makes use of shared memory and TCP for transferring data between processes. As mentioned in Section 2.1, TCP sacrifices speed for reliability. Remember that UDP, on the other hand, sacrifices reliability for speed. A nice addition to the RTC protocol would be to have UDP as yet another underlying protocol which a user may choose.

By adding UDP as another underlying protocol, RTC becomes even more flexible. This addition will allow the user the ability to send messages between remote processes, as does TCP, but even faster. However, the user must realize that UDP is unreliable. So messages sent using UDP, cannot be crucial messages. For instance, a good application for using UDP would be one process sending non-critical status messages to another process at a high data rate, where occasionally dropping messages is acceptable.

For example, consider a robot with a an embedded system housing several processors. Suppose that a wireless radio ethernet connection exists between a processor in this embedded system and a processor in a remote control station. Suppose that one of the processes on the robot continually sends noncritical status information to

a process in the control station via the wireless ethernet. If this status information is sent using TCP, problems could exist since TCP is so reliable. That is, if for some reason, the message is unable to be sent to the receiving process, the sending process will retransmit the message over and over until the message is received by the receiving process. This operation is blocking, meaning that the sending process will not be able to continue execution until the message is successfully sent. In essence, this scenario could result in a system deadlock if the wireless ethernet stop functioning temporarily or, even worse, permanently.

In the above example, UDP would solve the problem of the wireless ethernet failing since UDP does not care if the receiving process actually receives the message or not. UDP never retransmits messages. In the above example, if the status message were sent via UDP to the process in the control station, the sending process could continue to operate normally, regardless of whether the wireless ethernet connection failed. For this reason, the addition of UDP would make the RTC protocol be more complete for a wider variety of robotic systems.

UDP would be fairly easy to implant into the current RTC design. Although UDP is a connectionless protocol, like shared memory, it would be modeled as a connection-oriented protocol, as was done with shared memory. UDP would present one problem however. Remember from Section 2.1 that UDP has message boundaries. Large messages must be broken down into smaller messages. Therefore, to keep with the current model of hiding the low level networking code from the user, a way of automatically breaking large messages into smaller ones must be created. That is, RTC must be able to pack and unpack larger messages sent via the UDP protocol.

Packing and unpacking large messages sent via UDP presents another design decision. Small messages, within the UDP message size boundary, present no problem as no packing or unpacking is required. The large messages, however, must be broken down into smaller messages, or packets. But if one of these packets is not successfully delivered to the receiving process, then the *entire* message is invalid. Therefore, for large messages, a verification method needs to be developed to determine if any of the large message's packets were lost. This verification would slow down the transmission speed greatly. So the advantage of using UDP, namely the speed, has been compromised.

There are two obvious solutions. The first solution could be to only allow the user to send small messages via UDP. The second solution would allow the user to send any size message via UDP with a verification scheme in effect, with the under-

standing that large messages will not gain a large transmission increase, if any, over TCP. RTC would be designed to follow the second solution. The verification could be extremely simple, such that no two-way communication is required for verification. This way, the transmission time would most likely still be faster than the extremely reliable TCP protocol.

4.2.2 Allowing Dynamic Messages

As mentioned in Section 2.4.1.4, RTC does not explicitly support dynamic data structures such as linked lists, variable-length arrays, and trees. Currently, RTC requires the user to pack and unpack dynamic data structures into a static data structure. This packing and unpacking should be hidden from the user. This automatic packing and unpacking could easily be incorporated into RTC. Note, however, that this option would only be added for non-real-time systems. This feature was not originally incorporated because dynamic data structures are not common in real-time systems. However, adding this feature would allow RTC to be used more generally, outside its primary real-time domain.

For RTC to support dynamic data structures, a few more rules would need to be added to the grammar. One of the advantages of the RTC message queuing model would be compromised for dynamic messages. That is, queues for handling such messages would need to be grown and shrunk on the fly. Since the messages are dynamic, that means that there is no way to know the size of the message arriving beforehand. Only when a dynamic message arrives is it possible to know how much memory to allocate in the corresponding RTC message queue. So dynamic messages require dynamic memory allocation and restoration every time a message is received and read out of the RTC queue, which will cut into the transmission time of such messages. Therefore, dynamic messages would be principally intended for non-real-time systems.

4.2.3 Header Message Consolidation

As mentioned in Section 2.7, a separate header message is sent between processes before the actual message is sent. Although small, this lowers communication throughput. If the header message were prepended to the actual message, throughput would be increased. However, the act of prepending the header message is also costly, mainly because both the header message and the actual message would both need to be copied into one data buffer. If the size of the actual message is large, this operation could be costly, also affecting communication throughput.

The ideal solution would allow the header message to be prepended to the actual message without any copying. That is, the optimal solution would entail the low-level IPC code having the ability to accept merely two pointers to memory where the header and actual data can be found. This would eliminate the extra copy that would otherwise need to occur. Many operating systems support this functionality, but some do not, which is why RTC was originally designed to send a separate header message.

In retrospect, the penalty of an extra copy is probably less than the penalty of having an extra message. There are of course cases where the reverse would be true, but those cases are probably fewer. Therefore, future revisions of RTC would consolidate the header and actual message into one message. For some operating systems, this consolidation would be a good throughput improvement, whereas for other operating systems, it would be a minor improvement, if at all. The choice to do this consolidation would require an architectural analysis to determine if improvement would be seen on the majority of the operating systems.

Appendix

References

[birrell78] Birrell, A. and Nelson, B., “Implementing Remote Procedure Calls”. ACM Transactions on Computer Systems 2, 1978.

[corba] CORBA is maintained by the Object Management Group. *http://www.omg.org*

[fong95] Fong, T., Pangels, H., Wettergreen, D., Nygren, E., Hine, B., Hontalas, P., Fedor, C., “Operator Interfaces and Network-Based Participation for Dante II”. SAE 25th International Conference on Environmental Systems, San Diego, CA, July 1995.

[gowdy96] Gowdy, J., “IPT: an object-oriented toolkit for interprocess communication”. Technical Report CMU-RI-TR-96-07, Carnegie Mellon University, 1996.

[hoffman96] Hoffman, R., Fitzpatrick, K., Ollis, M., Pangels, H., Pilarski, T., Stentz, A., “Demeter: An Autonomous Alfalfa Harvesting System”. ASAE Annual International Meeting, Paper No. 963005, July 14-18, 1996.

[leger98] Leger, C. and Bares, J., “Automated Synthesis and Optimization of Robot Configurations”. To appear in the proceedings of the 1998 ASME Design Engineering Technical Conferences.

[ndds] NDDS is maintained by Real-Time Innovations, Inc. *http://www.rti.com*

[nml] NML is maintained at the National Institute of Standards and Technology, Intelligent Systems Division of the Manufacturing Engineering Laboratory. *http://www.isd.cme.nist.gov/proj/rcs_lib*

[pvm] PVM is maintained at the Oak Ridge National Laboratory, Computer Science and Mathematics Division. *http://www.epm.ornl.gov/pvm*

[rtc] RTC is maintained at the National Robotics Engineering Consortium. *http://cronos.rec.ri.cmu.edu/technology/rtc*

[simmons94] Simmons, R., “Task-Level Control for Autonomous Robots”. Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS ‘94). Houston, TX, Mar 21, 22 1994.

[tca] TCA is maintained at Carnegie Mellon University. *http://www.cs.cmu.edu/afs/cs/project/TCA/release/tca.html*

[stentz98] Stentz, A., Bares, J., Singh, S., and Rowe, P., “A Robotic Excavator for Autonomous Truck Loading”. Submitted to IEEE/RSJ International Conference on Intelligent Robotic Systems, 1998.