Real-Time Interprocess Communication in the Spring Kernel

Erich Nahum, Krithi Ramamritham, and John A. Stankovic

May 1992

Abstract

This document describes the design and ongoing implementation of real-time interprocess communication (IPC) in the Spring Kernel. Spring IPC is designed to offer *predictable* communication. Thus, the communication primitives have bounded execution time and synchronous communication does not introduce unpredictable blocking. We achieve this by tight integration of IPC with Spring scheduling and through compile-time support that maps processes to schedulable tasks.

We discuss our design decisions, explain our programming interfaces, and also give the pseudocode for the IPC primitives. We have designed the subsystem to facilitate analysis of programs for scheduling. We outline the interaction of the IPC subsystem, the SGS, the scheduler, and memory management.

Contents

1 Introduction **4**

2 Design of Spring IPC **6**

2.1 Rationale & Driving Considerations 6

2.2 Ports 6

2.3 Messages 9

2.4 Network Connections 10

2.5 Protection and Name Services 10

3 Communication Semantics in Spring **12**

3.1 Non Real-Time Definitions 12

3.2 Available Programming Abstractions 12

3.3 Mapping Processes to Tasks 13

3.3.1 Mapping Processes that use Synchronous Communication 14

3.3.2 Mapping Processes that use Asynchronous Communication 15

3.4 Restrictions on Synchronous Communication 16

4 Distributed Communication **17**

4.1 The Communication Processor 17

4.2 A Proposed Network Layered Protocol 17

4.3 The Medium Access Protocol 18

4.4 The Transport Layer 19

4.5 The Kernel-to-Kernel Protocol 20

4.6 Network Buffer Management 20

5 Impacts of Communication on other Spring Subsystems **21**

5.1 Scheduling 21

5.2 The Software Generation System (SGS) 21

5.3 Memory Management 22

6 Communication Primitives **23**

6.1 Asynchronous Communication Primitives 23

6.1.1 Asynchronous Send 23

6.1.2 Asynchronous Receive 24
6.2 Synchronous Communication Primitives 25
6.2.1 Synchronous Send 25
6.2.2 Synchronous Receive 26
6.3 Request-Accept-Reply Primitives 27
6.3.1 Request 27
6.3.2 Accept 29
6.3.3 Reply 30
6.4 Port Management 31
6.4.1 Port Creation 31
6.4.2 Port Deletion 33
6.5 Bandwidth Management 33
6.5.1 RTVC Creation 33
6.5.2 RTVC Deletion 34
6.5.3 RTDG Creation 35
6.5.4 RTDG Deletion 35

7 Implementation, Status, and Future Work **37**

8 Acknowledgments **38**

1 Introduction

Communication has been recognized as a central component to any operating system [3] [7]. This paper presents an interprocess communication (IPC) [1] system for the Spring kernel [20]. In conventional systems, IPC can be unpredictible due to the potentially unlimited blocking time of applications synchronizing or waiting for messages. Our IPC subsystem is unique in that it presents a conventional IPC programming interface yet provides predictable and bounded synchronous communication in a hard real-time environment.

Spring is a Distributed Real-Time Operating System that supports the requirement that programs not only produce correct results, but that they produces them *on time*. Intrinsic in Spring is the notion of a *guarantee* that assures completion of hard real-time tasks within deadlines. A necessary (although not sufficient) condition for this is to make all operating system primitives predictable, namely bounded in their execution time.

The Spring environment is composed of multiprocessor nodes, each containing shared memory, and connected via a local area network. A multiprocessor node is composed of several application processors that run user programs, and one system processor that handles job scheduling. Other components of the kernel are present on both the application processors and the system processor. Programs under Spring are known as *tasks*. Each task has an associated value that it adds to the system as a whole, depending on its timing properties, known as its *task type*. Task types are defined as follows:

- Hard Real-Time: These are tasks that have hard deadlines; i.e. after their deadline, running them adds no value to the system. These are further broken up into:
  - Critical: These must complete by the specified time, or a catastrophic result will occur. More specifically, the value attached to the job becomes an infinitely negative number if the deadline is missed. Critical tasks are guaranteed statically by preallocating resources for them and prescheduling their execution.
  - Essential: These tasks have no value after their deadlines, but failure to meet them does not result in a disaster. Essential tasks are guaranteed as and when they arrive using a dynamic guarantee algorithm [14].
- Soft Real-Time: Deadlines are associated with these tasks, but a task's value decays gradually after the deadline. These tasks are scheduled in Spring on a best-efforts basis during times not allocated to hard real-time tasks.
- Non-Real-Time: These tasks have no time requirements, i.e. their value is independent of time. These tasks are executed when idle times occur.

The IPC subsystem provides predictable primitives that have bounded execution time. It is integrated with Spring scheduling policy [14], the Spring compiler [11], and memory management.

[1] Although Spring defines an executable entity as a *task*, we retain the acronym IPC for historical purposes.

The rest of this document is organized as follows. First, we present the design of the IPC system. Then, we define the semantics of communication. We then discuss the interaction between the IPC subsystem and other Spring kernel components. We present the IPC primitives and their respective pseudocode. We then compare Spring IPC with communication mechanisms in other operating systems. We end with our conclusions and summary.

2 Design of Spring IPC

2.1 Rationale & Driving Considerations

The goal of IPC in the Spring Kernel is to provide an efficient and predictable communication system that allows programs to exchange information. Tasks may communicate with other tasks running on the same processor, present on the same node but executing on a different processor, or running on different nodes. The IPC primitives are available to hard, soft, and non real-time tasks.

The subsystem is designed to facilitate analysis of programs so that their schedulability with respect to timing and resource requirements can be determined. Thus, a major consideration in the design has been ease of discerning resource requirements needed for communication, and providing this information to the scheduler, in order to schedule the comunicating entities.

In designing the semantics of the IPC mechanism, we have considered the end-to-end argument by Saltzer et al [16]. System design needs to carefully weigh the utility of a feature against the cost of providing that feature, the likelihood that it will be used, and whether the feature must be provided anyway at a higher level. We argue that programs with hard real-time deadlines will need to perform checks at the application level in order to ensure correctness.

This is also consistent with the idea that in a real-time system, a receiver is in the best position to handle errors and instigate exception handling [8]. We therefore require the application to handle exceptions at the programming level. Applications will be notified of these exceptions through error codes returned by the system calls.

We also realize that utilization cannot be a primary goal, due to the inherent worst-case assumptions that must be made about real-time tasks in order to guarantee predictability.

2.2 Ports

Ports are kernel-protected memory objects (and their associated control information) that hold units of data called *messages*. Processes can communicate with each other by placing messages into ports (henceforth called *queueing*) and removing messages from ports (henceforth called *dequeueing*).

The port abstraction provides unidirectional, indirect communication to the *owner* of the port. Only the owner may remove messages from the port by performing *receives*.

In the distributed case, a *link* must be established between the sender and receiver, in order for two processes to communicate. The link is the network bandwidth required to *send* to the port. Thus, it is the sending side that must provide the link. Links are described in more detail later in Section 2.4.

Ports have a *bounded capacity* that quantifies their ability to store messages. The capacity of the port must be explicitly specified by the owner of the port.

Ports have a *queueing policy* that determines in what order messages are stored and retrieved.

The programmer can specify one of a limited number of policies, with the default discipline being dequeueing the message with the *minimum laxity first*. Laxity is defined as the time remaining before the deadline of the message.

Ports have an *overflow policy* that is taken when a message arrives at a port that has no storage available to queue the message. This is also a programmer-specified option, with the default of discarding the arriving message.

Ports are *typed* in two ways, both based on the kind of communication employed. The components of the type are:

1. *Task Type.* This mirrors the types available to tasks: critical, essential, soft real-time, or non real-time. Tasks may own ports of types up to and including their native task type, reflecting the innate hierarchy of Spring tasks. For instance, a critical task may own a critical port and a soft real-time port, but a soft real-time task cannot own a critical port. This is in order to prevent non-guaranteed tasks from affecting guaranteed tasks timing properties.

2. *Semantic Type.* This describes how messages in this port are to be used. These contexts are asynchronous, synchronous, request, and reply, and will be explained in more detail later.

Ports are *resources* that are composed of control information and message buffers. Ports are owned over the lifetime of the *process*, rather than the task. Processes acquire their ports upon starting and relinquish them when they finish.

Some might ask:

1. *Why unidirectional?* By making ports unidirectional, resource requirements are simpler. Each process needs only to specify the port memory required for receiving. In order to make ports manageable and predictable, it seems wise to place the port buffer memory on the same node as the receiver; that way, all receives can presume the port is local to the node, and all sends are location independent. However, this does not completely cover the memory resource requirements. A sender may need to have buffer space set aside for it in the kernel to store a message after it has been sent, but before it has been shipped over the network. Note that this is only required for processes that send to remote ports in the distributed case.

   Bidirectional ports complicate the resource requirements. If a port represents a 2-way connection between two nodes, where is the buffer memory? What is the ratio of sending bandwidth versus receiving bandwidth? Having each end specify its receiving requirements just seems simpler.

   The unidirectional abstraction is also more powerful and less restrictive. Bidirectional ports limit us to 1:1 semantics. 1:1 semantics may be the initial case with guaranteed tasks, but we would like to leave open in the future the possibility of N:1 ports, at least for non-guaranteed tasks. N:1 semantics can obviously also be built from bidirectional ports, but it is clumsier and perhaps more wasteful.

Bidirectional ports assume a two-way flow of information, which might not necessarily be the right model for real-time communication. A simple example is a sensor that sends data periodically to a controller; the sensor might not be interested in getting messages from that controller. More complicated communication patterns can be constructed from unidirectional semantics if required by the programmer.

2. *Why not communicate directly with a task?* This is essentially for abstraction and fault-tolerance purposes. A port can represent an available service that senders can request without knowing the server task. If the server dies, another process might take its place without clients knowing.

3. *Why not have the receiver handle bandwidth?* Receiving is a passive activity; it is the sending side which will need to gain access to the network media. The sender will need to acquire bandwidth in order to send, in the form of real-time virtual circuits (RTVC's) in the case of hard real-time programs, or real-time datagrams (RTDG's) for soft and non real-time programs. If the receiver supplies the bandwidth, the sender will need to know the receiver's bandwidth, either statically or through some lower-level negotiation, which seems too complicated. However, in the case of guaranteed tasks, the receiver will need to know *when* to expect the message.

4. *Why only one receiver?* This might not stand as a requirement; however, for the moment, it seems reasonable and appears to make guaranteeing hard real-time tasks easier.

5. *Why a minimum laxity queueing discipline?* Arvind has mentioned that this discipline has been shown to be optimal. Other queueing disciplines are FIFO, LIFO, and priority-based. This is obviously a scheduling issue, and needs to be investigated.

6. *Why discard the most recent message in overflow?* This is just for simplicity. An alternative example is to discard the *oldest* message.

7. *Why typed ports?* This is to prevent non-guaranteed tasks from affecting the timing properties of guaranteed tasks. We claim that if a non real-time task could send in an unpredictable manner to a port belonging to a hard real-time task, the port could overflow and thus affect the behavior of the hard real-time task. We require different port types to separate the stream of messages.

   The semantic information is necessary because, as we will describe later, synchronous communication affects scheduling decisions made for guaranteed tasks.

8. *Why do port resources span task boundaries?* Ports must be available over the lifetime of the process for several reasons. First, ports can hold several messages at once, each of which might be used by different tasks in the same process group. Secondly, messages may arrive early before the receiving task executes. Associating the port exclusively with the task means that an arriving message might have nowhere to go. In addition, binding ports only to tasks may further complicate Spring scheduling.

2.3 Messages

Messages are units of information that are passed between processes via the *send* and *receive* calls. Messages have fixed sizes, and strict copy-by-value semantics. Messages can have *deadlines* that determine when they must be delivered to a port. Messages are also *typed* in the same way that ports are, by both real-time and semantic content. The type is determined by the port the message is sent to.

Some might ask:

1. *Why not variable-sized messages?* Again, for resource requirement purposes, fixed size makes the most sense. The question then becomes, how many sizes and how big are they? This question is still open to debate, but we argue that there should be one or two sizes. Most messages in conventional operating systems are small [3]. Since network packets are limited by an upper bound, we argue the maximum message size should be the maximum packet length. Having a message completely contained within a packet allows a simple transport layer and no complications about incomplete messages from lost packets. FDDI [17] provides packets of up to 4K, so this is not as bad as it sounds.

2. *Why not copy-by-reference?* Although copy-by-reference saves us copying time, it presents several problems:

   - In the remote case, copying *must* be performed anyway. We could avoid copying if we had a memory-management scheme such as copy-on-write [7], but Spring does not support copy-on-write yet. Copy-by-reference would give different semantics between the local and remote cases (unless Spring supported some form of distributed shared virtual memory), and we would like the IPC to be as network-transparent as possible. Again, since shared memory is available to the programmer for local communication, we feel comfortable about the copying costs.
   - It makes resource requirements unclear, as the memory required for a port becomes part of the resource requirements for the sender. Port overflow becomes difficult to detect. This gets even more complicated in the distributed case.
   - It isn't as clean an abstraction as copy-by-value. It places a burden on the application programmer, who must exercise discipline in not using buffers that have been sent. If the buffers are treated as use-once objects ("use it and kiss it goodbye"), it will require programmers to explicitly acquire all the buffers needed up front, as opposed to implicitly reserving only the number of buffers needed at any single instance in time. It is also not symmetric in that the programmer has one way to acquire memory, but two ways to dispose of it.
   - It may not work due to architectural considerations. If memory is not *truly* dual-ported (as is the case with the current system), copy-by-reference might slow down another application's memory references. Consider the example where a task wants to receive from a port buffer that is present on a different board. The receiver may 'victimize' another task because the memory on that board cannot service two

requests simultaneously. This problem can be avoided by copying the buffer to a third place outside of either processor, where contention is assumed. This problem is dependent on the outcome of our architectural discussions.

3. *Why typed messages?* This is a result of our decision to type ports, in order to prevent asymmetric communication that could disrupt the timing properties of guaranteed tasks.

4. *Why not type them by the sender?* It is the receiver that is affected by the arrival of the message. The receiver needs to act based on what kind of message it is that arrives.

2.4 Network Connections

We have identified two types of communication abstractions [1] to cater to the needs of the system.

Real-time virtual circuits (RTVC's) are dedicated channels that guarantee network access within a bounded amount of time. RTVC's are generally used by hard real-time tasks, and are allocated appropriately. Critical tasks have their RTVC's pre-allocated at system boot time; essential tasks have their RTVC's allocated at guarantee time.

Real-time datagrams (RTDG's) are communication channels that provide network access on a *best efforts* basis. RTDG's attempt to deliver a message within its deadlines, but no guarantee is made. RTDG's are used by soft and non real-time tasks, and are allocated at run time.

The communication processor distinugishes between these classes of messages by maintaining different transmission queues for each class. RTVC's are given priority over RTDG's to be sent over the network media. Similarly, two classes of receive buffers are mantained for arriving packets. Guaranteed tasks have an explicit number of buffers reserved for them at the time of guarantee. Upon receipt of a packet, an interrupt is invoked and the packet is received into a buffer. The buffer is examined and queued into the appropriate receive list.

Applications specify their connection requirements as they do other resources. They request connections via the provided IPC system calls, which allocate, free, and query network connections. Connection information is available to all processors on the node, and is kept in shared memory on the communication processor, described in Section 4.1.

2.5 Protection and Name Services

Our design supports the notion of *capabilities* associated with each port that are possessed by tasks. Ideally, whenever a task performs a send or receive, we would like to check that the task is authorized to perform that operation. Providing protection from errant tasks helps ensure the correctness of our system. Protection checks, if capabilities are kept locally, can be performed quickly with no loss of predictability. If a production system desires optimum performance, protection could be enabled during system development and testing, and then disabled in the field.

Being able to enforce remote protection locally, however, requires that each node maintain all capability information about all ports. This scenario is likely not feasible. However, in the case of hard real-time tasks, each node should have the capability information about every hard real-time port that the node could potentially communicate with. Thus, every send to a hard real-time port could be checked locally. This would be quite fast using perfect hash tables. It also improves predictability in that it avoids broadcasting. Allowing hard real-time tasks to be interrupted by broadcasting would make make predictability difficult, if not impossible, especially if non real-time tasks were allowed to broadcast to critical tasks.

Capabilities for soft and non real-time ports would be checked only at the destination node. Each node would maintain its own capability lists and enforce them at the receiving end. Distributing the capabilities this way would improve fault tolerance, since if a node went down, so would the capability information that was relevant to it.

The protection and naming issues are not yet resolved. It is clear that the kernel needs to be able to perform name resolution. Tasks need to register port names with the kernel; statically in the case of hard real-time tasks, dynamically with soft and non real-time tasks. Problems such as accommodating heterogeneity or responding to growth of the domain name space are generally not issues with a real-time distributed system.

3 Communication Semantics in Spring

Our next step is to define what communication forms are available in Spring and what synchronous and asynchronous communication *mean*.

3.1 Non Real-Time Definitions

In conventional distributed operating systems literature, the following definitions in communication hold sway:

1. *Synchronous Communication.* The two agents involved in communicating wait for each other and synchronize at some agreed point in their execution path.
2. *Asynchronous Communication.* The two agents involved in communication do *not* synchronize. The sender does not wait for the receiver, and vice-versa. However, within asynchronous communication are two sub-types:
   - *Blocking Asynchronous Communication.* In this case, processes block until an action can be taken with the queue that acts as an intermediary. Sends will block until a free buffer is available in the queue; receives will block until a buffer with a message in it is available in the queue.
   - *Non-Blocking Asynchronous Communication.* Here, processes do not block. A sender does not wait to see if the message can be queued at the destination port; a receiver does not wait if no messages are available.

We wish to provide what is known as *synchronous* and *asynchronous non-blocking* communication for both sends and receives. Henceforth when we mention asynchronous communication, we will imply the non-blocking form.

3.2 Available Programming Abstractions

The abstractions we wish to provide to the programmer are the following:

The primitives we provide to the programmer are the following:

1. *Asynchronous Send and Receive.* The sender does not wait for the receiver, and also does not wait to see if the message can be queued at the destination port; a receiver does not wait if no messages are available. Messages sent have a *deadline* that they *should* be delivered by.
2. *Synchronous Send and Receive.* The source process suspends after a send until the destination process performs a receive on the port and dequeues the message. The message sent has a *deadline* that it must be delivered by. The sender gets no explicit response with any from the receiver with any meaningful data. Instead, the sender gets an *implicit*

response that signifies that the receiver has received the message. This implicit response is supplied by the scheduler in the case of guaranteed tasks, and by an event-driven kernel-to-kernel acknowledgement in the case of non-guaranteed tasks. The semantics are slightly different between the guaranteed and non-guaranteed cases. The continuation of control flow beyond the primitives assures the sender that the message has been received in the case of non-guaranteed tasks. Receipt can *not* be inferred with guaranteed tasks.

3. *Request-Response.* These define the conventional synchronous client-server model. A request is made by a client, who must supply a reply port. The client blocks until the server accepts the request, acts upon it, and returns a meaningful response to the reply port.

The sender of a message specifies a deadline – this is the time by which the IPC system should deliver the message. The delivery is guaranteed to arrive on time for synchronous communication involving hard real-time tasks. For asynchronous communication the system does its best to deliver the *message* on time. However, note that the deadlines of the individual hard real-time tasks are still guaranteed.

The supplied abstractions are unusual in that normally both the synchronous and request-response primitives are not offered together in one system. We call both sets of abstractions *synchronous constructs*. Synchronous constructs in hard real-time tasks require recognition by the compiler and support in the scheduler. Request-accept-reply is a higher-level abstraction that can typically be constructed using the synchronous abstractions. We supply both because more flexibility is provided and, most importantly, they use the same set of internal primitives. Thus our development cost for providing the extra abstractions is minimal.

3.3 Mapping Processes to Tasks

Synchronous communication conventionally implies that a task can be blocked for an indefinite period. For hard real-time tasks, unpredictable blocking can obviously not be allowed. To overcome this, we take advantage of the semantics of communication and the Spring system paradigm, mapping *processes* to *tasks*. Processes are the conventional process model abstraction seen by the programmer. Tasks are the units of execution that are scheduled and run by the Spring system. Synchronous communication in hard real-time tasks require recognition by the compiler and support in the scheduler.

A group of programs that cooperate and communicate among themselves form a *process group* [11]. Each process is be decomposed into tasks that have precedence constraints attached to them. The precedence constraints allow the scheduler to construct a schedule that, if feasible, will guarantee that the entire process group will complete on time. In the distributed case, before any task of a process group is run, the local scheduler knows the execution plan of each task in the entire process group and their allocated communication channels.

Thus at run-time, for example, if a guaranteed task performs a synchronous receive, the scheduler knows *when* to schedule that task such that the message is *guaranteed* to be present.

Thus, suspending a process defines a boundary between two tasks. The duration of the suspension time is determined by the scheduler when the process group as a whole is guaranteed.

In addition to compiling code, the Spring Compiler Environment, called The *Software Generation System* (SGS), identifies the correspondence between matching synchronous primitives within a process group. Information is passed between the SGS and the scheduler so that the scheduler can identify the precedence constraints between tasks.

3.3.1 Mapping Processes that use Synchronous Communication

To illustrate process mapping, we give the following example:

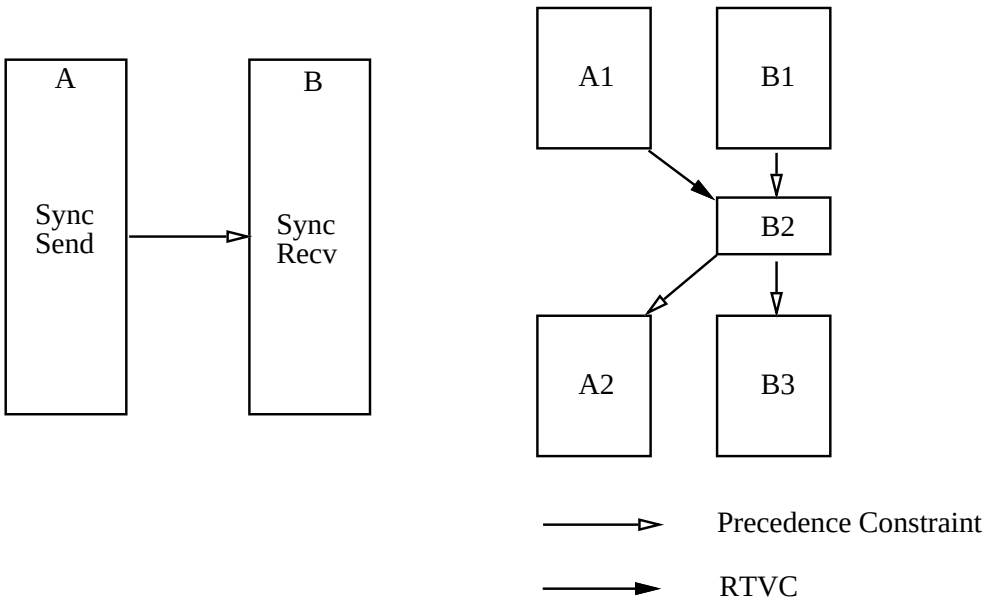

Figure 1. Mapping a Synchronous Send.

A real-time systems designer has written programs A and B with hard deadlines that wish to synchronously communicate. The processes A and B are written using the process model. Upon compilation, however, the processes are decomposed into the set of tasks $A1$, $A2$, $B1$, $B2$, and $B3$. A set of precedence constraints are generated that determine the running order of the group of tasks. In order to preserve the semantics of communication, $B2$ may not run before $A1$ and $B1$, and $A2$ and $B3$ must run after $B2$. No other constraints are made, i.e. $B3$ may run before, after, or concurrently with $A2$.

When the process group as a whole is scheduled to run, the scheduler follows these precedence constraints when constructing an execution plan. Each task is assigned an execution time such that the process group meets its original deadline. For example, in the set of tasks $B1$, $B2$ and $B3$, each task has a scheduled execution time such that the *task* $B3$ finishes by the original deadline of the *process* B.

In addition, since process A is actually sending information to B, a communication channel with predictable performance is allocated to A. The channel has a worst-case *guaranteed* propagation time to send a message, and this time is used to determine the minimum time that must occur after $A1$ finishes but before $B2$ begins. A deadline is attached to the message from $A1$ to indicate it must arrive by $B2$'s start time. Note that no *explicit* information is sent from $B2$ to $A2$; the semantics of the precedence constraint are preserved by the order of execution of the two tasks.

3.3.2 Mapping Processes that use Asynchronous Communication

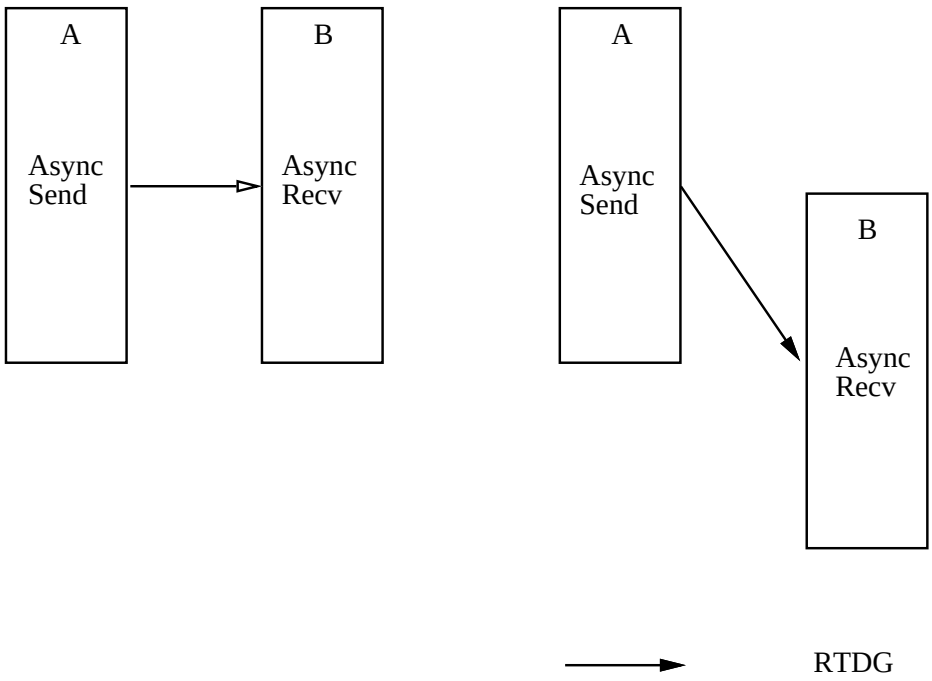

Figure 2. Mapping an Asynchronous Send.

In this example, A and B are again processes with hard real-time deadlines, but the semantic primitives used are for *asynchronous* communication. This means that process B does not *require* the message from process A to be delivered on time. The deadline for the message is the point where process B performs the receive. A best-efforts attempt is made to deliver the message on time, through an appropriate channel; in this case, a real-time datagram. A and

B are not decomposed any further, that is, the task boundaries are identical to the process boundaries. The scheduler guarantees that A and B will make their respective deadlines, and attempts to execute the tasks such that the message appears at the appropriate time for B. However, the message deadline is not a firm requirement as is the case with synchronous communication.

3.4 Restrictions on Synchronous Communication

Since synchronous communication affects scheduling decisions made for guaranteed tasks, we must maintain some restrictions on use of the synchronous IPC primitives. For example, we cannot allow a non-guaranteed task to communicate synchronously with a guaranteed task, as it would make the guaranteed task unpredictable. We cannot even allow a critical hard real-time task to synchronously communicate with an essential task. Because essential tasks are guaranteed dynamically, there is no *a priori* guarantee that any essential task will *ever* execute.

Thus, we must restrict synchronous communication in hard real-time tasks such that only tasks of the same type can synchronously communicate. In other words, critical tasks can only use synchronous communication with other critical tasks. Essential tasks may communicate synchronously only with other essential tasks. We do not need to restrict synchronous communication between soft and non real-time tasks, as they do not affect scheduling decisions. The following table summarizes the restrictions on synchronous communication between two tasks.

Sending Task	Receiving Task			
	Critical	Essential	Soft-RT	Non-RT
Critical	Yes	No	No	No
Essential	No	Yes	No	No
Soft-RT	No	No	Yes	Yes
Non-RT	No	No	Yes	Yes

Figure 1. Synchronous Communication Permitted

In order for the scheduler to construct execution plans, the SGS must know that every synchronous IPC primitive must be used symmetrically. The IPC subsystem enforces the symmetric use of these abstractions, e.g., we forbid a *synchronous send* being met by an *asynchronous receive*. For this reason, we type the ports (and the messages that are sent to them) by their semantic type as being asynchronous, synchronous, request, or reply, in addition to their timing properties. Typing messages and ports by semantic content also enforces good programming style and improves ease of debugging.

4 Distributed Communication

As Spring IPC is designed with network communication in mind, we have carefully thought out the integration between the IPC subsystem and the network hardware and software. Since Spring runs on a local area network, we can simplify the network interface considerably. We use the RTLAN description presented in [1]. In particular, our protocol model only requires four layers, as opposed to ISO's more conventional seven. The network architecture can be seen in Figure 1.

4.1 The Communication Processor

The Spring architecture includes a dedicated board that handles network communication. On this communication board is a dedicated network microprocessor with dual-ported memory. The network processor communicates with an on-board MAC chip, and has software that handles the logical link control. The board also has components of the kernel IPC code required for interacting with other boards on a node. The IPC subsystem communicates to the board via the communication processor's on board memory. Other processors are allowed to read and write to the dual-ported memory, but the network CPU cannot access memory on the application processors. Since the application memory is not dual-ported, allowing accesses from agents other than the host CPU would slow down the applications running on that board, an event we call 'victimization'.

Physically separating the communication processor from the rest of the Spring node modularizes the design and facilitates analyzing the board to guarantee the timing properties of network communication. It shields the rest of the architecture from the asynchronous event of an arriving packet. It also allows us to experiment with different real-time network protocols. Other work has used dedicated communication processors for performance [2] [5]; our intent is to guarantee predictability.

4.2 A Proposed Network Layered Protocol

We describe a hypothetical protocol layer that can be used for predictable network communication. Following Saltzer's argument [16], each network layer service needs to be evaluated to see if the benefits provided are worth the cost to both the programmer and to the scheduler's complexity.

- *Packet Corruption.* Although Saltzer gives an example of how an application needs to perform its own checksums to guarantee correctness, it seems clear and simple enough that the MAC layer should check bad packets and discard them. This is typically performed in hardware anyway.
- *Packet Loss, Acknowledgements, and Retransmissions.* The end-to-end argument shows that, in order to ensure correctness, end-to-end acknowledgements must be used. Given high reliability of future network media, we argue that this will not be a large problem,

that the network protocol should not deal with it, and must be handled by the application. Application programmers will be responsible for timeouts and retransmissions.

- *Out-of-Order Delivery.* Given that Spring runs on a LAN, the network medium will not provide storage capacity and thereby introduce duplicates. Therefore, the IPC system will not guard against out-of-order messages. If applications send multiple messages, they will have to protect themselves from duplicates. In this case, if duplicate messages are introduced at some lower level, it seems clear that the same layer should be responsible for removing them.

- *Deadline Violation.* Messages may arrive after their deadlines. This has different consequences for different types of tasks. One consideration is that we would like to aid the real-time programmer in finding timing violations (as in the ARTS kernel [22]). A message missing a deadline for a hard real-time task could be handled in several different ways, such as posting an event to the kernel, or passing a flag to the owning process the next time it performs a receive.

Our architecture attempts a minimum amount of copying in the remote case. To do this, our layers are somewhat tightly coupled, although we try to keep each layer's knowledge of one another limited. Our idea is to pass buffers via pointers through the layers, thus eliminating copying. When a packet is received from the network, the buffer it is received into is passed up. Each layer examines the same buffer, but restricts its' knowledge to an offset into the buffer where header information for that layer is kept. Hence the restriction that a user message must be some subset of the maximum packet size.

Our argument for this is that for next generation real-time systems, speed and predictability will be the primary requirements, and that memory and network bandwidth will be cheap. Fiber optic technology presents network bandwidth on the order of gigabits per second. Current philosophy in high speed networks argues for 'lightweight' protocols, as the performance bottleneck is shifting from the medium to the software. Similarly, memory densities are continually increasing, and 256 megabit chips are predicted by the end of the decade. Our design is basically a space for time tradeoff. We are willing to trade wasted space in the form of internal fragmentation of message buffers for the time gained with avoiding copying.

4.3 The Medium Access Protocol

The MAC layer in the network is the device interface that gains physical access to the medium. Although we do not specify a MAC protocol, we do make the requirement that the MAC layer support both dedicated (e.g. real-time virtual circuits) and contention-based bandwidth (e.g. real-time datagrams) that can be acquired in a bounded amount of time [1]. This is to provide guaranteed bandwidth to guaranteed hard real-time tasks. Several protocols support this, such as FDDI II, TDMA, and INTP2 [1]. MAC protocols are typically implemented in hardware.

The MAC layer is what determines the maximum packet size. Other layers are affected by this in that they need to know both the maximum packet size and the offset into a packet where

they can place their framing information. The physical packet sent out on the net might have the following frame information:

- Source Address
- Destination Address
- Packet Size
- Cyclic Redundancy Check

4.4 The Transport Layer

The transport layer in the IPC subsystem provides reliable, error-free transmission of packets through the MAC layer. Its duty is to shield the kernel from the unreliable aspect of the network. Since the underlying network is presumed to be a LAN, there can be no out-of-order packet delivery or storage capacity that produces 'ghost' messages. Therefore, the transport layer needs only manage packet loss, retransmission, and duplicate suppression.

Since the message size is limited to within the maximum packet size, the transport need not break up messages into TPDUs. Our transport service is therefore inherently connectionless. In our initial design, we burden the programmer with the task of breaking up large amounts of data into packets.

Our transport layer is similar to a data link control layer in that it must maintain sequencing information and prepare outgoing packets for a MAC chip. When a message is ready to be sent, the transport constructs a packet with both MAC-level and transport level information, and places the packet in a queue of requests for transmission. This level must be able to distinguish between classes of traffic and place packets in the appropriate queues, namely the dedicated and contention-based send queues.

The transport layer provides delivery within a fixed amount of time. The *Byzantine Generals Problem* shows that there is no way to completely guarantee agreement between two nodes that a message has been delivered. Since there always exists the probability of packet loss, there is always a possibility, however small, that a packet can *never* get through. However, there are several methods to overcome this problem: through improved media reliability (e.g. fiber-optics), through *spatial redundancy* (i.e. physically replicated links), and through *temporal redundancy* (i.e. retransmissions). Each of these methods can be used to reduce the probability of message loss. This is essentially a fault-tolerance problem. We must, however, bear timing considerations in mind, in order to bound packet service time for hard real-time tasks. For example, we would need to bound the number of retransmissions. In any event, the transport layer can be made to have the same or higher reliability as the CPU machine hardware through these methods. Thus, the transport layer is a modular entity that can be replaced and improved as necessary.

The following items need to be placed in the transport layer frame header:

- Sequence Number (on a node-by-node basis for identifying duplicates)

- Packet Type (DATA or ACK)

4.5 The Kernel-to-Kernel Protocol

The kernel-to-kernel protocol is responsible for queueing messages at ports. To do this, it needs to manage port buffering and overflow, enforce protection, perform name resolution, distinguish between messages destined for local and remote sites, and if necessary, transmit messages to the appropriate node. The protocol also provides reply mechanisms for non-guaranteed synchronous sends that go over the network. When a message arrives from a remote node that has been marked synchronous, the kernel attempts to place that message in the local port. The kernel then sends back a *reply* that indicates the success or reason for failure of the attempt.

The kernel to kernel protocol requires the following data to be placed in its frame header:

- Transaction ID (to uniquely identify a task/port interaction)
- Message Type (SEND or REPLY to a send)
- Synchronous/Asynchronous Indication (whether a reply is required)
- Task ID (the source of a SEND, the destination for a REPLY)
- Port ID (the destination for a SEND, the source of a REPLY)

4.6 Network Buffer Management

Generally, for speed purposes, the IPC system avoids calling the memory management routines to acquire or release port buffers. Instead, the subsystem maintains its own free buffer pool that is shared by all ports. As shown in Figure 2, the IPC system is composed of circulating buffers that are passed between layers via pointers. In order to guarantee predictability, it is necessary for the kernel to monitor the availability of network buffers to ensure that an appropriate amount of buffers is available for hard real-time tasks. Since ports are memory resources, applications will be required to specify the amount of port buffers needed. The situation can occur then, where a soft real-time packet is received into a buffer but must be dropped, because to accept that packet would cause the available buffers to fall below a certain threshold necessary for hard real-time tasks.

5 Impacts of Communication on other Spring Subsystems

Synchronous communication conventionally implies that a task can be blocked for an indefinite period. For hard real-time tasks, unpredictable blocking can obviously not be allowed.

5.1 Scheduling

A group of programs that communicate among themselves will form a *process group* [11]. Each process will be decomposed into tasks that have precedence constraints attached to them. The precedence constraints will allow the scheduler to construct a schedule that, if feasible, will guarantee that the entire process group will complete on time. In the distributed case, before any task of a process group is run, the local scheduler will know the execution plan of each task in the entire process group.

Thus at run-time, for example, if a guaranteed task performs a synchronous receive, the scheduler will know *when* to schedule that task such that a message *should* be present. In this example, if there are no messages available, it means that either the scheduler has not scheduled the tasks correctly, or that an error has occurred, such as a lost message or failed task. The error code returned to the receiver is interpreted as an *exception* that must be handled by the application.

Henceforth, when we say that a synchronous construct *blocks*, we mean that the *process* is suspended. Tasks in Spring never block. Thus, suspending a process defines a boundary between two tasks. The duration of the suspension time is determined by the scheduler when the process group as a whole is guaranteed.

In the case of soft and non real-time tasks, it is possible (albeit undesirable) for these types of tasks to block indefinitely. The blocking of their synchronous constructs is not planned by the scheduler, and thus is event-driven as in conventional operating systems. In order for the kernel to know when to unblock the primitives, a kernel-to-kernel protocol will be used that supplies an *acknowledgement* for non-guaranteed tasks.

5.2 The Software Generation System (SGS)

A very tricky problem will be how the SGS identifies the correspondence between matching synchronous constructs within a process group. Some form of information will need to be passed between the SGS and the scheduler so that the scheduler can identify the precedence constraints between tasks. The mechanics of how this will be done require much more investigation. Related to this is how and when (compile-time, guarantee-time, or run-time) deadlines are determined for individual messages in synchronous communication. It is clear that the coupling between the SGS and the Spring Kernel will need to be very tight.

The IPC primitives require a *deadline* that is attached to the message and is used by the underlying internals of the IPC subsystem. Ideally, this deadline would be constructed by the SGS as part of the timing analysis of a process group. For the moment, the programmer will

need to explicitly supply it. How the deadline is provided does not affect the semantics of the primitives.

5.3 Memory Management

The IPC subsystem should not cause any major affects on memory management and logical address mapping. However, it will need to take advantage of the MM primitives to copy information between address spaces. The communication board memory is a special memory resource that will have to be specially allocated.

6 Communication Primitives

6.1 Asynchronous Communication Primitives

Asynchronous communication in Spring is similar to asynchronous communication in most other operating systems. The use of these primitives does not involve the scheduler or precedence constraints, regardless of the type of task.

6.1.1 Asynchronous Send

```
ipc_return_t
send_async(port, msg, deadline, type, importance)
port_t          port;
msg_t *         msg;
task_t          type;
deadline_t      deadline;
importance_t    importance;
```

The `send_async()` call takes the passed message and sends it asynchronously to the specified port. The passed message is *typed as asynchronous* and copied to a kernel buffer. The call returns to the application, without waiting for the message to be received at the other end. Note that the sender is *not* necessarily informed whether a message is delivered successfully or not. The pseudocode for this primitive is:

1. *Port Lookup.* The local name server is queried to find the location of the port.
2. *Capability Lookup.* If possible, the sending task is checked against the port capability list to enforce protection.
3. *Message Buffer Lookup.* Depending on where the destination port is, a message buffer is acquired from a free pool:
   - If the port is local to the processor, the buffer is taken directly from the free pool of that port.
   - If the port is on another node, the buffer is taken from the kernel's free pool of network transmission buffers.
4. *Message Copy.* The passed message is copied into the acquired message buffer.
5. *Message Queueing.* Depending on where the destination port is, the message buffer is queued into the appropriate list:
   - If the port is local to the processor, the buffer is queued into the received list of that port.

- If the port is on another node, the buffer is queued into either a guaranteed or non-guaranteed network transmission list, depending on the type of the message.

Possible error codes from `send_async()` are:

```
IPC_ERROR_PORT_UNKNOWN          IPC_ERROR_CAPABILITY_VIOLATION
IPC_ERROR_PORT_OVERFLOW         IPC_ERROR_NETWORK_PORT_OVERFLOW
IPC_ERROR_PORT_TYPE_MISMATCH
```

6.1.2 Asynchronous Receive

```
ipc_return_t
recv_async(port, msg, flags)
port_t          port;
msg_t *         msg;
ipc_flags *     flags;
```

The `recv_async()` call checks the specified port to see if a message is available. If so, the message is copied into the passed message buffer. If not, the call returns immediately without suspending. In addition to the return code, a bit map indicating possible errors is returned in the `flags` parameter. The `recv_async()` call will only accept messages *typed as asynchronous* sent via the `send_async()` call. The pseudocode for this primitive is:

1. *Port Lookup.* The local name server is queried to find the location of the port.
2. *Capability Lookup.* The receiving task is checked against the port capability list to enforce protection.
3. *Message Buffer Lookup.* The port is checked to see if any messages have been queued for receiving.
4. *Message Buffer Dequeueing.* If a message is available, a message is dequeued according to the queueing policy of the port.
5. *Message Copying.* If a message has been dequeued, it is copied to the message buffer provided by the caller.
6. *Buffer Recycling.* After being copied, the used port message buffer is returned to the port free list.

Possible error codes from `recv_async()` are:

```
IPC_ERROR_PORT_UNKNOWN          IPC_ERROR_MESSAGE_UNAVAILABLE
IPC_ERROR_PORT_TYPE_MISMATCH    IPC_ERROR_CAPABILITY_VIOLATION
```

Possible bits set in the `flags` parameter from `recv_async()` are:

```
RECV_MASK_PORT_OVERFLOW         RECV_MASK_DEADLINE_VIOLATION
```

6.2 Synchronous Communication Primitives

In synchronous communication, *scheduling constraints* are used with a hard real-time process to ensure that when such a process performs a receive, a message will be available. Since these primitives rely on the scheduler to unblock them, they are only available to guaranteed tasks.

6.2.1 Synchronous Send

```
ipc_return_t
send_sync(port, msg, deadline, type, importance)
port_t          port;
msg_t *         msg;
task_t          type;
deadline_t      deadline;
importance_t    importance;
```

The `send_sync()` call takes the passed message, *types it as synchronous*, and sends it synchronously to the specified port. The calling application is suspended until the owner of the specified port receives the passed message. Note that returning from the call does *not* guarantee that the message has been received.

The SGS will expect every `send_sync()` call to be matched by a corresponding `recv_sync()` call in another process in the same process group. The SGS is responsible for recognizing the `send_sync()` call as a scheduling point which defines a task boundary.

The pseudocode for this primitive is:

1. *Port Lookup.* The local name server is queried to find the location of the port.
2. *Capability Lookup.* If possible, the sending task is checked against the port capability list to enforce protection.
3. *Message Buffer Lookup.* Depending on where the destination port is, a message buffer is acquired from a free pool:
   - If the port is local to the processor, the buffer is taken directly from the free pool of that port.
   - If the port is on another node, the buffer is taken from the kernel's free pool of network transmission buffers.
4. *Message Copy.* The passed message is copied into the acquired message buffer.
5. *Message Queueing.* Depending on where the destination port is, the message buffer is queued into the appropriate list:
   - If the port is local to the processor, the buffer is queued into the received list of that port.

- If the port is on another node, the buffer is queued into either a guaranteed or non-guaranteed transmission list, depending on the type of the message.

6. *Suspension.* The `process_suspend()` function is called to return control to the dispatcher. The dispatcher will return control to the `send_sync()` primitive in the context of the succeeding task when that task is scheduled to be run.

Possible error codes from `send_sync()` are:

```
IPC_ERROR_PORT_UNKNOWN          IPC_ERROR_CAPABILITY_VIOLATION
IPC_ERROR_PORT_OVERFLOW         IPC_ERROR_NETWORK_PORT_OVERFLOW
IPC_ERROR_PORT_TYPE_MISMATCH
```

6.2.2 Synchronous Receive

```
ipc_return_t
recv_sync(port, msg, flags)
port_t          port;
msg_t *         msg;
ipc_flags *     flags;
```

The `recv_sync()` call suspends until a message is available at the specified port. When a message is received, it is copied into the passed buffer. In addition to the return code, a bit mask indicating possible errors are returned in the `flags` parameter. The `recv_sync()` call will only accept messages *typed as synchronous* sent via the `send_sync()` call.

The SGS will expect every `recv_sync()` call to be matched by a corresponding `send_sync()` call in another process in the same process group. The SGS is responsible for recognizing the `recv_sync()` call as a scheduling point which defines a task boundary.

The pseudocode for this primitive is:

1. *Suspension.* The `process_suspend()` function is called to return control to the dispatcher. The dispatcher will return control to the `recv_sync()` primitive in the context of the succeeding task when that task is scheduled to be run.

2. *Port Lookup.* The local name server is queried to find the location of the port.

3. *Capability Lookup.* The receiving task is checked against the port capability list to enforce protection.

4. *Message Buffer Lookup.* The port is checked to see if any messages have been queued for receiving. If none are, an exception is returned to the application via the error return code

5. *Message Buffer Dequeueing.* If a message is available, a message is dequeued according to the queueing policy of the port.

6. *Message Copying.* If a message has been dequeued, it is copied to the message buffer provided by the caller.

7. *Buffer Recycling.* After being copied, the used port message buffer is returned to the port free list.

Possible error codes from `recv_sync()` are:

```
IPC_ERROR_PORT_UNKNOWN          IPC_EXCEPTION_MESSAGE_UNAVAILABLE
IPC_ERROR_PORT_TYPE_MISMATCH    IPC_ERROR_CAPABILITY_VIOLATION
```

Possible bits set in the `flags` parameter from `recv_sync()` are:

```
RECV_MASK_PORT_OVERFLOW         RECV_MASK_DEADLINE_VIOLATION
```

6.3 Request-Accept-Reply Primitives

Since the `request()`, `accept()`, and `reply()` primitives are higher-level synchronous constructs, scheduling constraints will also be used with them in the case of hard real-time processes to ensure messages will be available when needed.

With non-guaranteed processes, these synchronous constructs will block for unpredictable amounts of time. In the lower levels of the IPC system, messages will be delivered on a best-efforts basis, and will have lower priority than hard real-time tasks. No scheduling points or precedence constraints will be involved.

As the `request()` and `reply()` primitives send messages, programs that use them will be required to supply the bandwidth for those sends. The client making a `request()` will also be required to supply a reply port, and all relevant memory resource requirements that are associated with a port.

6.3.1 Request

```
ipc_return_t
request(request_port, request_msg, reply_port, reply_msg,
        deadline, type, importance)
port_t          request_port;
msg_t *         request_msg;
port_t          reply_port;
msg_t *         reply_msg;
task_t          type;
deadline_t      deadline;
importance_t    importance;
ipc_flags_t *   flags;
```

The `request()` call submits a message *typed as a request* to the specified request port and waits for a message *typed as a reply* to arrive at the specified reply port. Once it has arrived, the reply is copied into the passed reply message buffer. The calling application is suspended until the owner of the specified port receives the passed message (via the `accept()` call), replies to the message (via the `reply()` call) and the reply is received at the specified reply port. In addition to the return code, a bit map indicating possible errors is returned in the `flags` parameter.

For each `request()` call made, the SGS will expect the corresponding `accept()` and `reply()` calls in another process in the same process group. The SGS is responsible for recognizing the `request()` call as a scheduling point which defines a task boundary.

The pseudocode for this primitive is:

1. *Port Lookup.* The local name server is queried to find the locations of the request and reply ports.
2. *Capability Lookup.* If possible, the requesting task is checked against each of the port's capability lists to enforce protection.
3. *Message Buffer Lookup.* Depending on where the request port is, a message buffer is acquired from a free pool:
   - If the port is local to the processor, the buffer is taken directly from the free pool of that port.
   - If the port is on another node, the buffer is taken from the kernel's free pool of network transmission buffers.
4. *Message Copy.* The passed message is copied into the acquired message buffer.
5. *Message Queueing.* Depending on where the destination port is, the message buffer is queued into the appropriate list:
   - If the port is local to the processor, the buffer is queued into the received list of that port.
   - If the port is on another node, the buffer is queued into either a guaranteed or non-guaranteed network transmission list, depending on the type of the message.
6. *Suspension.* The `process_suspend()` function is called to return control to the dispatcher. The dispatcher will return control to the `request()` primitive in the context of the succeeding task when that task is scheduled to be run.
7. *Message Buffer Lookup.* The port is checked to see if the reply message has arrived at the reply port. If it has not, an exception is returned to the application via the error return code.
8. *Message Buffer Dequeueing.* If the reply is available, it is dequeued from the port.

9. *Message Copying.* If a message has been dequeued, it is copied to the reply message buffer provided by the caller.

10. *Buffer Recycling.* After being copied, the used reply port message buffer is returned to the reply port free list.

Possible error codes from `request()` are:

```
IPC_ERROR_PORT_UNKNOWN          IPC_EXCEPTION_REPLY_UNAVAILABLE
IPC_ERROR_PORT_OVERFLOW         IPC_ERROR_NETWORK_PORT_OVERFLOW
IPC_ERROR_PORT_TYPE_MISMATCH    IPC_ERROR_CAPABILITY_VIOLATION
```

Possible bits set in the `flags` parameter from `request()` are:

```
RECV_MASK_PORT_OVERFLOW          RECV_MASK_DEADLINE_VIOLATION
```

6.3.2 Accept

```
ipc_return_t
accept(port, msg, flags)
port_t          port;
msg_t *         msg;
ipc_flags *     flags;
```

The `accept()` call waits for a request to arrive at the specified port. When a message is received, it is copied into the passed buffer. In addition to the return code, a bit mask indicating possible errors are returned in the `flags` parameter.

The `accept()` call will only receive messages *typed as a request* sent via the `request()` call. The SGS will expect every `accept()` call to be followed by a matching `reply()` call. In addition, the SGS will expect a corresponding `request()` call in another process in the same process group. The SGS is responsible for recognizing the `accept()` call as a scheduling point which defines a task boundary.

The pseudocode for this primitive is:

1. *Suspension.* The `process_suspend()` function is called to return control to the dispatcher. The dispatcher will return control to the `accept()` primitive in the context of the succeeding task when that task is scheduled to be run.

2. *Port Lookup.* The local name server is queried to find the location of the port.

3. *Capability Lookup.* The receiving task is checked against the port capability list to enforce protection.

4. *Message Buffer Lookup.* The port is checked to see if any messages have been queued for receiving. If none are, an exception is returned to the application via the error return code.

5. *Message Buffer Dequeueing.* If a request message is available, it is dequeued according to the queueing policy of the port.

6. *Message Copying.* If a message has been dequeued, it is copied to the message buffer provided by the caller.

7. *Buffer Recycling.* After being copied, the used request port message buffer is returned to the request port free list.

Possible error codes from `accept()` are:

```
IPC_ERROR_PORT_UNKNOWN          IPC_EXCEPTION_MESSAGE_UNAVAILABLE
IPC_ERROR_PORT_TYPE_MISMATCH    IPC_ERROR_CAPABILITY_VIOLATION
```

Possible bits set in the `flags` parameter from `accept()` are:

```
RECV_MASK_PORT_OVERFLOW         RECV_MASK_DEADLINE_VIOLATION
```

6.3.3 Reply

```
ipc_return_t
reply(port, msg, deadline, type, importance)
port_t          port;
msg_t *         msg;
task_t          type;
deadline_t      deadline;
importance_t    importance;
```

The `reply()` call takes the passed message and sends it synchronously as a *typed reply* to the specified reply port. The calling application is suspended until the owner of the port receives the the reply in response to its `request()` call.

The SGS will expect every `reply()` call to be preceeded by a matching `accept()` call. In addition, the SGS will expect a corresponding `request()` call in another process in the same process group. The SGS is responsible for recognizing the `reply()` call as a scheduling point which defines a task boundary.

The pseudocode for this primitive is:

1. *Port Lookup.* The local name server is queried to find the location of the port.

2. *Capability Lookup.* If possible, the sending task is checked against the port capability list to enforce protection.

3. *Message Buffer Lookup.* Depending on where the destination port is, a message buffer is acquired from a free pool:

   - If the port is local to the processor, the buffer is taken directly from the free pool of that port.
   - If the port is on another node, the buffer is taken from the kernel's free pool of network transmission buffers.

4. *Message Copy.* The passed message is copied into the acquired message buffer.

5. *Message Queueing.* Depending on where the destination port is, the message buffer is queued into the appropriate list:

   - If the port is local to the processor, the buffer is queued into the received list of that port.
   - If the port is on another node, the buffer is queued into either a guaranteed or non-guaranteed transmission list, depending on the type of the message.

6. *Suspension.* The `process_suspend()` function is called to return control to the dispatcher. The dispatcher will return control to the `reply()` primitive in the context of the succeeding task when that task is scheduled to be run.

Possible error codes from `reply()` are:

```
IPC_ERROR_PORT_UNKNOWN          IPC_ERROR_CAPABILITY_VIOLATION
IPC_ERROR_PORT_OVERFLOW         IPC_ERROR_NETWORK_PORT_OVERFLOW
IPC_ERROR_PORT_TYPE_MISMATCH
```

6.4 Port Management

Other primitives are also required to create, delete, and manage ports.

6.4.1 Port Creation

```
ipc_return_t
ipc_port_create(port, name, task_type, semantic_type, capability,
                remote, queue_policy, overflow_policy)
port_t *        port;
name_t          name;
type_t          task_type;
semantic_t      semantic_type;
capability_t    capability;
```

```
boolean_t       remote;
q_policy_t      queue_policy;
flow_policy_t   overflow_policy;
```

The `ipc_port_create()` call creates a *port* for use in communication. A new port identifier is returned via the `port` parameter. The parameters include:

- `name`. This is the unique name by which the port is known.
- `task_type`. This is the type of task that can communicate with the port *synchronously*. This includes critical, essential, soft, and non real-time.
- `semantic_type`. This types the port's communication as either asynchronous, synchronous, accept, or reply.
- `capability`. This is the access control for the port. Currently not implemented.
- `remote`. This flags whether or not the port is visible to the network.
- `queue_policy`. This specifies the queueing policy for the port, currently either minimum laxity first or most recent arrival.
- `overflow_policy`. This specifies the overflow policy for the port, at the moment either discarding arriving message or discarding oldest message.

The pseudocode for this primitive is:

1. *Parameter Check.* The parameters are checked for security and consistency.
2. *Port Lookup.* The local name server is queried to make sure the port name is unique.
3. *Port Creation.* The port control structure is allocated, initialized, and linked into the port list.
4. *Buffer Allocation.* The message buffers for the port are allocated and linked into the port.
5. *Name Registration.* The port name is added to the name server.

Possible error codes from `ipc_port_create()` are:

`ERROR_IPC_PORT_ILLEGAL_ARGUMENTS` `ERROR_IPC_PORT_NAME_USED`

6.4.2 Port Deletion

```
ipc_return_t
ipc_port_delete(port)
port_t *        port;
```

The ipc_port_delete() call deletes the specified port. A valid port identifier must be supplied. The parameters include:

- port. A valid port identifier, originally determined via ipc_port_create, that is owned by this process.

The pseudocode for this primitive is:

1. *Parameter Check.* The parameters are checked for security and consistency.
2. *Port Lookup.* The local name server is queried to make sure the port exists.
3. *Buffer Recycling.* The message buffers for the port are returned to the system.
4. *Port Deletion.* The port control structure is returned to the system.
5. *Name Deregistration.* The port name is removed from the name server.

Possible error codes from ipc_port_delete() are:

```
ERROR_IPC_PORT_ILLEGAL_ARGUMENTS          ERROR_IPC_PORT_UKNOWN
```

6.5 Bandwidth Management

6.5.1 RTVC Creation

```
ipc_return_t
ipc_rtvc_create(port, bandwidth, rtvc_id)
port_t *        port;
bandwidth_t     bandwidth;
rtvc_id *        rtvc_id;
```

The ipc_rtvc_create() call creates a real-time virtual circuit (RTVC) for use in communication. A new rtvc identifier is returned via the rtvc_id parameter. The parameters include:

- port_name. This is the unique name by which the port is known.
- bandwidth. This is the amount of bandwidth required to send to the port.

- `rtvc_id`. This returns the unique identifier to the RTVC.

The pseudocode for this primitive is:

1. *Parameter Check.* The parameters are checked for security and consistency.
2. *Port Lookup.* The local name server is queried to find the location of the port named.
3. *RTVC Allocation.* The bandwidth and appropriate buffer space are allocated for the sender if possible. If the port is determined to be local, *no* bandwidth is actually allocated.

Possible error codes from `ipc_rtvc_create()` are:

```
ERROR_IPC_RTVC_ILLEGAL_ARGUMENTS          ERROR_IPC_PORT_NAME_UKNOWN
ERROR_IPC_RTVC_UNAVAILABLE
```

6.5.2 RTVC Deletion

```
ipc_return_t
ipc_rtvc_del(rtvc_id)
rtvc_id          rtvc_id;
```

The `ipc_rtvc_del()` call frees an allocated real-time virtual circuit (RTVC) used in communication. The parameters include:

- `rtvc_id`. This is a valid RTVC identifier, previously allocated via the `ipc_rtvc_create()` call.

The pseudocode for this primitive is:

1. *Parameter Check.* The parameters are checked for security and consistency.
2. *RTVC deletion.* The bandwidth and appropriate buffer space are returned to the kernel for use.

Possible error codes from `ipc_rtvc_del()` are:

```
ERROR_IPC_RTVC_ILLEGAL_ARGUMENTS          ERROR_IPC_RTVC_UNKNOWN
```

6.5.3 RTDG Creation

```
ipc_return_t
ipc_rtdg_create(port, bandwidth, rtdg_id)
port_t *        port;
bandwidth_t     bandwidth;
rtdg_id *       rtdg_id;
```

The `ipc_rtdg_create()` call creates a real-time datagram channel (RTDG) for use in communication. A new rtdg identifier is returned via the `rtdg_id` parameter. The parameters include:

- `port_name`. This is the unique name by which the port is known.
- `bandwidth`. This is the amount of bandwidth required to send to the port.
- `rtdg_id`. This returns the unique identifier to the RTDG.

The pseudocode for this primitive is:

1. *Parameter Check.* The parameters are checked for security and consistency.
2. *Port Lookup.* The local name server is queried to find the location of the port named.
3. *RTDG Allocation.* The bandwidth and appropriate buffer space are allocated for the sender if possible. If the port is determined to be local, *no* bandwidth is actually allocated.

Possible error codes from `ipc_rtdg_create()` are:

```
ERROR_IPC_RTDG_ILLEGAL_ARGUMENTS          ERROR_IPC_PORT_NAME_UKNOWN
ERROR_IPC_RTDG_UNAVAILABLE
```

6.5.4 RTDG Deletion

```
ipc_return_t
ipc_rtdg_del(rtdg_id)
rtdg_id          rtdg_id;
```

The `ipc_rtdg_del()` call frees an allocated real-time virtual circuit (RTDG) used in communication. The parameters include:

- `rtdg_id`. This is a valid RTDG identifier, previously allocated via the `ipc_rtdg_create()` call.

The pseudocode for this primitive is:

1. *Parameter Check.* The parameters are checked for security and consistency.
2. *RTDG deletion.* The bandwidth and appropriate buffer space are returned to the kernel for use.

Possible error codes from `ipc_rtdg_del()` are:

`ERROR_IPC_RTDG_ILLEGAL_ARGUMENTS` `ERROR_IPC_RTDG_UNKNOWN`

7 Implementation, Status, and Future Work

We have begun implementation using Systran's SCRAMNet distributed memory architecture as a platform for our distributed system.

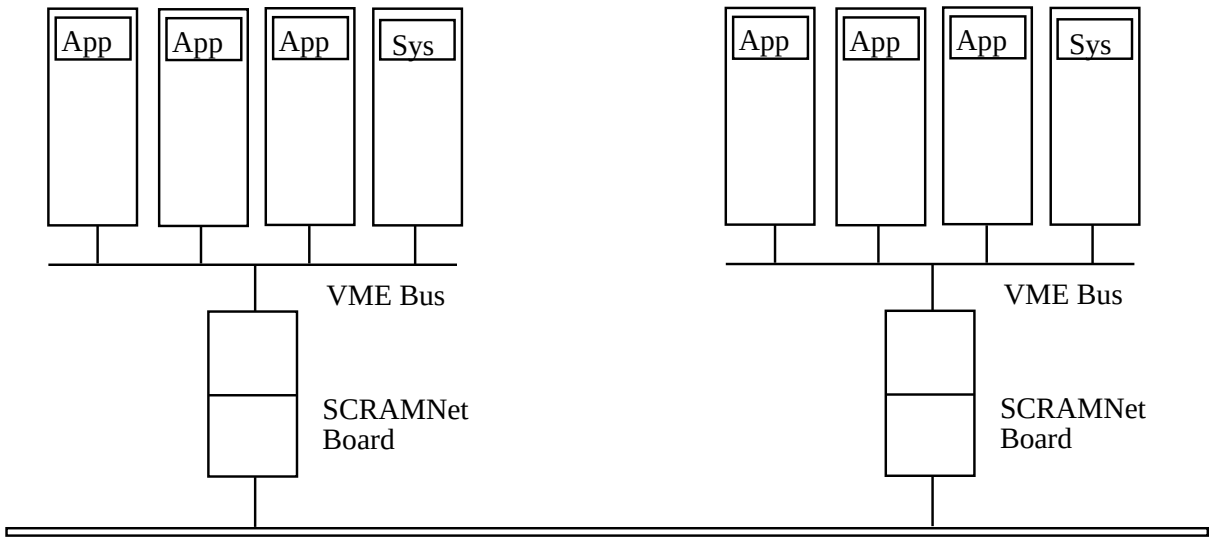

Figure 3. SCRAMNet in the Spring System

SCRAMNet is a replicated shared memory architecture that uses a fiber optic register insertion ring, running at 150 MBits/sec. Each node has 2 MB of shared common memory, and writes to this memory are broadcast (circulated) about the ring. The fiber optic medium has bit error rates of around 10^{-11} [21]. This reliability is probably greater than our node hardware, and thus we can assume that transmission errors do not decrease the reliability or predictability of our system.

We are currently implementing the IPC primitives with SCRAMNet. SCRAMNet presents some attractive features, which introduce other issues but require more investigation:

- *Distributed Scheduling.* Dynamic distributed scheduling [14] becomes very simple. The shared common RAM can be used to hold information pertaining to tasks still awaiting guarantees.
- *Fault Tolerance.* The shared common RAM can be used to hold the system task table, which describes the execution plan of the tasks. Assuming code is replicated in multiple nodes, one node can pick up another node's workload by noticing that the latter node has failed. Each node can broadcast periodic messages regarding its state.

8 Acknowledgments

This paper has benefited from discussions with K. Arvind, Lory Molesky, Doug Niehaus, and David Yates.

References

[1] Arvind, K., Ramamritham, K., and Stankovic, J.A., "A Local Area Network Architecture for Communication in Distributed Real-Time Systems", *Real-Time Systems*, Vol. 3, No. 2, May 1991.

[2] Cooper, E.C., Steenkiste, P. A., Sansom, R. D., and Zill, B.D., "Protocol Implementation on the Nectar Communication Processor" *ACM SIGCOMM '90*, Philadelphia, Pennsylvania, September 1990, pp. 135-144.

[3] Cheriton, D.R., "The V Kernel: A Software Base for Distributed Systems", *IEEE Software*, Vol. 1, No. 2, April 1984, pp. 19-42.

[4] Cheriton, D.R., and Williamson, C.L., "VMTP as the Transport Layer for High-Performance Distributed Systems", *IEEE Communications Magazine*, June 1989, pp. 37-44.

[5] Cheriton, D.R. and Kanakia, H., "The VMP Network Adapter Board (NAB): High-Performance Network Communication for Multiprocessors", *ACM SIGCOMM '88*, Stanford, California, August 1988, pp. 175-187.

[6] Damm, A., Reisinger, J., Schnakel, W., and Kopetz, H., "The Real-Time Operating System of MARS", *Operating Systems Review*, July 1989, pp. 141-157.

[7] Fitzgerald, R., and Rashid, R.F., "The Integration of Virtual Memory Management and Interprocess Communication in Accent", *ACM Transactions on Computer Systems*, Vol. 4. No. 2, May 1986, pp. 147-177.

[8] Kopetz, H. and Merker, W., "The Architecture of MARS", *Proceedings 15th FTCS*, June 1985, pp. 274-279.

[9] Molesky, L.D., Shen, C., and Zlokapa, G., "Predictable Synchronization Mechanisms for Multiprocessor Real-Time Systems", *Real-Time Systems*, Vol. 2, No. 3, pp. 163-180.

[10] Nahum, E. and Yates, D., "Real-Time IPC for the Spring Kernel", *COINS 777 Project Report*, University of Massachusetts, Amherst, MA, May 1990

[11] Niehaus, D., Molesky, L. and Stankovic, J.A., "Spring System Programming and Run-Time Models", *Spring Project Documentation*, University of Massachusetts, Amherst, MA, June 1990

[12] Peterson, J.L. and Silbershatz, A., *Operating System Concepts*, 2nd Ed., Addison-Wesley, October, 1986, pp. 349-361.

[13] Pflugel, M., Damm, A., and Schwabl, W., "Interprocess Communication in MARS", *ITG/GI Conference on Communication in Distributed Systems*, Stuttgart, Germany, February 1989.

[14] Ramamritham, K., Stankovic, J.A., and Zhao, W., "Distributed Scheduling of Tasks with Deadlines and Resource Requirements", *IEEE Transactions on Computers*, Vol. 38, No. 8, August 1989, pp. 1110-1123.

[15] Ramamritham, K., Stankovic, J.A., and Shiah, P., "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, pp. 184-194.

[16] Saltzer J.H., Reed, D.P., and Clark, D.D, "End to End Arguments in System Design", *ACM Transactions on Computer Systems*, Vol. 2. No. 2, November 1984.

[17] Sevcik, K.C., Johnson, M.J., "Cycle Time Properties of the FDDI Token Ring Protocol", *IEEE Transactions on Software Engineering*, Vol. 13, No. 3, March 1987, pp. 376-385.

[18] Stankovic, J.A. and Ramamritham, K., "The Spring Kernel: A New Paradigm for Real-Time Systems", *IEEE Software*, Vol 8., No. 3, May 1991, pp. 62-72.

[19] Stankovic, J.A., "Misconceptions about Real-Time Computing: A Serious Problem for Next-Generation Systems", *IEEE Computer*, October 1988, pp. 10-19.

[20] Stankovic, J.A. and Ramamritham, K., "The Design of the Spring Kernel", *Proc. of the 8th Real-Time Systems Symposium*, December 1987, pp. 146-157.

[21] SYSTRAN Corporation, "SCRAMNet Network Reference Manual", Dayton, Ohio, 45432.

[22] Tokuda, H., Mercer, C.W., Ishikawa, Y. and Marchok, T., "Priority Inversions in Real-Time Communication", *Proc. of the 10th Real-Time Systems Symposium*, May 1989, pp. 348-359.

[23] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R., "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System", *Proc. Eleventh ACM Symposium on Operating System Principles*, Nov 1987, pp. 63-76.