A Programming Paradigm Based on the Send-Receive-Reply Task Communication Primitives

by

Peter John Mark Baker
B.Sc., University of Victoria, 1984

A Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Applied Science
in the Department
of
Electrical and Computer Engineering

ACCEPTED
FACULTY OF GRADUATE STUDIES
DATE

[redacted]
Dr. Warren D. Little, Supervisor

[redacted]
Dr. Eric G. Manning, Departmental Member

[redacted]
Dr. Hausi A. Müller, Outside Member

[redacted]
Dr. M. Stella Atkins, External Examiner

Copyright © 1988 by Peter John Mark Baker
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part,
by any means, without the permission of the author.

QA 76.6
B32

Supervisor: Dr. Warren D. Little

Abstract

Multitasking software design and implementation is a difficult endeavour. There are a number of widely varying design approaches, but few capitalize on the power and expressiveness of particular operating system primitives. This thesis presents a programming paradigm based on the Send-Receive-Reply primitives that simplifies the design of multitasking software and makes the resultant system more understandable and, therefore, more easily maintained. Use of the paradigm also allows application programs to achieve a high degree of concurrency while avoiding deadlock and race conditions. A non-trivial example application is also presented which attempts to challenge and illustrate the programming paradigm as well as provide a prototype software control system for a proposed KAON Factory.

Examiners:

[redacted]

Dr. Warren D. Little, Supervisor

[redacted]

Dr. Eric G. Manning, Departmental Member

[redacted]

Dr. Hausi A. Müller, Outside Member

[redacted]

Dr. M. Stella Atkins, External Examiner

Contents

Tables

Figures

Listings

Acknowledgements

I wish to express my sincere thanks to the following individuals and institutions for their valued assistance in the development of this thesis:

The Natural Sciences and Engineering Research Council of Canada and the Advanced Systems Institute of British Columbia, for their generous financial assistance;

TRIUMF, for supplying an exciting and challenging application and funding its development;

The University of Victoria, for financial assistance and providing an excellent academic environment;

Morven Gentleman, for laying the foundations upon which this thesis rests; Doug Dyment, for inspiring me to pursue a Master's degree; Warren Little, my supervisor, for seeing me through with humour and friendship;

and my wife, Lynn, for her infinite moral support and many valuable suggestions for improvements to this thesis.

To my dad,

who gave me the clues, the tools, and the confidence

to explore how things work

Preface

The aim of this thesis is to present a programming paradigm that can be used in the design and implementation of large multitasking systems. The emphasis of the paradigm is on software engineering concerns specific to these types of systems.

This thesis is divided into two parts. Part I describes the paradigm. Background on its development is provided in Chapter 1. Chapter 2 describes several message passing mechanisms and justifies the choice of the Send-Receive-Reply primitives as the basis for the paradigm. Chapter 3 describes the components of the paradigm in detail and forms the core of this thesis. Part II describes an example application, the control system of a proposed KAON Factory, to illustrate the use of the paradigm. Chapter 4 provides a brief overview of the application, Chapter 5 describes the development and runtime environments used to develop the application, and Chapter 6 describes the design and implementation of the control system software. Conclusions and suggestions for further research are presented in Chapter 7. Issues relevant, but not central, to the programming paradigm are reserved for the appendices.

The scope of this thesis is restricted to software engineering issues important in the development of real-time, multitasking, multiprocessor software. Although the paradigm can be used to develop both hard and soft real-time systems, a general mechanism for

satisfying hard real-time constraints is not suggested. Rather, design techniques are suggested which do not restrict the adoption of techniques for achieving other important and desirable characteristics of this type of software. It is expected that the reader is familiar with the concept of tasks and task communication mechanisms. This thesis should be particularly useful for anyone considering the development of a large multitasking system.

PART I

THE PROGRAMMING PARADIGM

Chapter 1

Introduction

1.1. Why is a Programming Paradigm Necessary?

Large, complex computer applications with demanding timing constraints and interfaces to the real world can be viewed on three levels of abstraction. The hardware upon which an application runs is the lowest level and presents a controlled view of the real world. To satisfy the demand for more powerful and appropriate hardware, multiprocessor architectures have been recognized as the only feasible solution, a few of which are described in [GREN83, MOKH84, SIMP86, WILS84]. The second level of abstraction is the operating system, which may provide the generally accepted model of multiple communicating tasks to the application programmer. Tasks also provide a natural mechanism for the separation of concerns [PARN88]. The third level of abstraction is then the application program consisting of a number of tasks which directly use the task communication primitives provided by the operating system. These three levels of abstraction are faced by application programmers when confronted with developing a large multitasking system. However, problems of preventing deadlock, avoiding critical race conditions, achieving concurrency, and structuring task hierarchies remain to be solved for

each application. Clearly, another level of abstraction is necessary.

This thesis proposes a level of abstraction between that of an application and the multiple communicating task model provided by an operating system. Various useful task types based on the Send-Receive-Reply task communication model [CHER79, GENT81] and upon which a large class of multitasking, multiprocessing, real-time applications can be constructed are characterized. The communication semantics of the task types are designed such that static detection and, consequently, avoidance of deadlock is possible. The particular communication primitives upon which the paradigm is based allow the existence of multiple sequential tasks without the need for shared memory, thereby avoiding race conditions arising from uncontrolled access to memory. Concurrency can be achieved using many cooperating tasks within the structure of the paradigm and multiprocessor hardware can be exploited to achieve true concurrency. Task structuring and, consequently, system design and maintenance are simplified since the tasks required by most applications are provided and *standardized* by the programming paradigm. The Send-Receive-Reply programming paradigm can effectively aid the design and implementation of correct, efficient, and *maintainable* real-time systems.

1.2. Background

Concurrent programming techniques have their roots in the late 1960's and early 1970's when the concepts of semaphores [DIJK68] and monitors [HOAR74] were introduced for task synchronization. However, these primitives assume a shared memory environment for data transfer which makes them obsolete in light of recent advances in distributed processing. In 1978, the concept of Communicating Sequential Processes (CSP) was introduced [HOAR78]. Since tasks cooperate via synchronous messages both

synchronization and data transfer are embodied within a simple set of message primitives. Although more general than previous shared memory primitives, the CSP primitives require that a task know the identity of all tasks with which it communicates. This is a serious deficiency since it prevents straightforward implementation of libraries where arbitrary tasks make requests of utility tasks. Since CSP, numerous other communication models have been suggested, but they all suffer from various deficiencies [CASH80].

Many popular commercial real-time operating systems in current use are based on the concepts that were introduced in the late 1960's and early 1970's and have not advanced significantly since that period. Data General's real-time operating system, RTOS, emerged about 1972 [DYME85]. Prominent real-time operating systems for microcomputers are VRTX from Ready Systems [VRTX87, HAUS87], pSOS from Software Components Group[PSOS82], and MTOS from Industrial Programming Inc. [MTOS84]. These operating systems are all based on a combination of shared memory, CSP, and other communication primitives. In an attempt to support multiprocessor architectures, specialized primitives have been added to these (often already cumbersome) operating systems. As a result, none of these operating systems effectively addresses all important issues, in fact they all introduce their own problems.

In 1979, the Send-Receive-Reply primitives were introduced [CHER79]. These primitives provide both synchronization and bi-directional data transfer unlike other synchronous message primitives which provide only uni-directional data transfer. In the CSP model a task typically sends data to another task as in the traditional producer-consumer problem. In the Send-Receive-Reply model, messages initially seem backwards since the sender usually sends a *request* to another task, and the receiving task replies the

results to the sender. Part of the request message defines the type of operation required of the destination task and the flow of conventional data is typically from the receiver to the sender. This model becomes extremely attractive as will be seen when the primitives are described in more detail.

At least four operating systems based on the Send-Receive-Reply primitives are available. Harmony [GENT83] and its commercial derivative, Unison [MTI87], are high-performance real-time operating systems for tightly coupled multiprocessors. QNX [QUAN88] and Port [MALC81] are both microcomputer based network operating systems and the V-kernel [CHER84] is a workstation network operating system. Software written for these operating systems can exploit hardware re-configuration, such as the addition of processors, without major software changes and will migrate effectively to the next generation of real-time computation. It is for such operating systems that the programming paradigm presented here is targeted.

The programming paradigm has been evolving since about 1979 when the Send-Receive-Reply primitives were first introduced. Descriptions of some of the paradigm components (e.g. clients, servers, workers, couriers, and notifiers) have been mentioned in [GENT81], but their application as a software engineering tool has not been emphasized. Vultures have been discussed in [ATKI85]. Agents, watchdogs, and buffered notifiers are the result of the author's experience with Harmony and Unison over the last four years. In its evolving forms, the paradigm has been used by the author in the development of a high performance multiprocessor three dimensional graphics renderer and a real-time videotape controller [VERT86], a real-time digital voice messaging system [BAKE87a], and has culminated in the development of a prototype control system for the KAON Factory described in Part II of this thesis.

1.3. Terminology

Parr defines a *real-time* system to be "a computer system which is somehow electromechanically interfaced to an external physical or biological processes and which must be able to capture or generate, in a timely manner, asynchronous signals produced by the process or signals to accepted by the process ... a *hard* real-time system must be able to *guarantee* performance and responsiveness within time constraints imposed by the nature of the external process or the system is considered to have failed, while a *soft* real-time system must at least be able to *approximate* them" [PARR86]. This definition of real-time is adopted throughout this thesis. The distinction between hard and soft real-time systems is not made since the paradigm is applicable to both.

A *task* is a unit of sequential execution on a single processor and has state, priority, a stack, and a context. A *multitasking* system is one which permits many such tasks to exist concurrently and share resources. Throughout this thesis the term *task* is used rather than *process* to avoid possible confusion with other uses of the root word *process*.

The term *multiprocessor* refers to a single system with more than one autonomous CPU and possibly sharing busses and peripherals but not necessarily sharing memory. *Transparent* multiprocessing [GENT83], where communicating tasks do not know on which processors their correspondents reside, is an extremely desirable characteristic since software can be written to be largely independent of the hardware architecture. Although the paradigm does not require transparent multiprocessing, it is assumed for convenience.

1.4. Notation

1.4.1. Blocking Diagrams

Chapter 3 provides blocking diagrams for each of the component task types. Blocking diagrams illustrate the dynamic blocking characteristics of communicating tasks. The diagrams do not show messages passed during task initialization since initialization messages do not contribute to the dynamic blocking characteristics.

In the blocking diagrams a task is represented by an ellipse and a message between two tasks is represented by a directed arc between ellipses. An arc is directed from the sending task, which blocks, to the receiving task as shown in Figure 1.1. Since the reply message is non-blocking, it is not shown in the blocking diagram but may be assumed. Special arrows are used to indicate different types of messages such as interrupt events where a blocking send is not used ( ), or messages that are never actually sent but a corresponding receive is executed ( ) .

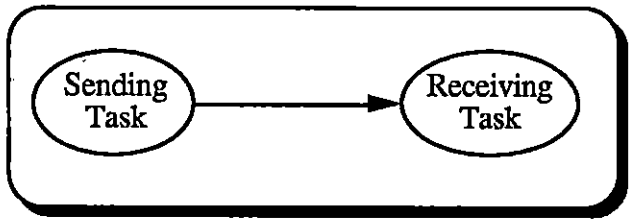


Figure 1.1 — Blocking Diagram Notation

1.4.2. Code Templates

Chapter 3 provides code templates for each of the component task types. The code

fragments are based on the C programming language [KERN78], but a set of macros is used for clarity (see Appendix C for the macro definitions). Many implementation details are omitted to emphasize the code structure. Data types, messages in particular, are used in the templates but their definitions are omitted since they are heavily application dependent. Parameterized types are used to ensure consistency and to simplify the construction of new message types (see Appendix E for a description of parameterized types in C).

Chapter 2

Message-Based Task Communication

2.1. Introduction

A message-based task communication model is essential for good designs of systems involving many communicating tasks distributed across multiple processors. Although other inter-task communication models exist [CASH80], *message passing* can elegantly encapsulate both task synchronization and data transfer into a small set of simple primitives having well-defined semantics.

Message passing models are categorized according to whether the *send* primitive is blocking or non-blocking [GENT81]. In either case, the *receive* primitive is typically blocking since use of a non-blocking receive creates a polling situation which is an undesirable characteristic of a multitasking system. A further specialization of blocking message passing introduces the *reply* primitive. It is this last model that is adopted as the basis for the programming paradigm presented in Chapter 3. A description of each of these models and the motivation for the selection of the *Send-Receive-Reply* model is the topic of this chapter.

Message addressing (i.e. the means of identifying the communicating tasks) is also

discussed briefly in this chapter since it plays an intrinsic part in the semantics of message passing. Message format and communications failure are important issues which are not addressed since they are not directly relevant to the development of a programming paradigm based on a message passing model of task communication. The programming paradigm functions independently of these issues.

2.2. Non-blocking Message Passing

Non-blocking message passing is characterized by a non-blocking send primitive where the task issuing the send does not block awaiting synchronization with the receiver but continues to execute subsequent instructions. The corresponding receive primitive may be either blocking or non-blocking. This model is referred to as the asynchronous or non-blocking Send-Receive (S-R) model.

Because task synchronization does not occur during a non-blocking send, the non-blocking S-R model must handle the situation where the receiving task is not ready to accept the message. Either the kernel buffers the message, the sending task buffers the message, or the send fails and the kernel returns an error code to the sending task. Each of these cases introduces problems for which only unsatisfactory solutions may be constructed. The blocking characteristics of the non-blocking S-R model with message buffering are illustrated in Figure 2.1.

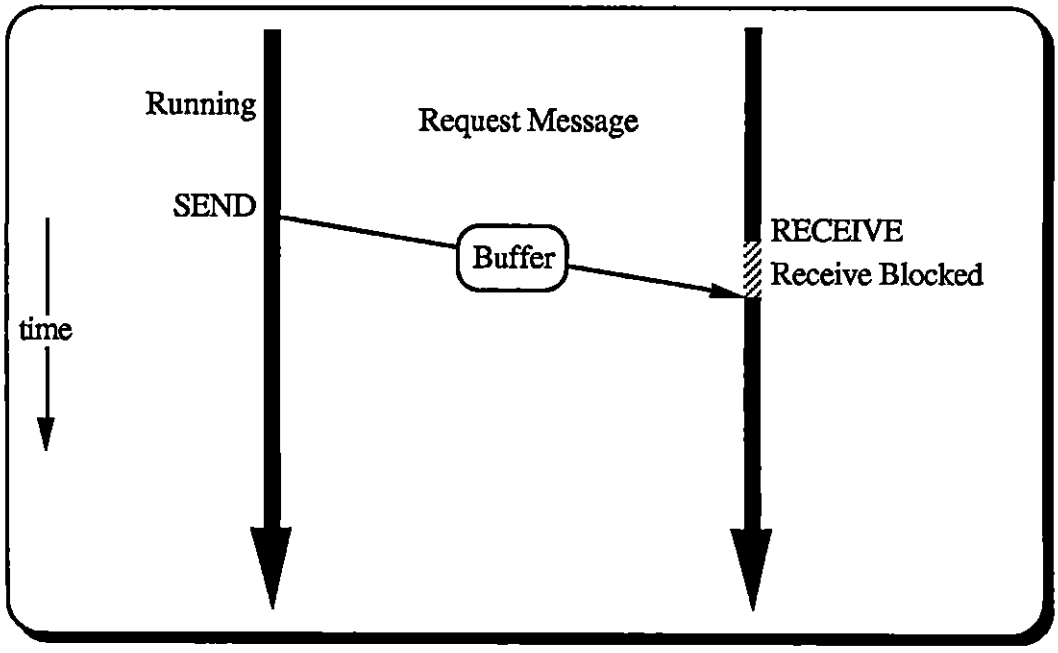


Figure 2.1 — Asynchronous Send-Receive (with buffering) Blocking Characteristics

If the kernel buffers the message, it is possible to exhaust the memory pool from which buffers are allocated, particularly if an ill-behaved task performs many sends to a task that rarely receives messages. When the pool is empty, the kernel either blocks the sending task or returns a failure code. The first approach can lead to deadlock since tasks assuming a non-blocking send may send to each other then execute a receive, as is necessary for task synchronization. If both sends block due to an exhausted memory pool, the tasks become deadlocked. The second approach results in a more complex programming environment since every time a task sends, it must consider what to do in the case of a failure due to message pool exhaustion (e.g. try again?, delay then try again?, or abort?). The pSOS operating system [PSOS82] is an example of a system based on the non-blocking S-R model in which the kernel buffers pending messages and returns a

failure code if the memory pool becomes exhausted. Buffer allocation overhead is also an important consideration since the overhead must be small and the algorithm bounded if real-time performance is required. Bounded time memory allocation algorithms are often less space efficient than non-bounded time algorithms.

If the sending task buffers messages, then this task must not modify messages that have been sent but not yet received. Therefore, the sending task must know when messages have been processed by the receiver so that they can be released. While the first problem places a burden on the programmer, the second problem introduces the need for an alternate communication mechanism. Assuming that the receiving task cannot release the message,[1] either the kernel or the receiving task must inform the sending task that the message transaction is complete and indicate which of the possibly many pending messages can be released. Since this communication involves the transfer of information, and using a message will result in the same problem in reverse, an alternate communication mechanism must be used.

If a non-blocking send is unbuffered, then the kernel must return an error code to the sending task when the receiving task is not ready to accept the message. In response, the sending task must either attempt the send primitive periodically, until the message is accepted, or it must ensure that the receiving task is ready to receive the message before sending. Although the first approach allows the sending task to perform other computations while waiting for the receiving task to accept the message, it is a polling situation. The second approach introduces the need for an alternate synchronization

---

1 In a shared memory environment where *all* memory is shared, it may be possible for the receiver to release the message but this violates modularity and the performance penalty might be prohibitive in a distributed memory environment.

mechanism since the message itself is not used for synchronization.

Non-blocking message passing allows a sending task to perform other activities while a message is processed by a receiving task. This appears to be a desirable characteristic because of the apparent increase in concurrency. However, solutions to the inevitable problem of a receiving task that is not ready to accept a message are unsatisfactory. So the question arises: *"why should a single task be responsible for more than one activity?"* If a task is sending a message as a work request for which it requires a result before continuing, then there is no need for a non-blocking send. If the sending task can continue with an alternate activity, then the same functionality and degree of concurrency can be achieved simply by decomposing the task into two, or more, tasks[2].

2.3. Blocking Message Passing

Blocking message passing is characterized by a blocking send primitive where the task issuing the send blocks until the receiving task executes the receive primitive. The corresponding receive primitive blocks until a message is available. A non-blocking receive may be implemented in addition to a blocking receive. This model is referred to as the blocking or synchronous Send-Receive model. The blocking S-R model can be extended with a third primitive, *reply*, which is non-blocking. This model is referred to as the Send-Receive-Reply (S-R-R) model.

2.3.1. Synchronous Send-Receive

The blocking characteristics of the synchronous S-R model are illustrated in Figure 2.2. When a sending task sends to a receiving task that is not ready to accept a

---

2 The degree of task granularity also depends on the overhead of messages and tasks.

message, the sending task becomes send-blocked. When the receiving task executes the receive primitive, the kernel transfers the contents of the sender's message buffer to the receiver's message buffer. Both the sender and the receiver are then ready to continue execution, possibly simultaneously in time if multiple processors are employed. If a sending task sends to a receiving task that is already receive-blocked, the kernel immediately transfers the message to the receiver and both tasks continue execution.

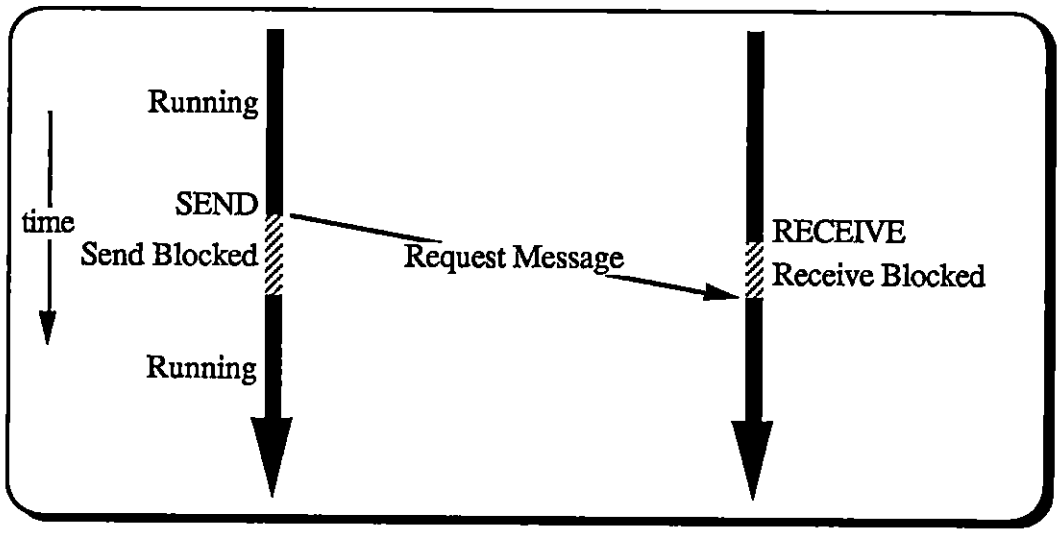


Figure 2.2 — Synchronous Send-Receive Blocking Characteristics

This model does not share the buffer related problems suffered by the asynchronous S-R model. Message buffering is not required since a message is only transferred when the receiving task is ready to accept. A single message buffer for each communicating task is sufficient. This model also inherently provides synchronization since the correspondents cannot continue until *both* tasks have executed their message primitives. Task synchronization occurs upon message transfer. Furthermore, concurrency is achieved

through the use of many tasks (if cheap enough) and each task is a completely sequential execution unit. These constraints lead to a more effectively decomposable and understandable system.

2.3.2. Send-Receive-Reply

The blocking characteristics of the Send-Receive-Reply model are illustrated in Figure 2.3. A sending task is send-blocked until the receiver executes the corresponding receive primitive. At this point, the message is transferred to the receiver, the receiver continues execution and the sender is now reply-blocked. The sender remains reply-blocked until the receiver issues the reply primitive. The reply primitive does not suffer the problems of a non-blocking send primitive since the sender is known to be blocked and no message buffering is required. A key characteristic of the reply primitive is that the receiver can reply to messages in arbitrary order, unrelated to the order in which messages are received. This characteristic and its importance is discussed in greater detail in Chapter 3.

An extra primitive, Forward, is provided by some S-R-R based operating systems such as Thoth [CHER79], the V-Kernel [CHER84], and QNX [QUAN88]. This primitive is not required by the programming paradigm for the reasons discussed in detail in Appendix B.

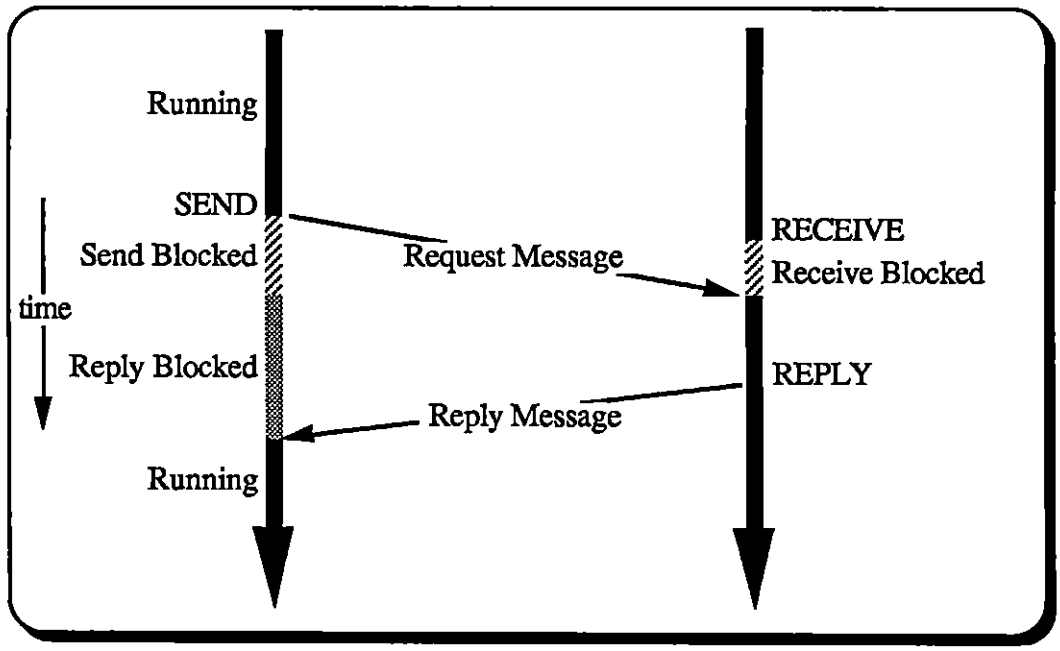


Figure 2.3 — Send-Receive-Reply Blocking Characteristics

A Send-Receive-Reply transaction has two possible state transition sequences, depending upon which primitive, send or receive, is executed first. These are illustrated in Figures 2.4 and 2.5. The state transitions indicate the number of context switches required for a message transaction.

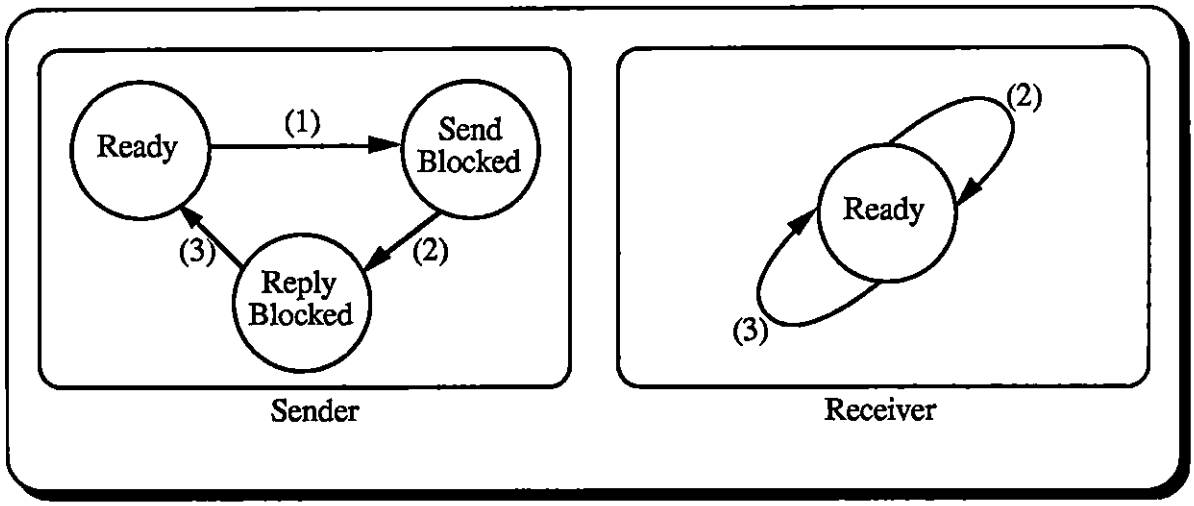


Figure 2.4 — Send-Receive-Reply State Transitions

If the send primitive is executed first, the state transition sequence is as follows:

(1) The sender *sends* a message and becomes *send-blocked*

(2) The receiver *receives* (and does not block since a message is waiting); the kernel transfers the message contents from the sender to the receiver; the sender is now *reply-blocked*

(3) The receiver *replies* and unblocks the sender; the kernel transfers the message contents from the receiver to the sender

Note that the receiver is never blocked in this case.

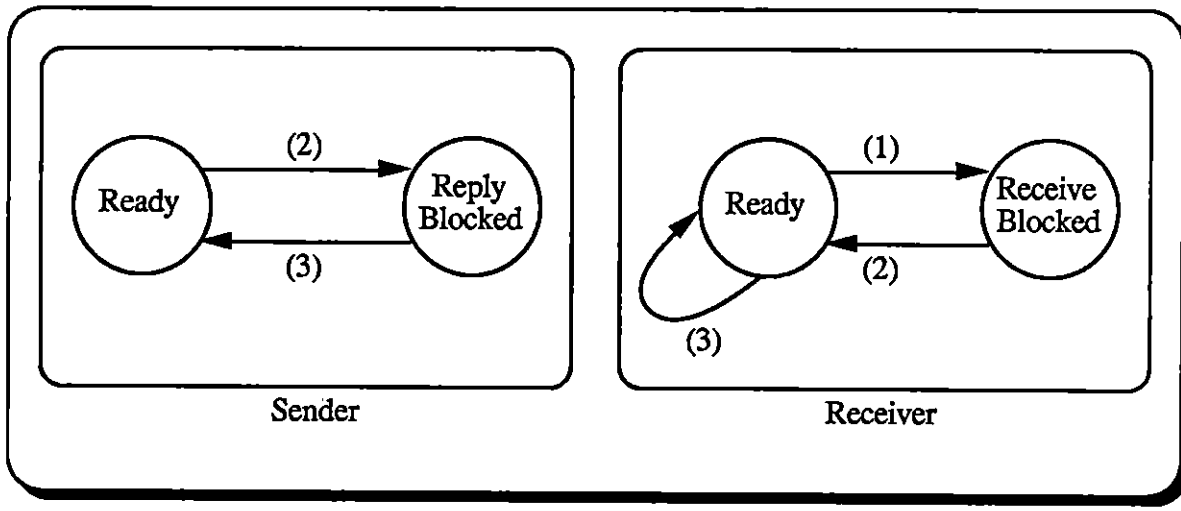


Figure 2.5 — Receive-Send-Reply State Transitions

If the receive primitive is executed first, the state transition sequence is as follows:

(1) The receiver *receives* and becomes *receive-blocked*

(2) The sender *sends* a message and unblocks the receiver; the kernel transfers the message contents from the sender to the receiver; the sender is now *reply-blocked*

(3) The receiver *replies* and unblocks the sender; the kernel transfers the message contents from the receiver to the sender

2.4. Comparison of Send-Receive-Reply with Other Models

2.4.1. Send-Receive-Reply and Synchronous Send-Receive versus Asynchronous Send-Receive

The Send-Receive-Reply and the synchronous Send-Receive models enjoy a number

of advantages over the asynchronous Send-Receive model. First, since no message buffering is required, the problems resulting from message buffering disappear. In particular, it is much easier to make statements about program correctness since the operating system does not introduce non-determinism (due to message buffer exhaustion). Second, there is no need for a separate synchronization mechanism since the messages themselves transfer only when the corresponding tasks are in well-defined states. Finally, tasks based on the Send-Receive-Reply model are completely sequential and thus easier to understand than the potentially non-sequential tasks based on the asynchronous Send-Receive model. This issue is of paramount importance in large systems where long-term maintenance is critical and the system must be as easy to understand as possible. The apparent disadvantage of lack of concurrency is counteracted by the introduction of many small sequential tasks. In particular, the programming paradigm of Chapter 3 illustrates how parallelism can be achieved using sets of cooperating tasks. Note, however, that achieving good concurrency with the Send-Receive-Reply model can be more difficult in systems where task overhead[3] is very large.

2.4.2. Send-Receive-Reply versus Synchronous Send-Receive

Synchronous Send-Receive does not suffer many of the problems of asynchronous Send-Receive, however, it only provides a simplex communication path. That is, information only flows from the sender to the receiver. If it is necessary to communicate results from the receiver back to the sender, then an independent communication is required. While on the surface this does not seem like a problem, its manifestation is

---

3 Task overhead includes the time and memory overhead incurred during task creation as well as the time overhead of context switching.

threefold.

First, it is easy to reach deadlock since we have two corresponding tasks, both sending and receiving. If, due to some programming error, they both send to each other (or both receive) then deadlock occurs. Figure 2.6 illustrates send-send deadlock and Figure 2.7 illustrates receive-receive deadlock. The Send-Receive-Reply model provides a simple mechanism for statically detecting and avoiding deadlock. Deadlock can be detected in a task blocking diagram by the presence of a message cycle. Figure 2.8 illustrates this type of deadlock. Deadlock is avoided by eliminating all cycles. If there are no cycles in the blocking diagram, then deadlock cannot occur. The programming paradigm described in Chapter 3 describes a number of task types, some of which are designed expressly for avoiding message cycles.

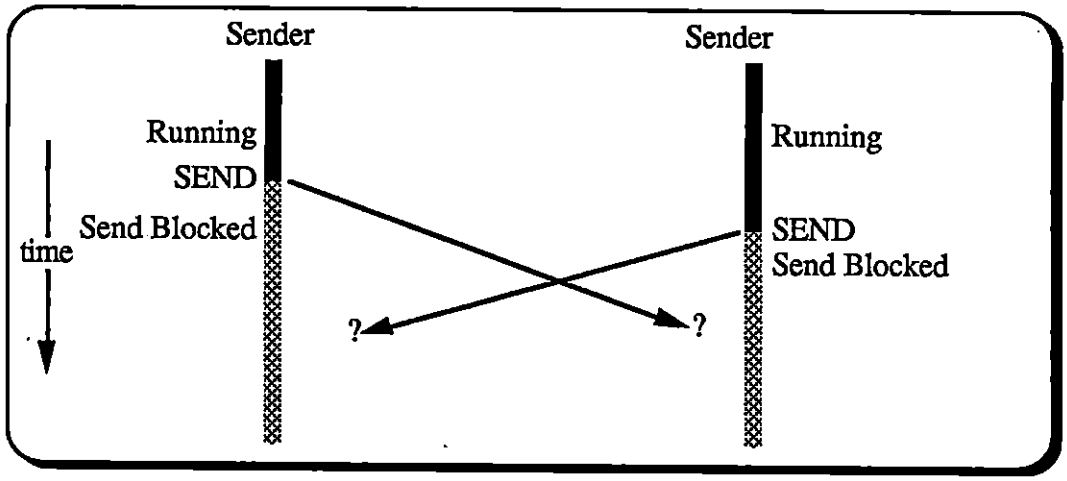


Figure 2.6 — Send-Send Deadlock

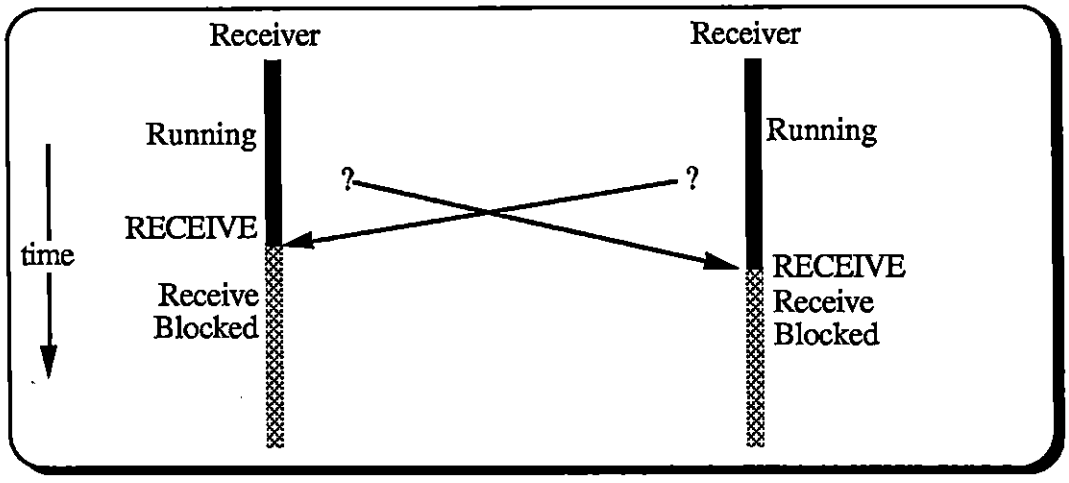


Figure 2.7 — Receive-Receive Deadlock

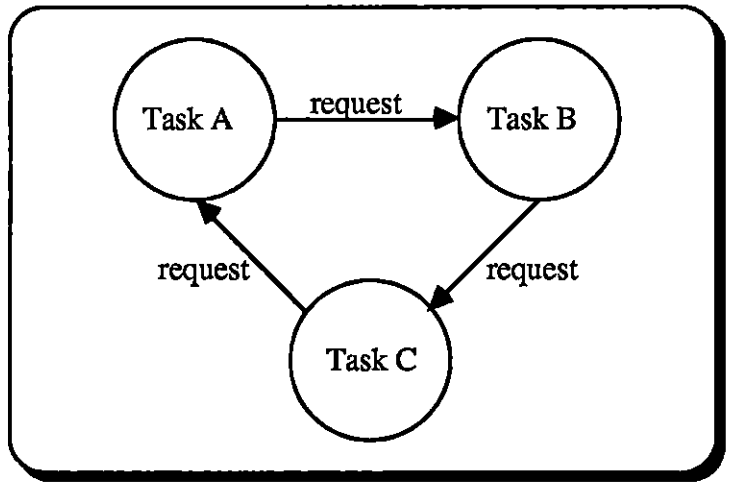


Figure 2.8 — Send-Receive-Reply: Message Cycle Deadlock

To illustrate the second problem, consider a client task sending a message to a server task. If the server is to communicate any results back to the client, then the server must explicitly send a response message to the client. The server (an important shared resource)

is dependent on the client. If the client fails to execute a receive or delays before executing a receive, then the server becomes send-blocked on an unknown, potentially unreliable task. Figure 2.9 illustrates this condition. This dependency must be resolved by the use of timeouts or a special non-blocking send primitive. The special non-blocking send primitive is the reply primitive of the Send-Receive-Reply model. The Send-Receive-Reply model provides a safe mechanism for servers or other important tasks. In this case the server only receives and replies messages. Since it never sends, it is never dependent upon a client or other external tasks. When results need to be replied to a client, the client is known to be reply-blocked on the server due to the semantics of the Send-Receive-Reply primitives.

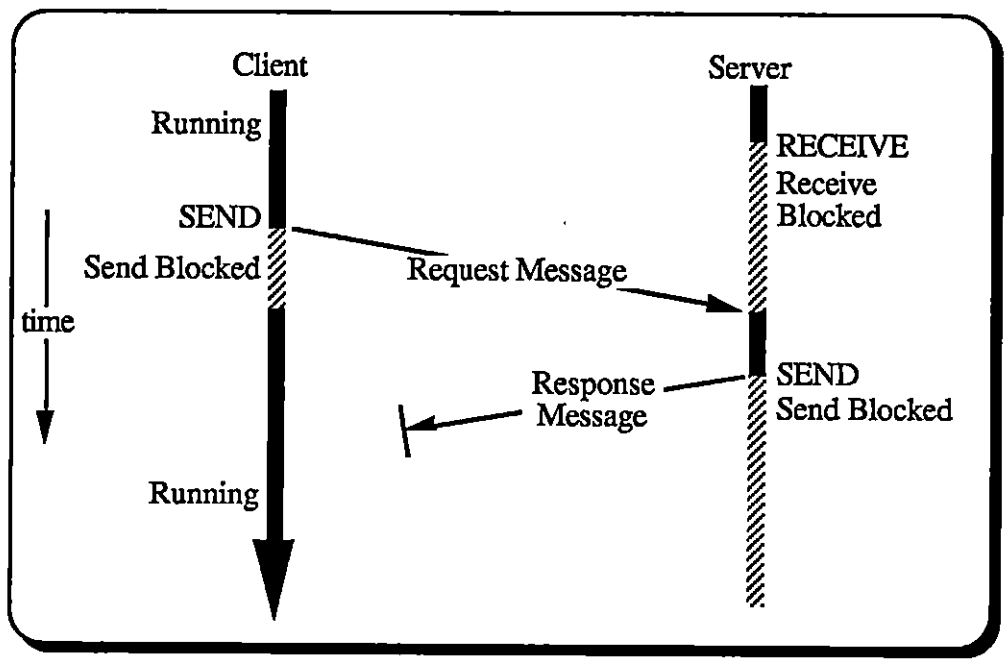


Figure 2.9 — A Server Send-blocked on an Unreliable Client

Third, a data driven programming methodology often results from the synchronous Send-Receive model. Tasks communicate data to other tasks. While this seems logical, it

has problems similar to a sequential program that uses many global data types and global variables. Frequent passing of data between many tasks means that many tasks must know the format and meaning of the data. Such lack of data encapsulation violates the principle of information hiding [PARN72]. The Send-Receive-Reply model encourages an object-oriented approach [BOOC83, GOLD83, BYTE81, STRO86] where a request message is sent to an object (a task), the object performs some processing on the internal representation, and finally replies the result of the request to the sender. It is this approach and the resultant task models which are the focus of this thesis.

2.4.3. Send-Receive-Reply versus Ada's Rendezvous

The Ada programming language [ICHB79] is one of the few commercial languages designed to support tasks and strongly typed inter-task communication. Ada's communication mechanism is based on the rendezvous concept. While rendezvous is a procedure-oriented task communication mechanism, its properties are similar to the Send-Receive-Reply model with the restriction that messages are replied to in the same order in which they were received. The Ada rendezvous consists of a calling task passing input parameters to a called task that may return output parameters to the calling task. The calling task is analogous to a sending task and the called task is analogous to a receiving (and replying) task. The invocation is conceptually equivalent to the send primitive. The receiver executes an *accept* statement that is conceptually equivalent to a receive primitive. The sender is blocked until the receiver task completes the execution of the accept statement,[4] possibly returning some results in output parameters. The input parameters

---

4 The accept statement permits compound statements to be included in the body of the statement.

are similar to a request message. The output parameters are similar to a reply message. The key difference is that the rendezvous must reply to messages in the order that they are received.[5] The advantages of an arbitrary reply order, as is possible with the Send-Receive-Reply model, will become apparent in Chapter 3.

Ada's strong typing is a significant advantage over many message-based systems where messages are often untyped. However, in Ada, distribution of tasks is difficult, especially in dynamic environments as pointed out by Gammage and Casey [GAMM85]. Ada has other problems when it comes to real-time programming, as described in [CORN86]. In [BOAR84] it is stated that "... Ada was designed for one-processor systems or for homogeneous multiprocessors with shared memory." It is likely that a change to the language specification is required to provide good support for future networks of real-time multiprocessors.

2.5. Message Addressing

Message addressing is the manner in which communicating tasks are identified. There are a wide variety of message addressing schemes, such as Unix pipes [RITC78], message ports [CASH80, MAO80], and message links [CASH80, BASK79]. Each method has its own semantics that are not necessarily directly comparable to the others. One general characteristic that can be compared and is relevant to the programming paradigm is the mapping between senders and receivers.

The mapping between senders and receivers may be one-to-one, many-to-one or one-to-many and any combination of these schemes may be incorporated into a system. One-

---

5 The effect of arbitrary replies is possible in Ada, however, only in a clumsy and inefficient manner involving nested accept statements.

to-one messages, such as in Hoare's CSP [HOAR78] have limited usefulness by themselves since it is impossible for library resources such as generic servers to know the names of all potential clients. Many-to-one communications, as in Harmony, Unison, Brinch Hansen's Distributed Processes [HANS78], and Ada's rendezvous allow simple support of servers and other tasks that do not know the senders in advance. One-to-many communications are also possible. The semantics of broadcast communications can be such that the message is complete when the first receiver has received the message or when all potential receivers have received the message. One-to-many communications are sometimes used in networks to distribute names or other information without knowing the actual names or addresses of other machines on the network.

2.6. Summary

The Send-Receive-Reply model has simple yet versatile message semantics. The same set of primitives provide both synchronization and bi-directional data transfer. This means that no message buffering is necessary during data transfer and no extra synchronization primitives are required. A simple and effective method of static deadlock detection is inherent in the model removing this concern from dynamic analysis. Given sufficiently low overhead tasks, a high degree of concurrency can be achieved through the use of many simple tasks. True concurrency can be achieved if these tasks are distributed across multiprocessors. These characteristics make the Send-Receive-Reply model for task communication a prime candidate for the basis of a design methodology for large multitasking systems.

Chapter 3

Paradigm Component Descriptions

3.1. Introduction

The programming paradigm presented in this chapter is an attempt to distil and characterize various useful task types based on the Send-Receive-Reply task communication model described in Chapter 2. It is the claim of this thesis that, given low overhead for task creation and destruction[1] and an inexpensive message passing mechanism[2], the S-R-R paradigm is helpful to the design and implementation of efficient and *maintainable* large multitasking systems.

All task types can be implemented using an operating system that supports the S-R-R model although their implementation may differ depending on the particular details of the message passing semantics. Differences between two such operating systems, Harmony

---

1 Overhead for task creation and destruction must be low enough to permit dynamic task creation and not hinder real-time activities. The actual times depend on application specific requirements.

2 A message passing mechanism is considered inexpensive if the message passing overhead does not significantly increase the cost of an operation. Consequently, message passing overhead dictates the minimum amount of work that a task should perform.

and Unison, are described where necessary to provide examples of alternative implementations. Appendix A provides a more detailed comparison between the kernels of these two operating systems.

A general overview of the programming paradigm is first presented followed by detailed descriptions of each of the task types that comprise the paradigm. A summary is also provided to reinforce the properties of each task type.

3.2. Overview

Figure 3.1 depicts the task structure of a sample application that makes use of all the task types in the S-R-R programming paradigm. A brief walkthrough of this system introduces each of the task types, their typical interconnections, and their general properties. Excerpts from this figure are then used in the detailed descriptions of the task types in the following section.

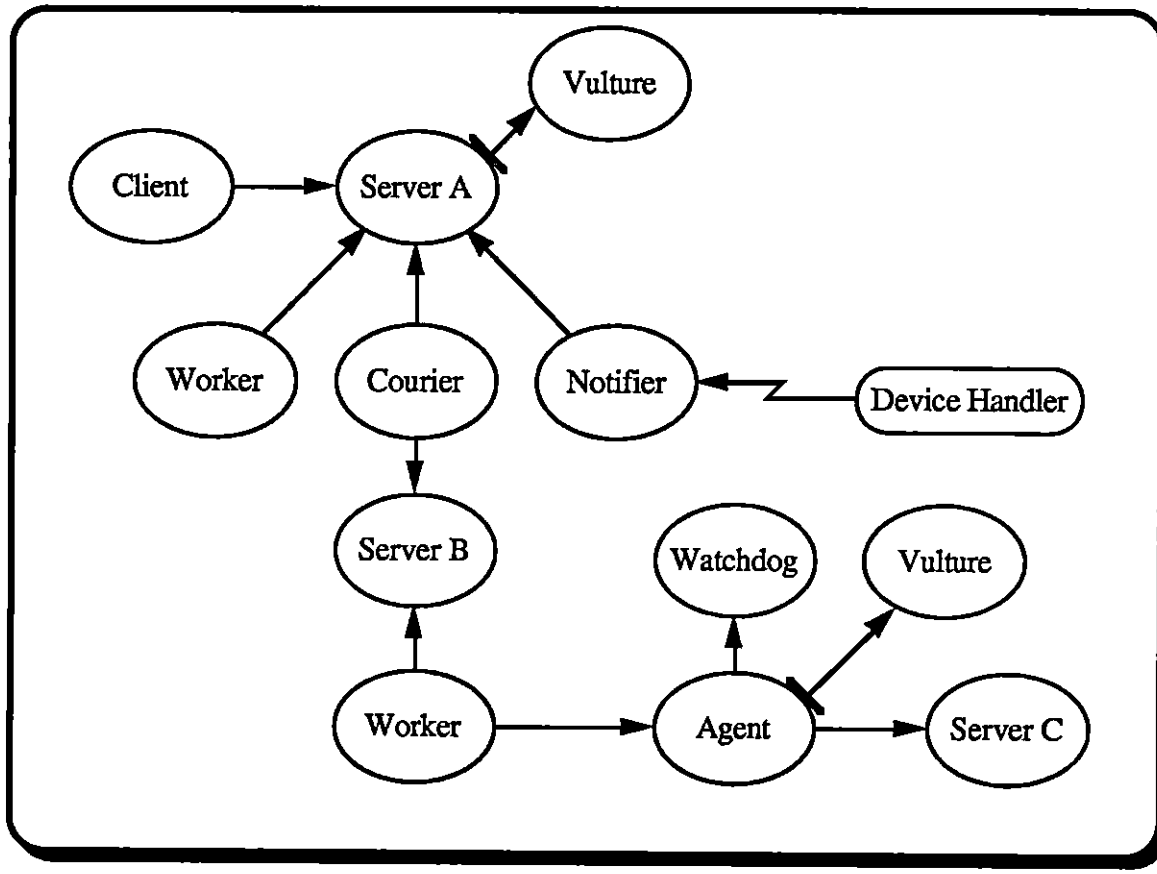


Figure 3.1 — Task Structure of a Sample Application

Beginning at the top left corner of Figure 3.1, a client sends a request to **server A** to perform some operation outside the client's scope. Typically there may be many clients sending requests to a particular server. Because server A is both important (i.e. it must continue to exist) and possibly unreliable (i.e. it may terminate abnormally) it has a **vulture** attached to it. This vulture performs a receive-specific from server A. The special arrow ↙ indicates that the server never sends to the vulture. Consequently, the vulture is only unblocked when the server terminates[3]. At this time, the vulture may either re-create the

3 Appendix A describes task termination semantics.

server, or send a message to some other task indicating that the server has terminated. Server A also employs a **worker** task. Typically, there are one or more worker tasks for each client communicating with the server. This worker task sends to server A, indicating its willingness to perform some work on the server's behalf. A **courier** task is employed to send a request to another task without causing server A to block. A **notifier** is used to inform server A of some external event. This notifier is effectively receive blocked on a device handler, as indicated by the special arrow  . When the notifier deems it necessary to inform server A of some event (e.g. a full buffer), it sends server A a notification message.

From the perspective of **Server B**, the courier is its client. Server B also employs a worker task to perform some processing on its behalf. This worker requires the services of **server C**, which is a device specific server. So that the worker remains device independent, an **agent** is employed to perform communication protocol conversion. The worker communicates with the agent using a device independent protocol whereas the agent communicates with server C using a device dependent protocol. Since the agent is important and possibly unreliable due to its dependency on a device specific server, two other tasks protect it from possibly blocking forever. The agent informs a **watchdog** whenever communication to server C is about to begin. If the agent remains blocked on server C beyond a specific time interval, the watchdog terminates the agent. When this occurs, the **vulture** detects the termination and re-creates the agent. The new agent is then used by the worker for subsequent operations.

This overview has introduced each of the task types in the programming paradigm and described a possible scenario to illustrate their use. Such scenarios are typical of large

multitasking systems; however, a thorough understanding of the characteristics of each task type is necessary to effectively apply the paradigm to a particular problem. The following section provides a detailed description of each task type.

3.3. Task Type Descriptions

The S-R-R programming paradigm consists of eight component task types: clients, servers, workers, couriers, notifiers, agents, vultures, and watchdogs. For each task type its role is presented, followed by a description of useful variations including those described in the literature, the communication semantics peculiar to the task type, especially the blocking characteristics, the lifecycle of task instances, other characteristics (such as the response time), and finally, a code template.

3.3.1. Clients

Role

Clients are the most general form of task and may perform almost any type of operation. Typically, a client task will send request messages to other tasks, such as servers and agents, in order to have some operation performed on its behalf. Clients are often designed as clients, however other task types, such as couriers, may temporarily take on the role of a client when they send request messages to a server. Since clients are especially application specific, most of the following discussion will be *in general* since clients do not have as strict communication semantics as other task types.

Variations

Clients have been discussed in the literature as part of the *client-server* model. This model typically refers to *client* tasks sending requests to *server* tasks, but usually refers only to the concept of the request message and not to the message passing mechanism or the particular communication semantics involved as presented here.

Communication Semantics

Client tasks typically send request messages to servers but since they may actually be another type of task and only temporarily acting as clients, they may also receive messages. When a client requires the services of a server, it sends a request message to the server (or other service-providing task), requesting that an operation be performed on its behalf. The client then waits, send-blocked, until the correspondent[4] has performed the work. When

4 A task communicates with another task, the *correspondent.*

a client is not engaged in communications with a server, it may communicate with other tasks according to their communication protocols. Since clients do not provide a general service to other tasks, they do not register with the Name Server[5] and are therefore known only to their creators and any recipients of client request messages.

Figure 3.2 shows a client which makes use of three different servers, each server providing a different type of service.

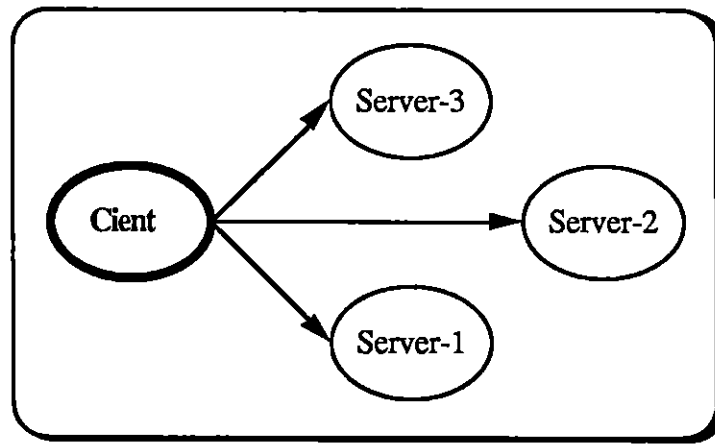


Figure 3.2 — Client Blocking Diagram

Life Cycle

Clients are typically created by a system configuration task and often exist for the duration of an application. Clients can, however, be created and destroyed many times throughout the lifetime of the application. Clients are generally responsible for their own intentional destruction, but they may be destroyed by watchdogs (see Section 3.3.8) or other tasks.

---

5 The Name Server is a well-known system supplied server that provides mapping between symbolic names and task ids.

Other Characteristics

Clients range in complexity from a simple task that merely creates a system of other tasks and waits forever, to complex tasks executing sophisticated algorithms. Clients do not generally have critical response times, though they may require servers to respond to requests within strict time constraints. When a client makes a time-critical request of a server, it may employ a watchdog to ensure the operation does not exceed the maximum time.

Code Template

Listing 3.1 describes a client task that makes requests of a single connectionless server. During initialization, the client obtains the task ID of the server from the Name Server. As with most tasks, the client goes into a loop after initialization. In each iteration of the loop, the client sends a request to the server, (whimsically) asking the server to pull a rabbit out of a hat. The client is blocked until the server replies at which time the client performs some processing based on the result of its request. This cycle repeats until the client terminates.

Listing 3.1 — Client Code Template

```
TASK Client()
{
    ServerRqst  rqst;
    ServerRply  rply;
    TaskId      magicServer;

    /*
     * perform local initialization then
     * find the ID of the server "MagicServer"
     */
    initialize();
    magicServer = GetServerId( "MagicServer" );

    WHILE( TRUE )
        do_local_processing();

        rqst.MSG_TYPE = PULL_RABBIT_FROM_HAT;
        send( &rqst, &rply, magicServer );

        SWITCH( rply.RESULT )
            CASE( NO_RABBIT )
                /* error handling */
            ENDCASE

            CASE( BIG_RABBIT )
                /* deal with big rabbit */
            ENDCASE

            CASE( SMALL_RABBIT )
                /* deal with small rabbit */
            ENDCASE

            DEFAULT
                /* Error: unknown result */
            ENDCASE
        ENDSWITCH
    ENDWHILE
}
```

3.3.2. Servers

Role

The clients described in Section 3.3.1 make use of servers which provide services to many unknown clients. To achieve this, a server must make itself available to other tasks and be prepared to respond to requests from other tasks. Since it is a shared resource, it must perform operations as quickly as possible and must not fail as a result of an error or failure on the part of a correspondent.

A server is analogous to a library subroutine except that it can maintain state across service transactions. Servers are a key structuring concept, around which other tasks play a subservient role. Many of the task types described in this paradigm are designed to fill some role in aiding a server.

Servers provide a variety of resource management facilities. Access can be provided to shared resources such as input/output devices, data structures, or special purpose processors. Because it accepts only one request message at a time, a server serializes accesses to its resource, avoiding race conditions that often plague multitasking applications. Although it only *accepts* one request at a time, a server may serve multiple requests concurrently since it can accept other requests before completing the service of a previous request.[6] A server can process requests in any order that is appropriate and, via workers and multiple processors, can achieve a high degree of concurrency. Resources allocated to clients can be automatically released when the client is destroyed (see Appendix A). Unless specifically mentioned, the term *server* refers to either connection-based or

---

6 Replying in a different order from the request messages is a key characteristic of the S-R-R primitives which makes them more attractive than other mechanisms such as Ada's rendezvous.

connectionless servers.

Variations

There are two types of servers. Connection-based servers require a client to *open* a connection prior to making requests of the server. Connectionless or transaction-based or datagram-based servers respond to *individual* client requests on a request-by-request basis. Connectionless servers are referred to as *administrators,* introduced by Gentleman in [GENT81].

A connection-based server provides access to a resource that has sequential characteristics or requires automatic reclamation in the event of client destruction. Two examples of connection-based servers are: the Harmony file server providing access to sequential files, and a window server providing access to windows created on a display. The file server needs to maintain an internal state on behalf of the client so it can access the correct *next* record in the file. The window server, having created windows on the client's behalf, must be informed of the client's termination so that it may delete the windows and reclaim resources allocated on the client's behalf. The operating system causes each open connection to be closed when a client terminates as explained in Appendix A.

An administrator administers resources that are used on a request-by-request basis. Resources allocated on behalf of a client are released at the completion of each request. For example, a clock administrator may provide services such as GET_TIME, DELAY_FOR, and DELAY_UNTIL. Each of these operations is independent of previous or subsequent operations. If its client suddenly terminates, the clock administrator need not be informed since no resources remain allocated after its reply to the client.

Another example of an administrator is a simple buffer administrator, shown in

Figure 3.3, for solving the standard bounded buffer problem [DIJK65]. Whenever the producer has data for the consumer, it sends the data to the buffer administrator. If the buffer is full, the administrator receives specifically from the consumer, and the producer remains send-blocked until there is room in the buffer. If the buffer is empty, the administrator receives specifically from the producer, and the consumer will remain send-blocked until there is data in the buffer. If the buffer is partially full, the buffer administrator receives messages from either the producer or the consumer.

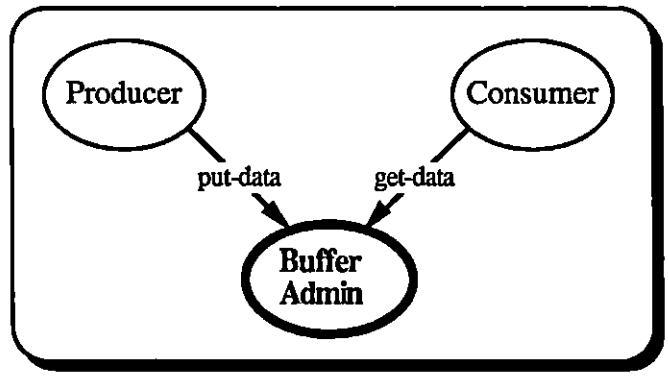


Figure 3.3 — Buffer Administrator

A *guardian* in Argus [LISK88] is a cluster of tasks (called *processes*) that performs a similar function to a server and its associated workers, couriers, and notifiers. There are a few important differences. First, Argus guardians are permitted to make requests of other guardians whereas the servers in the S-R-R paradigm are prohibited from directly sending requests to any other tasks. This restriction helps avoid deadlock, ensures quick server response, and isolates the server from unreliable correspondents. Second, the tasks that make up a guardian have, by default, direct access to all of the guardian's objects. Shared

objects introduce the possibility of race conditions if the objects are accessed in an uncontrolled fashion. In the S-R-R paradigm, shared objects are rarely needed (if at all) and must be explicitly exported. Finally, all the processes of a guardian must exist on a single node of a network whereas a server and its workers, couriers and notifiers can be distributed across multiple processors.

Communication Semantics

Servers receive requests from arbitrary and unknown client tasks. During its execution, a server may also receive other messages, not request messages but report-for-work messages from couriers and workers and notification messages from notifiers. These tasks are created by the server to improve server response. Typically, servers make direct use of workers, couriers, notifiers, and vultures. Through couriers, servers also make use of other servers and agents.

Since servers provide service to unknown clients, the server must make itself known to potential clients. It does so by registering a symbolic name and its task ID with the Name Server.

A server must be ready to respond to client requests as quickly as possible. Consequently, a server must never execute a send primitive. Sending to another task means that the server is dependent on the correct operation of the correspondent task. If the correspondent does not reply or does not reply soon enough, the server will block and be unable to respond to *any* client requests. A single malfunctioning task could cause the server and all its clients to block indefinitely; this ripple effect must be avoided. Servers must not even send to their helper tasks since they cannot predict when the helper task will be ready to accept the message. Therefore, all helper tasks must send to the server as

shown in the blocking diagram of Figure 3.4.

Any number of helper tasks may be used by the server. The number and type of each helper task is determined according to the server's needs and the overhead incurred by task creation, maintenance, and destruction.

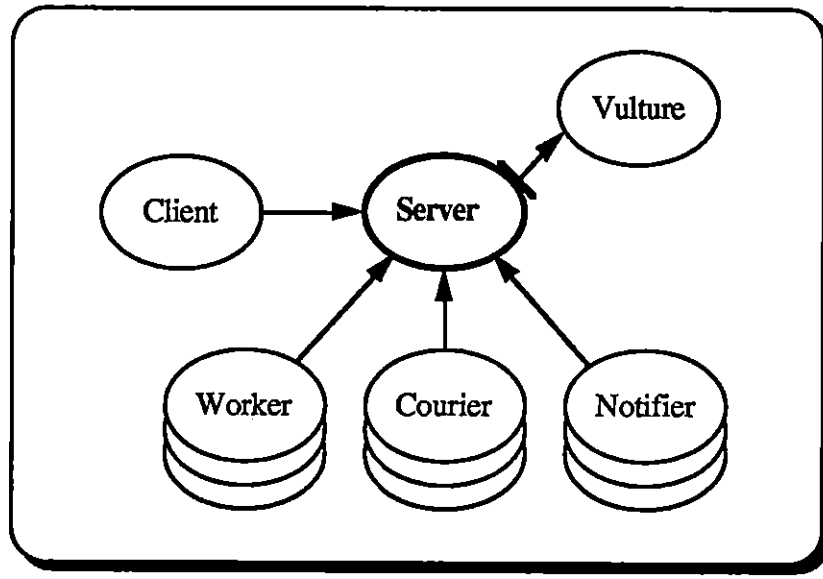


Figure 3.4 — Server Blocking Diagram

Life Cycle

Servers are created by a system configuration task. Since they are available to all clients, they usually exist for the duration of the application program. Servers occasionally destroy themselves when an unrecoverable error occurs. In this case, it may be useful to have a vulture re-create the server.

Other Characteristics

It is important that servers respond as quickly as possible to client requests, even if the requests are not immediately satisfied. Servers often provide a variety of related operations of varying duration. A client requesting a short operation should not be delayed due to the processing of another client's lengthy request. The server uses workers, couriers, and notifiers to off-load lengthy or blocking operations so that it can respond quickly to client requests.

Servers are typically the most complex of the task types described in this paradigm. The complexity arises since a) servers respond to requests from many clients and may have to maintain state information for each client or open connection, and b) internally, servers are not entirely synchronous since there may be many pending client requests, some of which have been assigned to workers and others relayed to other servers via couriers. The asynchronicity allows the server to achieve a high degree of concurrency but also increases its complexity since it must maintain the state of all partially completed requests.

Code Template

The connection-based server of Listing 3.2 performs some initialization and then registers its name with the Name Server. Once the server registers its name, it becomes available to clients. It is important that the server does not register its name until initialization is complete since there may be some communication with workers, couriers and notifiers during initialization and this communication could be confused with client requests. After initialization, the server goes into its loop, awaiting requests from clients and helper tasks. The loop repeats forever, processing messages from clients and workers.

If couriers were necessary, they would be created on demand in a similar fashion to the workers. If a notifier was required, it would be created during initialization.

If a client wishes to OPEN a connection, the server allocates a connection for the client and creates a worker for the client. The server does not reply to the client until the worker has reported that it is available for work.

If the client CLOSEs a connection, the server merely destroys the worker, deallocates the connection and replies to the client. All client resources are now reclaimed. When a client is terminated both Harmony and Unison provide an automatic CLOSE message for each open connection.

If the client sends a READ or WRITE request, the request is *replied* to the worker. The worker is now executing concurrently with the server. The client is reply-blocked on the server and remains so until the worker replies to the server.

If the worker sends a REPORT_FOR_WORK message to the server, then one of two things happens. If this is the first time the worker has reported to the server, the client is still blocked from issuing the OPEN request. The server merely replies to the client. If the worker is sending results of an operation, then the server copies the pertinent data into a reply message and replies this information to the client. In either case, the client is unblocked and the worker is reply-blocked on the server.

The code template for an administrator is similar to that shown in Listing 3.2, though somewhat simpler. An administrator does not require the OPEN and CLOSE message processing and does not require the maintenance of connections.

Listing 3.2 — Server Code Template: Dynamic Worker Creation

```
TASK Server()
{
    ServerRqst      rqst;
    ServerRply      rply;
    Connection      connection;

    /*
     * Initialization: the last part of initialization is
     * registering our name with the Name Server. Everything
     * must be operational before allowing client requests.
     */
    create_notifiers();
    other_initialization();
    register_name( SERVER_NAME, MyId() );

    WHILE( TRUE )
        requestor = receive( &rqst, ANY_TASK );

        SWITCH( rqst.MSG_TYPE )
            CASE( OPEN )
                connection = allocate_connection();
                connection.WORKER_ID = create_client_worker();
                connection.CLIENT_ID = requestor;
                /*
                 * Don't reply to the client until the worker
                 * has reported for work.
                 */
            ENDCASE

            CASE( CLOSE )
                connection = lookup_connection(
                    rqst.close.CONNECTION_ID );
                destroy( connection.WORKER_ID );
                release_connection( connection );
                rply.close.RESULT = OK;
                reply( &rply.close, requestor );
            ENDCASE

            /* continued on next page */
```

```
CASE( READ )
    /*
     * If the client issues this request,
     * then we know the worker is ready (since the
     * client is always blocked when the worker is
     * busy). So, reply the work request to the
     * worker.
     */
    connection = lookup_connection(
        rqst.read.CONNECTION_ID );
    /*
     * Reply the work request to the worker
     */
    rply.worker.READ_RQST = rqst.read;
    reply( &rply, connection.WORKER_ID );

    /*
     * The client is still reply blocked on us
     */
ENDCASE

CASE( WRITE )
    /*
     * Same situation as for READ
     */
    connection = lookup_connection(
        rqst.read.CONNECTION_ID );
    /*
     * Reply the work request to the worker
     */
    rply.worker.WRITE_RQST = rqst.write;
    reply( &rply, connection.WORKER_ID );

    /*
     * The client is still reply blocked on us
     */
ENDCASE

/* continued on next page */
```

```
            CASE( REPORT_FOR_WORK )
                connection = lookup_connection_by_worker(
                    requestor );
                IF( rqst.report.RESULT = NULL )
                    /*
                     * First time worker is reporting.  The
                     * worker is ready so we unblock the
                     * client
                     */
                    rply.open.CONNECTION_ID =
                        connection.CONNECTION_ID;
                    reply( &rply.open, connection.CLIENT_ID
                    );
                ELSE
                    /*
                     * The worker has just finished either a
                     * read or write operation for the client.
                     * Reply the results to the client.
                     */
                    IF( rqst.report.OPERATION == READ )
                        /*
                         * The RESULT field contains both the
                         * data and and error code
                         */
                        rply.read.RESULT = rqst.report.RESULT;
                        reply( &rply.read,
                            connection.CLIENT_ID );
                    ELSE
                        rply.write.RESULT =
                        rqst.report.RESULT;
                        reply( &rply.write,
                            connection.CLIENT_ID );
                    ENDIF
                ENDIF
            ENDCASE

            DEFAULT
                /* error processing */
            ENDCASE
        ENDSWITCH
    ENDWHILE
}
```

3.3.3. Workers

Role

If some server operations are lengthy and could delay processing of other short operations, worker tasks are employed by the server to perform the lengthy operations while the server continues to respond to subsequent client requests. In this manner, the desired degree of concurrency can be achieved.

Variations

Workers are employed by servers when a server operation 1) is too lengthy, and may delay servicing of subsequent client requests, 2) involves send-blocking the server on another task, making the server vulnerable, or 3) causes the server to become system dependent. A worker is often a member of a pool of similar workers, all employed by a single server. The individual workers may execute on different processors, thereby distributing the computation.

The first use of workers is to enhance concurrency and improve server response. When a server needs to perform some lengthy operation, a worker is employed to perform the operation on behalf of the server. Since the server and worker may execute concurrently, the server is free to respond to other requests, many of which may be short and easily satisfied. This type of configuration can achieve time simultaneity since the workers can be distributed across multiple processors.

The second use of workers is to prevent a server from send-blocking. When a server requires the services of another server, it must make its request without sending. To achieve this, the server makes use of a special type of worker called a *courier,* described in

Section 3.3.4, which sends to both servers.

The third use of workers isolates system dependent software from the server. Since it is often desirable to isolate system dependent code into modules, a worker task is a convenient container in which to encapsulate the system dependent code. In this case, a system independent communication protocol between the server and the worker is established. The worker contains all system dependent code. When the software needs to be ported to a new system, then only the worker needs to be modified. While it is possible to isolate the system dependent code into subroutines within the server, maintenance programmers would still be required to investigate a potentially complex piece of software, the server. Also, it may still be possible for programmers to intentionally or accidentally violate module boundaries, whereas the physical separation between workers and servers makes this unlikely.

Communication Semantics

Messages between workers and servers are sometimes considered backwards from normal messages. Generally, we think of the sender of a message sending a work request to the receiver (following the traditional client-server model). However, in the case of the worker, a report-*for*-work message is sent to its server. The server replies to the worker when it has some work to be performed. Note that the server uses the non-blocking reply primitive to issue its request to the worker, so that the server is not blocked if the worker malfunctions. The server is able to concurrently process incoming client requests while previous requests are processed by workers. When a worker is finished, it sends another report-for-work message to the server, along with the results of the previous operation.

During the course of its work, a worker will often make use of other servers and

agents. A worker will sometimes communicate directly with a client. If large amounts of data must be moved from a client to the server and subsequently to the worker, then the server may, upon opening a connection, set up communication paths such that the client communicates directly with the worker. In this case, the worker receives requests from the client, acting as a sub-server. The server, meanwhile, is responsible for creating the worker, establishing communication paths, enforcing access rights, and terminating the worker when the connection is closed or the client is destroyed. This approach violates module boundaries and is used only when the extra performance outweighs maintenance considerations.

Figure 3.5 shows the communication blocking diagram for a worker. The worker sends a report-for-work message to its server and blocks. When it gets some work from the server, it unblocks and carries out the work by sending a work request to Server-B and sending a message to the agent. Finally, when it has finished the assigned work, it sends the results, together with a request for more work to its server. This cycle repeats until the server determines that the worker is no longer needed and destroys the worker.

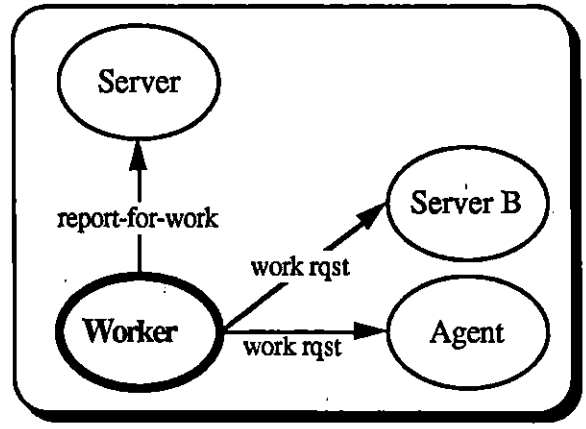


Figure 3.5 — Worker Blocking Diagram

Life Cycle

In environments where task creation overhead is low, workers are usually created on demand. For example, a connection-based server may create one or two workers for each open connection and terminate the workers when the connection is closed. If task creation overhead is too high to permit this, the server may create a pool of workers at initialization time. If task creation overhead is *very* low, a worker is created each time the server wants some operation performed. This may only be feasible if task support is assisted by hardware.

Depending on the creation scheme in use, a worker will exist either for the duration of a single operation or for many operations. Workers are terminated by the server that created them.

Other Characteristics

A worker does not usually have a specific response time, rather, it reports to the

server whenever it has finished the designated job. Each job performed by the worker may take a variable length of time.

Workers vary from very simple tasks, such as couriers, to very complex tasks, such as a task executing a three-dimensional shading algorithm.

Code Template

The worker template of Listing 3.3 describes a simple input/output worker task that performs two types of operations. The first is on behalf of the server which is requesting the operation *SERVER_OP1* on behalf of some client. Two message transactions are needed for each occurrence of operation SERVER_OP1. The client must send a request to the server and then the server must relay (via reply) the message to the worker. When the worker is finished, it sends the results to the server and the server then replies to the client.

The second type of operation involves direct communication between the client and the worker. In this case, the server informs the worker of the client's task ID, and informs the client of the worker's task ID. The client then sends requests directly to the worker and the worker replies results directly to the client. While this mechanism is more efficient when many small requests are necessary and caching is not possible, it has a few disadvantages. The client may malfunction and never send messages to the worker. In this case, the worker remains receive blocked and the penalty is wasted resources; however, the server itself is not impeded. More importantly, module boundaries may be violated, resulting in a system that is more difficult to maintain. Because of these disadvantages, direct client/worker communication should only be used when performance requirements demand the reduced message traffic.

Listing 3.3 — Worker Code Template

```
TASK Worker()
{
    WorkerRqst  rqst;
    WorkerRply  rply;

    /*
     * We have not yet done anything for the server, and
     * hence have no results yet.
     */
    rqst.RESULT = NULL;

    WHILE( TRUE )
        /*
         * Tell our parent (the server) that we are ready to
         * perform some work.  If we have just finished some
         * work, then rqst.RESULT contains the result of that
         * work.
         */
        rqst.MSG_TYPE = REPORT_FOR_WORK;
        send( &rqst, &rply, ParentId() );

        /* Decide what work the server wants us to perform */
        SWITCH( rply.WORK_TYPE )
            /*
             * A client has asked the server to perform
             * operation 1. We will do it for the server and
             * send the results to the server in our next
             * 'REPORT_FOR_WORK' message.
             */
            CASE( SERVER_OP1 )
                rqst.RESULT = operation1( rply );
            ENDCASE

            /* continued on next page */
```

```
            /*
             * A client has indicated to the server that it
             * wants to do a number of short operations.  The
             * server has told the client this worker's ID
             * and has told this worker the client's ID.
             * We now engage in direct communications with the
             * client.
             * Note that when communicating with the client,
             * the client sends to us.  When communicating
             * with the server, we send to the server.
             */
            CASE( CLIENT_OPERATIONS )
                ClientRqst  clientRqst;
                TaskId      client;

                client = rply.CLIENT_ID;

                WHILE (receive( &clientRqst, client ) !=
                        CLOSE )
                    handle_client_rqst( &clientRqst );
                ENDWHILE

                /*
                 * When we next send 'REPORT_FOR_WORK' to the
                 * server, tell it that we are finished
                 * talking with the client.
                 */
                rqst.RESULT = CLIENT_DONE;
            ENDCASE

            DEFAULT
                /* error */
            ENDCASE
        ENDSWITCH
    ENDWHILE
}
```

3.3.4. Couriers

Role

Since servers need to respond to incoming messages as quickly as possible, it is not acceptable for a server to *send* to another task. This would cause the server to send-block thereby delaying the servicing of client requests. Send-blocking would also make the server vulnerable since it would rely upon the other task to reply. If the other task malfunctions, then the server becomes permanently blocked and unable to service *any* requests. To enable one server to communicate with another server without sending, a courier task is employed. The courier sends to both its server and the destination server, thus preventing either server from sending. Couriers do not usually perform any protocol conversion or manipulation of the message. They are merely go-betweens whose sole function is to prevent their server (or any two tasks) from blocking.

Variations

Nielsen and Shumate discuss couriers [NIEL87] with respect to object-oriented Ada programming. In that paper, couriers are called *transporters*. Transporters are defined to be tasks that *call* one task to obtain data and *call* another task to deliver the data. Recall from Chapter 2 that Ada's *call* is similar to *send* in the S-R-R model.

A courier task is a special case of a worker task, however it is described separately from workers because it is so frequently used and performs a logically different function.

Communication Semantics

As with other types of workers, the courier sends a report-for-work message to its

server. When the server needs to send a message, it *replies* the message to the courier. The courier then sends the message to the destination task. Thus, the courier, not the server, becomes send blocked. When the courier is replied to, it sends the reply message to its server. Figure 3.6 depicts the blocking semantics of a courier task.

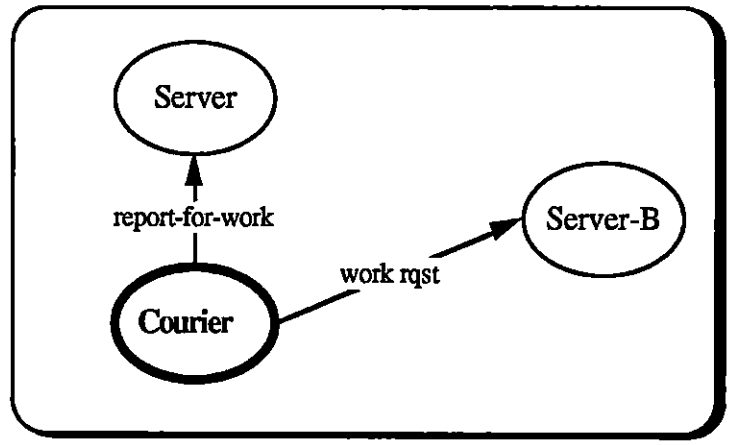


Figure 3.6 — Courier Blocking Diagram

A courier is only known to its parent and to any tasks with which it initiates correspondence. Couriers are generally associated with servers, however they may be used between any two tasks that do not wish to *send* messages. The buffered notifier described in Section 3.3.5 uses couriers.

Life Cycle

As with workers, couriers may be created upon demand or maintained in a pool, created at initialization time, depending upon the frequency of use and the overhead of task creation. Couriers may exist for only a single transaction or may exist for longer, depending on task overhead. Couriers are destroyed by their creator.

Other Characteristics

A courier is one of the simplest task types and is easy to implement. If polymorphic types [CLEA86] are available in the implementation programming language they can be coded as a generic task and hence are application independent. If polymorphic types are not available, couriers need to be coded with specific types for each server.

The response time of a courier is dependent upon the destination server to which it sends requests. The response time of the courier does not affect the response of its server since the courier and its server are running concurrently.

Code Template

The courier shown in Listing 3.4 performs its work in an infinite loop. It sends a REPORT_FOR_WORK message to its creator, indicating that it is ready to send a message to another task on behalf of its creator. When its creator replies, the courier copies the reply message (or some portion thereof) into a request message. This request message is then sent to the destination task. When the destination task replies, the courier *sends* the contents of the reply message to its creator and, at the same time, indicates that it is ready for the next message.

Listing 3.4 — Courier Code Template

```
TASK Courier()
{
    WorkRqst        workRqst;
    WorkRply        workRply;
    ServerRqst  serverRqst;
    ServerRply  serverRply;

    /*
     * No results for our parent on the first report for work
     */
    workRqst.REPLY = NULL;

    WHILE( TRUE )
        /*
         * Tell our parent server that we are ready for work
         */
        workRqst.MSG_TYPE = REPORT_FOR_WORK;
        send( &workRqst, &workRply, ParentId() );

        /*
         * We have a request for the other server
         */
        serverRqst = workRply.DESTINATION_REQUEST;
        send( &serverRqst, &serverRply,
        workRply.DESTINATION_ID );

        /*
         * Now send the reply from the other server to our
         * parent on the next report for work.
         */
        workRqst.REPLY = serverRply;
    ENDWHILE
}
```

3.3.5. Notifiers

Role

Servers sometimes provide access to hardware and hence need to respond to changes in hardware state. In order to do so, they must be capable of responding to hardware generated events (e.g. interrupts). However, it is not acceptable for a server to block, awaiting a hardware event, nor should a server execute at a high priority which is often necessary for event handlers since much of the server's processing is of lower priority. A notifier is used to solve both problems.

A notifier blocks, awaiting the desired event or interrupt, possibly performing event filtering and processing, and sends processed events to its server. The notifier may be hardware dependent, especially if it services hardware generated interrupts.

Variations

Circumstances may demand that a single notifier notify different tasks about different events. In this case the notifier is informed, at initialization time, of the mapping between events and task IDs.

Simulations are often straightforward when notifiers are used, since a simulation task to generate pseudo-events for the notifier. The entire software system, from the notifier upwards, can be tested before the actual hardware generated events are available.

A notifier may be a single task, as in a simple notifier, or it may consist of a few tasks designed to perform the same function as a simple notifier. A buffered notifier often consists of a few cooperating tasks. The server is unaware of this distinction since it is only concerned with receiving notification messages.

Communication Semantics

A simple notifier blocks, awaiting an event from some hardware or software system. The event can be a message from another task or possibly a hardware generated interrupt. When the event occurs and the notifier deems it necessary to inform some task of the event, it sends an event message to the task. Figure 3.7 illustrates the blocking semantics of a simple notifier.

A notifier does not register with the Name Server since it only communicates with its server and the event mechanism. During initialization, its server will often send it an initialization message. This message may inform the notifier of the logical interrupt to be monitored, the type of events in which the server is interested or other configuration information.

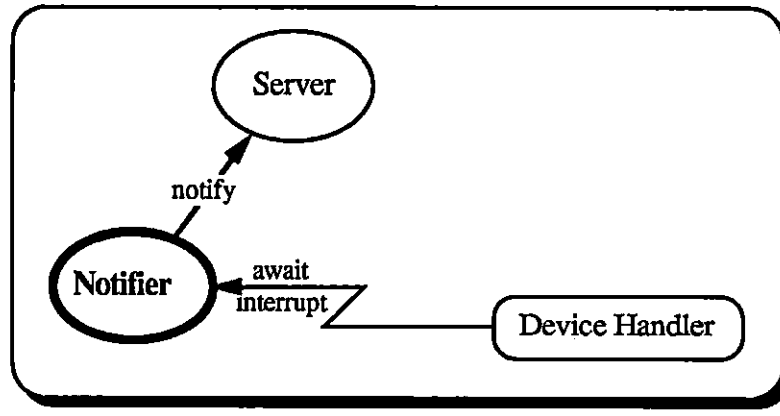


Figure 3.7 — Simple Notifier Blocking Diagram

Notifiers are often high-priority tasks with short execution cycles, and it is important that when they *send* to the server that the server replies very quickly. If the server is too slow,[7] then it is possible that the notifier may miss the next event while blocked on the server. To avoid this problem it may be necessary to build an notifier that buffers events until the server is ready to accept them. Note that in this configuration, shown in Figure 3.8, the notifier never sends and is therefore never send-blocked. The notifier is always ready to receive either a request for a notify event (from a courier) or an actual event. When an event occurs, the notifier is ready to receive it.

The message transactions of the buffering notifier are as follows. The courier sends a request-for-notify message to the notifier. If the notifier has an event buffered, it gives the event to the courier which then sends the event to the server. If no event is ready, the courier remains reply-blocked on the notifier until an event arrives. If an event arrives when the courier is not reply-blocked, the event is buffered. The buffered notifier structure can service devices that have high burst rates provided the bursts do not overflow the notifier's buffers.

---

7 A server may be slow to respond to a notifier if it has many send-blocked clients or if it performs substantial internal processing.

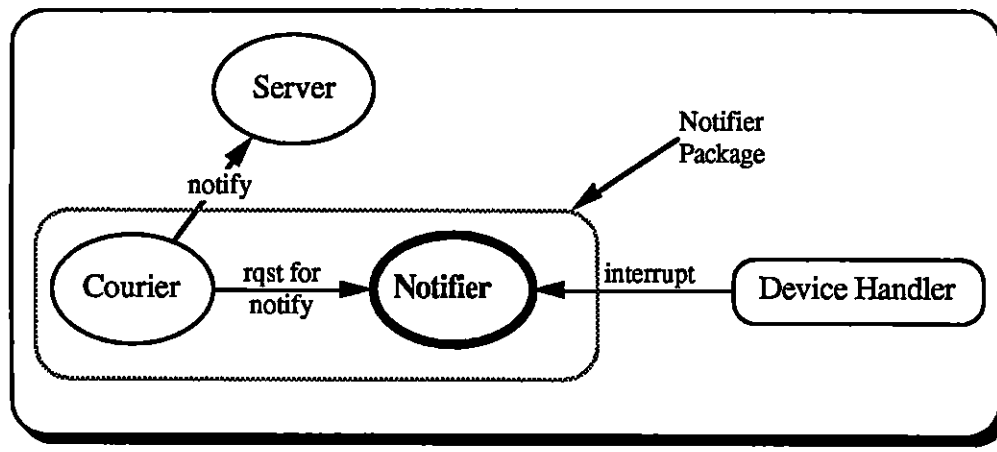


Figure 3.8 — Buffering Notifier Blocking Diagram: Unison Interrupt Mechanism

Harmony and Unison have different mechanisms for accepting interrupts. In Harmony interrupts are accepted via a special primitive, _Await_interrupt, which blocks the task until the specified interrupt occurs. In Unison, interrupts are accepted via the receive primitive. Although both the Unison and Harmony mechanisms for accepting interrupts can support buffered notifiers, the Harmony mechanism requires additional facilities so that the buffered events can be retrieved at any time. Because the _Await_interrupt primitive does not respond to messages, the courier would be blocked on the notifier until the next event occurs even if events are already buffered. To avoid this, either a shared buffer is required (undesirable) or a buffer administrator can be used. Figure 3.9 shows the notifier putting events into the event buffer when they occur and the courier retrieving them. When there are no events in the buffer, the courier remains send blocked on the buffer administrator. When the courier gets an event from the buffer, it sends the event to the server. Meanwhile, the notifier may be putting more events into the event buffer while the server is processing older events. This structure is less desirable than that of Figure 3.8

since it is more complex and involves more tasks and message transactions; however, this is not a serious problem in environments where communication overhead is low.

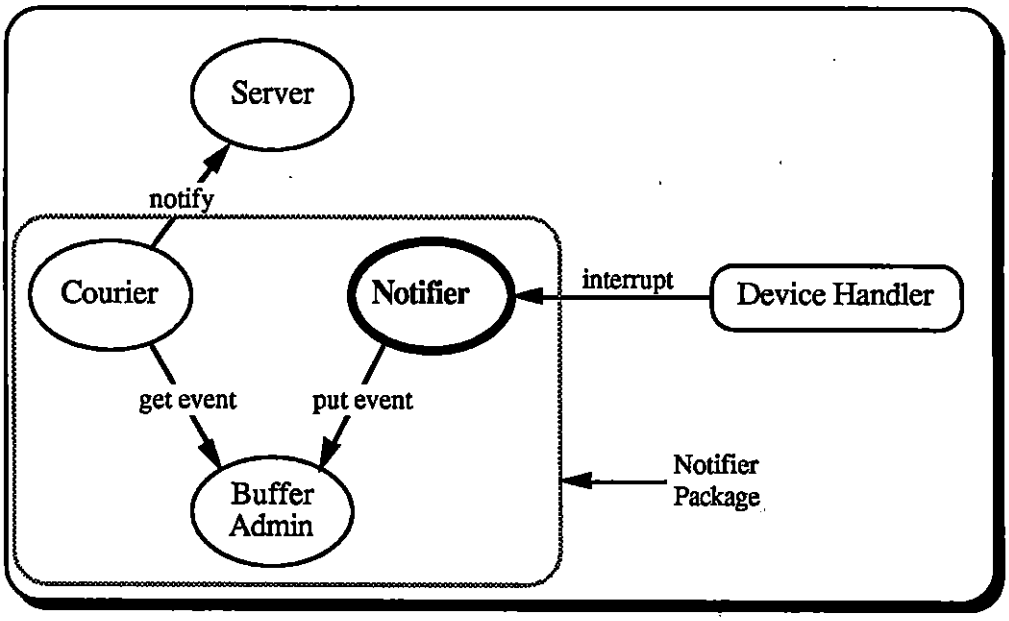


Figure 3.9 — Buffering Notifier Blocking Diagram: Harmony Interrupt Mechanism

Note that although the structure of the notifier is different for Harmony and Unison, the server is unaware of the internal structure of the notifier package. From the server's point of view, it merely gets events from a notifier.

Life Cycle

A notifier is created by its server during server initialization and usually exists as long as the server. It is sometimes necessary to destroy and re-create a notifier if the device temporarily fails or is otherwise placed in an indeterminate state.

When buffered notifiers are used, the server is only responsible for creating the

notifier. The notifier then creates the courier (and the buffer administrator for the Harmony implementation) during initialization. Since recursive task destruction is assumed, these sub-tasks are destroyed automatically when the notifier is destroyed.

Other Characteristics

Notifiers are generally small, simple tasks that execute very quickly and often at a high priority. They are blocked most of the time, awaiting some type of event, and do not consume much cpu time, as is desirable of high priority tasks. When an expected event occurs, a notifier performs minimal event processing and routes the event to whatever task is interested in the event, usually a server.

Code Template

Listing 3.5 describes a buffered notifier using the Unison interrupt mechanism. During initialization, the notifier creates the courier and sends it an initialization message which contains the task ID of the server. Once initialized, the notifier goes into a loop and receive-blocks. The notifier becomes unblocked by either a message from a task or a message from a device handler, an event. If the message is from the device handler, the message is either replied to the courier if the courier is ready or buffered if the courier is not ready. If the message is from the courier, the buffer is checked for events. If events are buffered, the oldest event is replied to the courier. If no events are buffered, the courier is marked as ready and remains reply-blocked on the notifier. The code template does not show any event filtering.

The difference for the Harmony implementation would be minimal. In this case, the

routine buffer_init() would create the buffer administrator. The routines buffer_put() and buffer_empty() would send messages to the buffer administrator. The data structure for the eventBuffer would likely contain the task ID of the buffer administrator.

Listing 3.5 — Buffered Notifier Code Template: Unison Interrupt Mechanism

```
TASK BufferedNotifier()
{
    EventBuffer *eventBuffer;
    NotifierRqst    rqst;
    NotifierRply    rply;
    TaskId      sender;
    TaskId      courier;

    create_courier();
    eventBuffer = buffer_init();
    initialize();           /* initialize the hardware */

    courier = NULL_ID; /* the courier is not reply-blocked */

    WHILE( TRUE )
        /*
         * receive a message from the courier or an interrupt
         * from an interrupt handler
         */
        sender = receive( &rqst, ANY_TASK );

        IF( sender == INTERRUPT_ID )
            /*
             * We have an event from an interrupt handler.  If
             * there is no courier ready, then buffer the.
             * event Otherwise reply the event to the
             * reply-blocked courier now.
             */
            IF( courier == NULL_ID )
                buffer_put( eventBuffer, rqst.event );
            ELSE
                rply.event = rqst.event;
                reply( &rply, courier );

                /*
                 * Mark the courier as being NOT blocked on us
                 */
                courier = NULL_ID;
            ENDIF

        /* continued on next page */
```

```
        ELSE
            /*
             * We have a message from a courier to process.
             * If there is no event ready just note that the
             * courier is blocked on us.  Otherwise give an
             * event to the courier.
             */
            IF( buffer_empty( eventBuffer ) )
                courier = sender;
            ELSE
                rply.event = buffer_get( eventBuffer );
                reply( &rply, sender );
            ENDIF
        ENDIF
    ENDWHILE
} .
```

3.3.6. Agents

Role

There are times when a task needs to communicate with another task, but must remain independent of the destination task. This phenomenon occurs if the communication protocol of the destination task is incompatible with the sending task, the communication protocol is likely to change, or the communication medium or destination task is unreliable. In these circumstances, an agent is employed as an intermediate task to isolate the sending task from its destination task. The sending task sends to the agent instead of the desired destination task and the agent sends the (possibly modified) message to the destination task. The agent performs error recovery and if necessary may even use non-standard communication links in order to transmit the message to the destination.

Variations

Agents are used in two different situations. When used for protocol conversion, the agent is called a *conversion agent.* When used to isolate the sender from either an unreliable communication medium (e.g. Unix datagram sockets [SUN86b]) or an unreliable destination task, the agent is called an *isolation agent.* An agent can also be a combination of these two types (an isoversion agent?).

Conversion agents are used when communicating tasks have different communication protocols or the communication protocols are likely to change. The agent is used as an intermediary to perform protocol conversion and help realize *separation of concerns* [PARN72] to reduce the amount of code affected by a change to the communication protocol.

Isolation agents are used when an unreliable communication link or an unreliable destination task must be used. In this case, the error recovery logic is localized in the isolation agent. Other tasks requiring the use of a failure prone link send to the agent which in turn sends over the link. The agent takes appropriate measures to ensure that the message is successfully communicated or, at the very least, that the failure is detected. The example implementation described in Chapter 6 uses this approach when Unison tasks are communicating with Unix tasks via unreliable UDP datagram sockets.

Agents are a more general form of the Ada *relay* tasks [NIEL87]. Relay tasks accept data by being *called* and deliver data by *calling* another task. Relay tasks are used merely as single message buffers to prevent the caller from blocking if the called task is not ready to accept the message.

Communication Semantics

An agent receives requests from arbitrary, unknown sending tasks (considered clients of the agent) and sends the message to a single destination task, often a server. The agent registers its symbolic name with the Name Server so clients can determine its task ID. When the destination task replies, the agent replies to the client. Because of this simple structure, the agent can only process one message at a time.[8]

Figure 3.10 illustrates the communication semantics of a conversion agent. The client sends a destination-independent message to the agent. The agent performs the protocol conversion and sends the destination-specific message to the destination task. When the destination replies to the agent, the agent will reply to the client.

---

8 If it is necessary to have multiple pending messages, either workers need to be employed or the more versatile structure of the server is necessary.

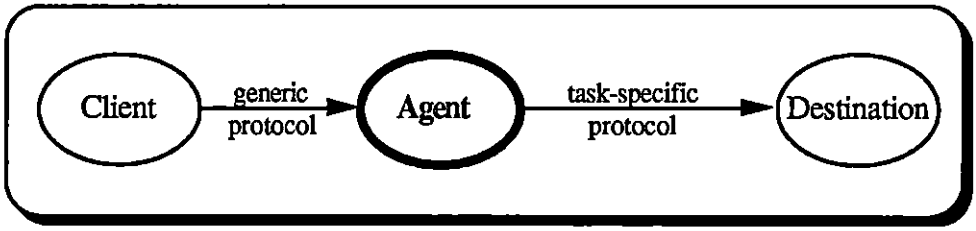


Figure 3.10 — Conversion Agent Blocking Diagram

Figure 3.11 illustrates the blocking semantics of an isolation agent. In this diagram, the agent receives a message from a client which needs to ultimately communicate with the destination task using an unreliable communication link. The agent informs the watchdog that an unreliable operation is about to begin then sends the message to the destination task. If the destination task does not reply to the agent within a limited period of time, the watchdog assumes the message transaction has failed and terminates the agent. The vulture then re-creates the agent and the client is informed by the operating system that the send to the agent failed and the client may try the operation again. If the destination *does* reply to the agent within the time limit, then the agent informs the watchdog that the operation succeeded and replies to the client.

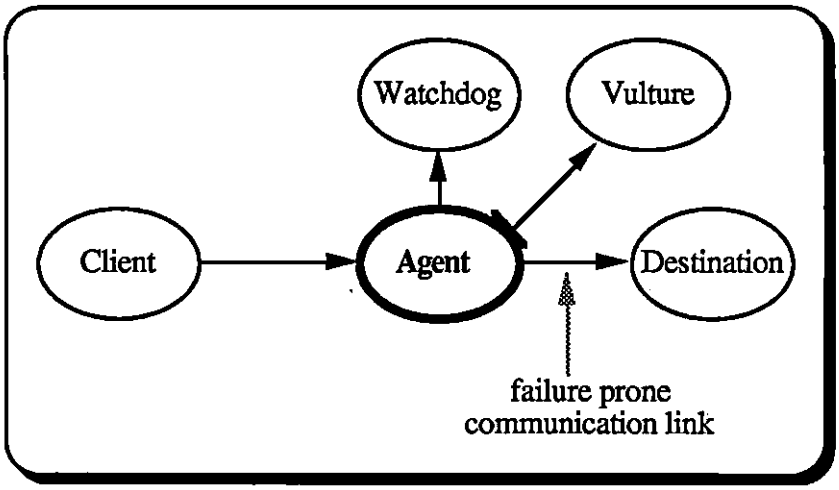


Figure 3.11 — Isolation Agent Blocking Diagram

Life Cycle

Agents are generally created by the creator of the destination task or by some application configuration task. Individual agents have a potentially short lifespan, however, the facilities of the agent usually exist for the duration of the application as generations of re-created agents.

Other Characteristics

The response time of the agent is determined by two factors. First, the response of the destination task directly affects how quickly the agent responds to the client. Second, since the agent only processes one request at a time, the number of requesting tasks directly affects the response for a particular client. If this is unacceptable, it is possible to use a number of workers, each performing the conversion or providing isolation for each client request. In practice, the destination task, not the agent, is usually the bottleneck.

Code Template

Listing 3.6 describes a combination isolation/conversion agent. The agent performs initialization, which includes the creation of the watchdog, then begins its loop. It receive-blocks until a message arrives from a client. The destination-independent request message is then converted to a destination-specific message. Before the message is sent via the unreliable communication link, the watchdog timer is started. The message is then sent to the destination task, a server. Should the server take too long or the communication link fail, the watchdog timer will time out and destroy the agent thereby causing the message transaction from the client to the agent to fail. The agent's vulture re-creates the agent and the client can re-try the operation as soon as the new agent has registered with the name server. If the server replies to the agent *before* the timer expires, the watchdog timer is stopped, the reply message is converted and replied to the client.

There is no mention of the vulture in Listing 3.6. The creator of the agent is responsible for the creation of the vulture. The agent is completely unaware of the vulture (as it circles overhead). When the agent is destroyed, the watchdog is recursively destroyed, leaving no unused tasks consuming system resources.

Listing 3.6 — Combination Isolation/Conversion Agent Code Template

```
TASK IsolationAgent()
{
    ClientRqst  clientRqst;
    ClientRply  clientRply;
    ServerRqst  serverRqst;
    ServerRply  serverRply;
    TaskId      watchDogId;
    TaskId      serverId;
    TaskId      requestor;

    /*
     * Register name as part of initialization so other tasks
     * can make use of this task.
     * Create our watchdog task.
     * Find the ID of the server we are using
     */
    initialization( &watchDogId, &serverId );

    WHILE( TRUE )
        requestor = receive( &clientRqst, ANY_TASK );

        /*
         * A client wants to send to the server using the
         * failure prone communication link.
         * Start the watchdog timer.
         * Send using the failure prone communication link.
         * Stop the watchdog timer.
         */
        WatchDogStart( watchDogId );

        serverRqst = ConvertRequest( clientRqst );
        send( &serverRqst, &serverRply, serverId );

        WatchDogStop( watchDogId );

        clientRply = ConvertReply( serverRply );
        reply( &clientRply, requestor );
    ENDWHILE
}
```

3.3.7. Vultures

Role

Servers and agents can be considered system resources, available to all tasks. It is important that these tasks continue to exist for the duration of the application. In the event that a serious or unrecoverable error occurs within one of these tasks, it may be necessary to terminate the task. It is necessary for the system configuration task to be able to detect the termination of the task and to re-create it. A vulture is a simple task whose sole purpose is to detect the termination of a particular task, a victim, and take some action when the victim terminates.

Variations

The action taken by a vulture upon detection of victim termination depends on the specific purpose for which the vulture was created. Vultures are commonly used to notify a configuration task that an important task has terminated due to some error condition. Sometimes the vulture itself will re-create the terminated task. In this case, the vulture is more like an angel, since it is keeping the task alive.

Less commonly, if the operating system does not provide a mechanism for servers to detect client termination, vultures can be attached to clients while the client has an open connection. If the client terminates without closing the connection, then the vulture will notify the server. If the client closes the connection, the server destroys the vulture. While this mechanism provides the same functionality as that provided by some operating systems,[9] it can consume considerable resources if a server has many open connections.

---

9 Harmony supplies connection-based servers with a CLOSE message when any of its clients are destroyed.

Communication Semantics

Vultures communicate once with their creator at initialization time, so they can receive the task ID of the victim, and possibly once again when the victim is destroyed. The second message may be omitted if the vulture re-creates the victim, hence it need not notify its creator. If the vulture does not re-create the task, it must notify its parent, indicating that the monitored task has terminated. Since vultures only communicate with their parent, they do not register their name with the Name Server.

Vultures can be considered a special form of notifier where the event detected is the destruction of a task. They are commonly associated with agents and other tasks that are making use of watchdogs (described in Section 3.3.8).

The exact message semantics of the vulture task depend upon the specifics of the communication primitives, however they are conceptually similar. The vulture must issue some blocking primitive that will never be satisfied by the potential victim. In Harmony, where a task can receive specifically from another task, the vulture can issue the receive primitive. The victim never sends to the vulture. Only when the victim terminates will the vulture become unblocked and is thus notified of the termination of the victim.

In Unison, message addressing is via ports, preventing the vulture from receiving specifically from the victim. In this case, the vulture must execute a send primitive using a port on which the victim will never receive. This mechanism is somewhat less desirable since it is difficult to guarantee the victim never executes a 'receive-on-any-port'. Receive-on-any-port may be executed inside a library routine, unbeknownst to the victim, possibly disrupting the communication semantics assumed by the vulture. The Harmony

mechanism is safer for two reasons. The vulture does not send to the victim so the victim cannot inadvertently receive from the vulture. The victim cannot send to the vulture because it is never given the vulture's ID. Figure 3.12 illustrates the difference between the Harmony and Unison vulture blocking semantics.

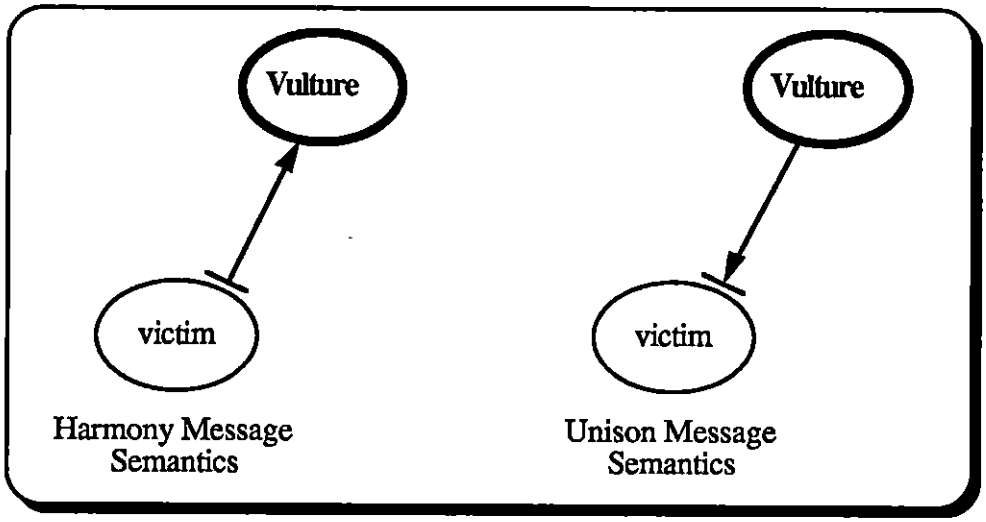


Figure 3.12 — Vulture Blocking Diagram

Life Cycle

Vultures are created by the creator of the victim task and last as long as the victim is required by the creator. Vultures are explicitly destroyed by their creator when they are no longer required.

Other Characteristics

Vultures are simple tasks. They can be programmed as generic tasks and do not usually need modification for individual applications. Vultures are often fairly high priority

since it is usually desirable to re-create the victim as soon as possible.

Code Template

Listing 3.7 describes a generic vulture task. The first time a vulture sends to its parent, it merely informs the parent that it is ready to start monitoring a task. The parent replies, specifying the task to be monitored (the victim). The vulture then executes a receive-specific on the victim. When the victim terminates, the receive-specific primitive will fail, hence the vulture becomes unblocked. The vulture now reports this event to its creator and the cycle repeats.

Note that in Listing 3.7 there is a message between the vulture and its creator that is not shown in the blocking diagram. This message occurs during initialization of the vulture and is not part of the active blocking characteristics.

Listing 3.7 — Vulture Code Template: Harmony Receive Semantics

```
TASK Vulture()
{
    VultureRqst rqst;
    VultureRply rply;

    /*
     * No task termination detected on the first message.
     */
    rqst.VICTIM_ID = NULL_ID;

    WHILE( TRUE )
        /*
         * Send to our parent to find out what task we are to
         * watch over.
         */
        rqst.MSG_TYPE = READY_TO_VULTCH;
        send( &rqst, &rply, ParentId() );

        /*
         * Watch over the task now
         */
        IF( receive( &null_msg, rply.VICTIM_ID ) != NULL_ID )
            /*
             * ERROR!  The victim should not even know our ID
             * and certainly should not have sent to us.
             */
        ELSE
            /*
             * The monitored task has terminated.  Let the
             * parent know the victim's ID in the next
             * message.
             */
            rqst.VICTIM_ID = rply.VICTIM_ID;
        ENDIF
    ENDWHILE
}
```

3.3.8. Watchdogs

Role

There are times when a task is forced to perform unreliable operations or operations that may take too long to execute. In particular, some server operations may only be considered successful if they complete within a certain period of time. In these situations it is convenient to use a watchdog to time unreliable operations and terminate the task if the operation exceeds the allotted time.

Variations

Watchdogs are used when 1) the recipient of a message is unreliable, possibly leaving the sender blocked forever, 2) the communication link between the sender and receiver is unreliable, or 3) the operation performed by the receiver is not a constant time operation and may, under non-deterministic conditions, take longer than the sending task is willing to wait.

Communication Semantics

From the point of view of the task using the watchdog, (the watchee), the message semantics are very simple. The watchee sends a start-watching message to the watchdog, informing it that a timed operation is about to be performed. The watchdog receives this message and immediately replies. Under normal conditions, the watchee will complete the operation in time and send a stop-watching message to the watchdog as shown in Figure 3.13. The watchdog will stop any internal timers and reply to the watchee. If the watchee operation takes too long, the watchdog will timeout and either destroy the watchee or

inform some task of the overdue operation.

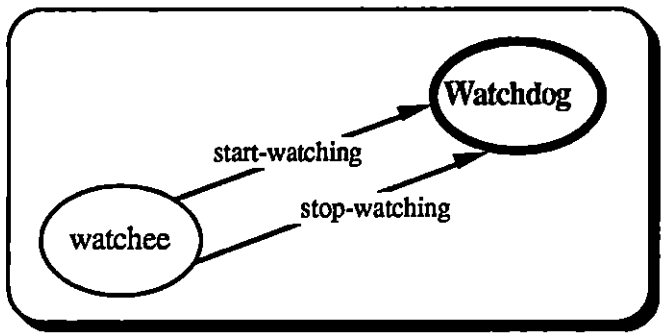


Figure 3.13 — Watchdog Blocking Diagram: Normal Operation

The implementation of the watchdog depends upon the details of the message passing primitives. If the receive primitive has a timeout facility as in Unison, watchdogs are quite simple. In this case, when the watchdog executes the second receive primitive, it specifies a maximum timeout. If the stop-watching message does not arrive before the timeout expires, the watchdog has detected an overdue operation and takes appropriate action. Figure 3.14 illustrates a Unison watchdog.

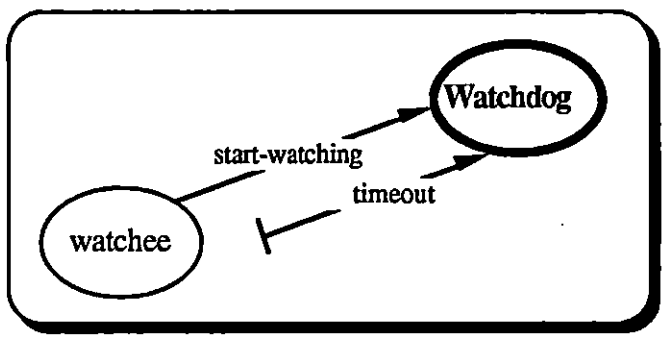


Figure 3.14 — Watchdog Blocking Diagram: Unison Timeout

If timeouts are not available on the message primitives as in Harmony, a slightly more complex structure is necessary. After the first message transaction with the watchee, the watchdog starts a timer. The timer is required to send a message to the watchdog when the timeout expires. The watchdog now executes a non-specific receive, so it will receive either a stop-watching message from the watchee or a timeout message from the timer. If the watchee message arrives first, the timer is cancelled. If the timeout message is received first, the watchee is terminated. Figure 3.15 illustrates a Harmony watchdog.

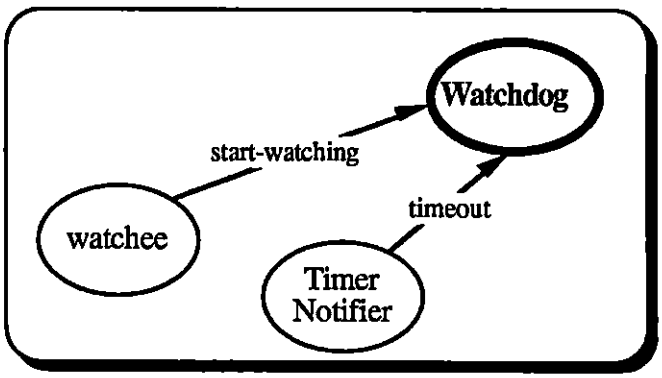


Figure 3.15 — Watchdog Blocking Diagram: Harmony Timeout

Watchdogs are often associated with agents, especially when agents are using unreliable communication mechanisms or other failure prone operations.

Life Cycle

Watchdogs are usually created by the task that needs to perform monitored operations, either at initialization time if the operation is used throughout the execution of the task or prior to performing the operation if the operation is only executed occasionally.

They are explicitly destroyed when the task no longer requires them or when the creator is terminated. In environments where task creation overhead is high, watchdogs can be maintained in a pool and allocated as necessary.

Other Characteristics

Watchdogs are simple tasks and should respond quickly to start and stop messages so that the timeout period is not affected by latency. The overhead of messages should be considerably less than the timeout period unless the message overhead is factored into the timeout period.

Code Template

The watchdog described in Listing 3.8 uses the Unison **receive** primitive which includes a timeout facility. Inside the main loop, the watchdog receive-blocks, awaiting a START_WATCHING message from the client. When the message arrives, the watchdog immediately replies to the client then receive-blocks again, however, this time, the timeout is specified by the START_WATCHING message. If the client does *not* send the STOP_WATCHING message in time, the receive primitive indicates the timeout has expired, and the watchdog destroys the client. If the client sends the STOP_WATCHING message before the timeout expires, the watchdog immediately replies to the client. The entire cycle then repeats.

Listing 3.8 — Watchdog Code Template: Unison Message Semantics

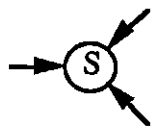
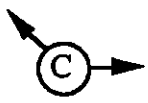
```
TASK Watchdog()
{
    WatchRqst   rqst;
    WatchRply   rply;
    TaskId      requestor1;
    TaskId      requestor2;

    WHILE( TRUE )
        /*
         * Wait until the client tells us to start watching.
         */
        requestor1 = receive( &rqst, FOREVER );

        IF( requestor1 != ParentId() )
            /* error: illegal requestor */
        ENDIF

        /*
         * Reply immediately, then start watching
         */
        rply.RESULT = OK;
        reply( &rply, requestor1 );

        IF( rqst.MSG_TYPE != START_WATCHING )
            /* error! */
        ENDIF

        /*
         * Now wait for either:
         *   • the client to tell us to stop watching
         *   • the timeout to expire
         * Note that the client tells us how long to watch.
         */
        requestor2 = receive( &rqst, rqst.TIMEOUT );
        IF( requestor2 == NULL_ID )
            /*
             * The receive failed, hence it timed out.
             * Note: it did not fail due to termination since
             * if the parent terminated we would terminate.
             */
            destroy( requestor1 );

        /* continued on next page */
```

```
        ELSEIF( requestor1 == requestor2 )
            /*
             * No need to destroy the client this time.
             */
            rply.RESULT = OK;
            reply( &rply, requestor2 );
        ELSE
            /* error: illegal requestor task */
        ENDIF
    ENDWHILE
}
```

3.4. Summary

This chapter has provided an overview of a programming paradigm based on the S-R-R primitives and a detailed description of each of the task types that make up the paradigm. For convenience, Table 3.1 summarizes the essential properties of each task type.

Table 3.1 — Essential Properties of Component Task Types

Property / Task Type	Purpose	Correspondents	Communication Semantics	Lifespan	Scope
Client	application processing using servers etc	servers and agents		any	private
Server	perform services on behalf of clients	clients, workers, courier and notifiers		long	global via name service
Worker	perform work on behalf of its server	parent server, other servers and agents		duration of its associated client connection	known to parent and sometimes a client
Courier	prevent server from send-blocking	parent server, notifiers and other servers		short	known to parent
Notifier	await some event and notify the server when it occurs	parent server		duration of its server (long)	known to parent
Agent	pass a message from any task to a particular server	clients and specific servers		long	global via name service
Vulture	monitor a task for death, notify its parent or re-create task	parent		duration of its parent or monitored task	known to parent
Watchdog	time client activity and destroy client if too long	parent		short	known to parent

PART II

AN EXAMPLE APPLICATION

Chapter 4

The KAON Factory Control System

4.1. Introduction

TRIUMF is Canada's national meson facility and the world's largest isosynchronous cyclotron. To further its research into the high intensity and precision frontiers,[1] TRIUMF has proposed the development of a KAON[2] Factory [TRIU85, SCHA86]. The proposed KAON Factory would be a synchrotron producing a 30 GeV proton beam which is two orders of magnitude more intense than that produced by existing accelerators at similar energy levels. The proposed computer control system performs beam diagnostics, ensures safe beam operation, and *tunes* the beam for optimal performance. This application is an good candidate for illustrating the use of the S-R-R programming paradigm in a realistic situation.

---

1 As opposed to the high-energy frontier.

2 KAON is an acronym for Kaons, Antiprotons, Other hadrons, and Neutrinos. A kaon is a particle that exists at a high energy level and is made up of a *strange* quark and a *strange* antiquark.

4.2. The KAON Factory

The existing TRIUMF cyclotron is used to inject a 100 microamp proton beam into the KAON Factory which accelerates the beam to 30 GeV through a chain of five fast-cycling synchrotrons and DC storage rings. The 450 Mev H^- ions from the cyclotron are stripped to protons and injected into the first ring, the Accumulator, which accumulates the beam over 20 millisecond periods. The second ring, a 50 Hz Booster synchrotron, then accelerates the proton pulse to 3 GeV. The third ring, the Collector, collects five Booster pulse trains and manipulates beam longitudinal emittance. The beam is then passed to the fourth ring, a 10 Hz Driver synchrotron, which accelerates it to 30 GeV. Finally, the 30 GeV beam is collected and stored in the fifth ring, the DC Extender, for slow extraction. The first two rings are approximately 68 metres in diameter and the final three rings are each 341 metres in diameter.

The KAON Factory control system provides control and monitoring of various synchrotron activities. A central control room provides a number of consoles with which operators can observe the synchrotron's current operating state and modify control parameters. The control computers (called real-time nodes), distinct from the consoles, are distributed around the synchrotron's rings, each controlling nearby devices. All consoles and real-time nodes are connected by a high-speed communication network. Figure 4.1 depicts the configuration of the control system.

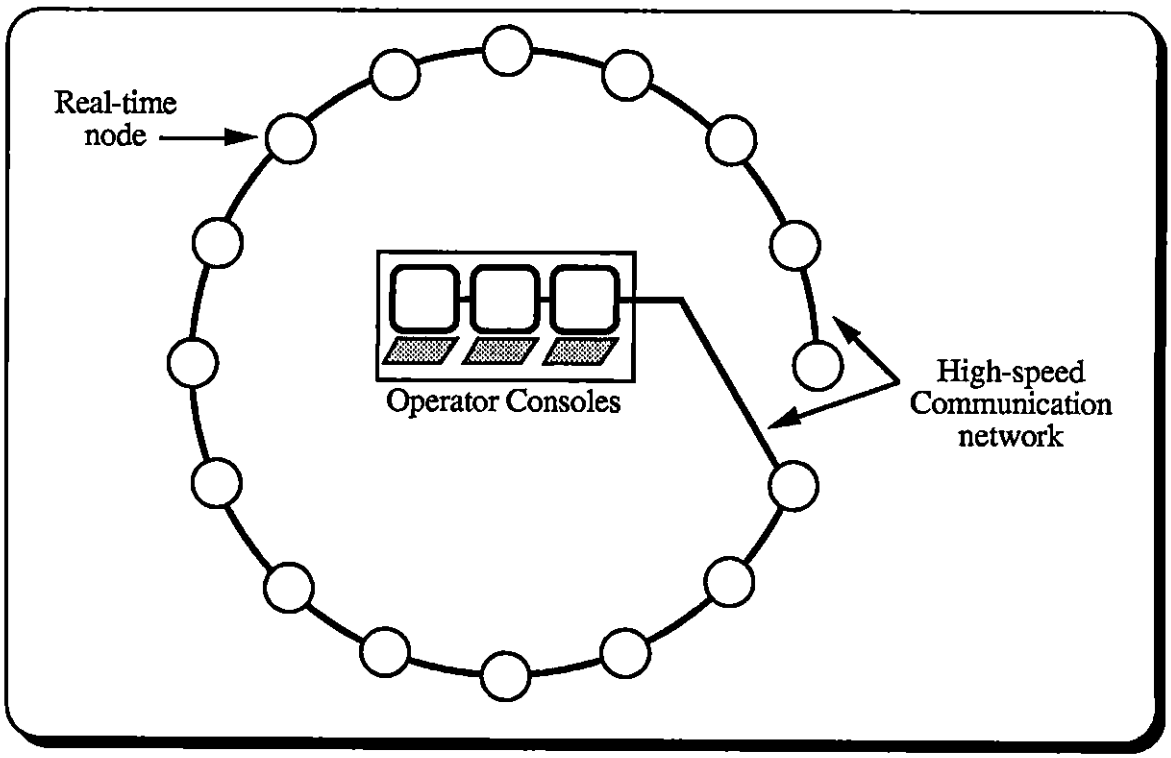


Figure 4.1 — The KAON Factory Control System

Data acquisition and control of the KAON Factory occurs at two distinct time scales. The time scale of the 42 radio frequency (RF) stations, used for beam acceleration, is unsuitable for software control since sub-nanosecond precision is required. The time scale of the 1032 dipole and quadrapole magnets, used for beam deflection, alignment, and focussing, can be software controlled since magnet response is on the order of one-half second. A site-wide external timing mechanism is used to trigger acquisition of data, such as vertical and horizontal beam position, from sensors located in the focussing magnets. Local timing mechanisms can also be used to trigger other activities.

4.3. Software Requirements

The preliminary[3] software requirements of the KAON Factory control system are dictated by the extent of the system which must be controlled. Since the KAON Factory control system is intended to be used for more than 20 years, a long term approach to its design and implementation must be taken. Current software engineering principles must be exploited to ensure long term maintainability of the software. Modular software and hardware is necessary so that functionality can be added for future experiments and improvements to the synchrotron. Standards must be adopted whenever appropriate for future compatibility.

The software for the KAON Factory control system consists of two components, the user interface (UI) and the acquisition and control system (ACS). The UI displays operational and alarm data and allows operators to modify operational parameters. The ACS monitors and controls physical activities throughout the KAON Factory.

4.3.1. User Interface[4]

The UI must respond in human-time (real-time with respect to human reaction time) and provide consistent response. Alarm and status indicators must reflect the actual state of the system within about 0.25 seconds [TRUI88] since the operator may be required to take action immediately. Trend data must be collected and displayed at a rate of about 2 Hz.

---

3 Detailed requirements of the KAON Factory are not yet available.

4 The UI requirements described here are *not* the result of a detailed study but are merely the thoughts and opinions of a few people connected with the project.

The interaction techniques used by the UI should be based on the windows, icons, mice, and pop-up menus (WIMP) interface that has been proven effective by Macintosh™ computers. Interactive devices, such as sliders, buttons, and menus, should be used to modify various ACS parameters. Windows should be used to display alarm, status, and trend data.

4.3.2. Acquisition and Control System

The ACS consists of a number of control computers, physically close to the function generators, power supplies and analog-to-digital converters that are controlled or monitored. Each control computer is a real-time processing node connected via a high-speed communication link to other controllers and to the operator consoles.

The entire system must respond to at least 6,100 devices, consisting of 12,800 analogue I/O channels and 24,300 digital channels. This number is an irreducible minimum which will grow as design of the KAON Factory is refined. The large number of devices gives rise to two design issues. The first is the need for distributed intelligence to control the physically separate devices distributed around the synchrotron. The second is that each real-time node requires a multiprocessor in which each processor handles a subset of the devices assigned to the node. The advantage of this approach is that, as more devices are added, more processors or real-time nodes can be added at incremental cost without affecting either code or response time of the existing system.

Many aspects of the ACS require real-time response. In particular, reaction to alarm conditions must occur within precise time limits for safety reasons. Reaction to different types of alarms must be intelligent. Potentially dangerous alarm conditions must shutdown beam generation and attenuate the beam as quickly as possible while other alarm conditions

only require the attention of an operator.

Since the purpose of the ACS is to achieve optimal control of the synchrotron, it is necessary to dynamically acquire operational data. The data acquisition software must be closely coupled with the control software so that devices can be continually monitored and adjusted. Some acquired data will be sampled according to a site-wide timing signal. Since the actual acquisition is too fast for software the data must be strobed into hardware latches by the timing signal. The control software subsequently performs data collection, time-stamping, processing, logging, and display at an operator's console. Each measurement lasts from a few microseconds to an entire acceleration cycle. Setup, collection, and display should add no more than about one-half a second to the data acquisition time since each controlled magnet requires a response time of about one-half second.

4.4. Summary

The size and scope of the KAON Factory control system has implications on many design decisions. Because there are many asynchronous activities (i.e. control of independent magnets) a multitasking system is necessary. The physically separate stations require physically separate control computers, making distributed control computers necessary. The expanding number of I/O channels requires that each control computer has multiple processors. Data acquisition and display require real-time response. Real-time response is also required of the ACS to ensure that safety features can be implemented effectively. To ensure maintainability and expandability, modern software engineering approaches to software design must be applied. Finally, such a large system requires a sophisticated development system on which software can be tested and debugged.

The next chapter describes a possible development system and runtime environment for the example implementation of the KAON Factory control system. Application of the programming paradigm to the design and implementation of the control system is the topic of Chapter 6.

Chapter 5

The Development and Runtime Environments

5.1. Introduction

The development and runtime environments consist of a host machine connected via Ethernet to a target machine. The host machine is a Unix based workstation for the development of target-based control software. The host machine is also used as a user interface device controlled by the target software during runtime. The target machine executes a real-time multiprocessor operating system and the application to monitor and control the physical activities of the KAON Factory.

5.2. Hardware

The hardware consists of two distinct systems connected via coaxial Ethernet cables. All development activities are performed by the host hardware system, while runtime activities are performed by both the target and host hardware.

5.2.1. Host Hardware

The host is a Sun 3/52 workstation. It supports the software necessary for software

development and testing as well as the software for the KAON Factory user interface. The user interface is only active during execution of the target software.

The Sun 3/52 workstation has a MC68020 central processor, 4 megabytes of memory, a local 70 megabyte hard disk drive, a high-resolution monochrome bitmapped display, a mouse, and an Ethernet interface. The graphical interface was used during the development phase, in particular by the debuggers, and during the execution phase by the KAON Factory user interface. The Ethernet interface provides access to other campus computers, in particular to various file servers and also provides the main communication link to the target system.

5.2.2. Target Hardware

The target hardware is the runtime hardware for the real-time multiprocessor operating system and the application software. A single box houses a backplane, two general purpose processor boards, an intelligent Ethernet board and a power supply. An external Ethernet transceiver is necessary to connect the Ethernet board to the campus Ethernet. The processor and input/output (I/O) boards are connected by a high-speed parallel backplane. The two processor boards provide most of the computational power. The intelligent Ethernet board provides host to target communication facilities. These components are shown in Figure 5.1 and described below. This hardware sub-system is called the Real-Time Node since this processing complex is capable of supporting real-time activities. In the KAON Factory, many of these nodes would be distributed around the accelerator.

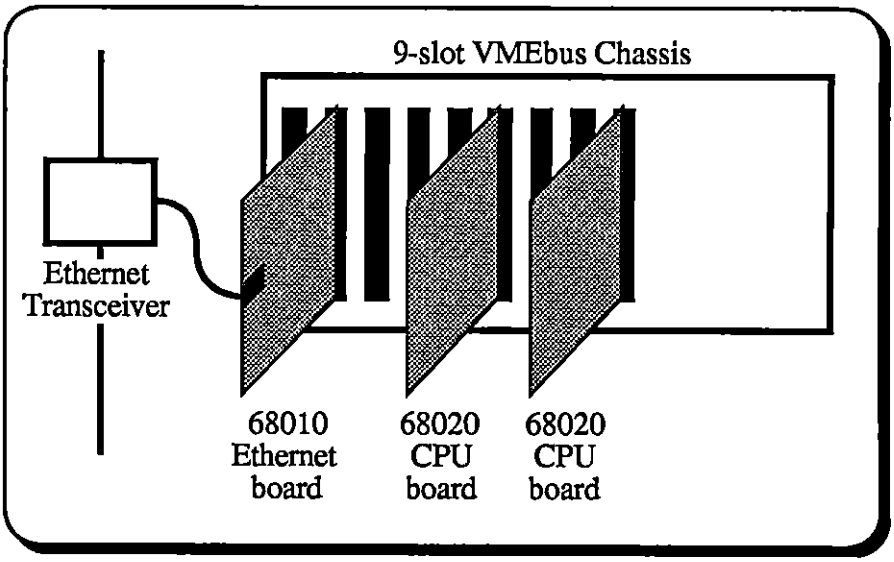


Figure 5.1 — Target Hardware System

A nine-slot VMEbus [MOTO85a] backplane was purchased for this project, however larger backplanes (i.e. more slots) are available. This bus was chosen by TRIUMF since it is the bus most likely to be used in the actual control system. The VMEbus is very well supported by a large user and vendor community so there is a large selection of VMEbus devices readily available. It is an asynchronous bus with 32-bit address and data paths. A discussion of computer bus issues can be found in [GUST84, DAWS86, BAKE87b]. A comparison of various standard computer buses (and many of the reasons why the VMEbus was chosen) can be found in [BORI85, BAKE88a].

Two DY-4 DVME-134 processor boards [DVME87] are the main computing engine of the target system. These boards have a number of features, some of which specifically support multiprocessing. The CPU on each board is a 12.5 MHz MC68020 [MOTO85b]. This is a powerful 32-bit, complex instruction set architecture processor often used in

current generation workstations. Each board has one megabyte of dual-ported memory[1]. This memory can be accessed very quickly by the local processor and also by any other device connected to the VMEbus, but at a slightly reduced rate due to the overhead of using the bus. Processor-specific interrupts can be generated by writing data into a location monitor.[2] The boards each have a single RS-232 serial port for terminal input and output which is used mainly for error and diagnostic output.

One DY-4 DVME-750 Ethernet controller board [DVME86] provides the high speed communication link to the host system and other networked computers. This board has a 10 MHz MC68010, 512 kilobytes of dual-ported memory, and a Local Area Network Controller for Ethernet (LANCE chip). The local MC68010 processor is responsible for servicing all Ethernet interrupts and for controlling the LANCE chip. The existence of a processor on the Ethernet board eliminates VMEbus incurred interrupt and data access delays that would occur if a non-local processor were to service a non-intelligent Ethernet controller. While this processor is primarily used to handle Ethernet communications, it is also a fully functional general purpose processor and can be used as a processing resource. As with the DVME-134 board the DVME-750 board has a location monitor. An Ethernet transceiver is necessary to convert the signals generated by the Ethernet board to those acceptable on a thin or thick Ethernet [DAWS87] cable.

The use of processor boards (including the intelligent Ethernet board) that have local dual-ported memory as opposed to all processors sharing one or more global memories can significantly reduce VMEbus traffic. This is because most memory accesses are local and,

---

1 Dual-ported memory is memory that can be accessed both by the local processor and by other devices connected to the bus.

2 A location monitor is a dual-ported memory location that, when written to, generates an interrupt on the local processor.

consequently, do not use the VMEbus. The current version of Unison supports up to sixteen processors, however, depending upon the actual application, VMEbus saturation may not occur until many more processors are present. Some Harmony derived systems using VMEbus have employed more than thirty processors [VERT86]. Multiple VME buses can be connected via Ethernet, however the links between chassis are not real-time.

5.3. Software

A number of commercial software packages were used in the development of the KAON Factory application software and for the runtime environment. The development environment consists of BSD 4.2 Unix [BSD84], a C compiler, and a number of tools to aid development and debugging of real-time software. The runtime environment consists of Unison, the real-time, multiprocessing kernel supporting the application software and the NeWS window system [SUN87a] supporting the host-based user interface software. The programming paradigm is based on the task communication primitives of Unison [MTI87].

5.3.1. Development Software

Development software is the software that is used during any phase of development, including simulation and debugging. All development was performed on a Sun Workstation™ running BSD 4.2 Unix. This operating system provides many of the tools necessary for software development. While Unix is a multitasking operating system, it is not a real-time system and for TRIUMF's purposes, is only suitable as the development operating system and for hosting user interface software. A separate real-time operating

system is required to support real-time process control software.

Many standard Unix utilities, such as *cc, make,* and *dbx* were used. The Unix C compiler and its associated tools[3] were used for cross development. Since the Sun C compiler supports compilation for the MC68010 and the MC68020, this compiler was suitable for both host and target software compilation. The Unix *make* program, along with some special utilities,[4] was used to ensure all program modules are up to date, minimizing the amount of re-compilation required after a source code change. Since this project involved only one version and only one programmer, no revision control system was employed. In a commercial setting, *SCCS* [ROTH75], *RCS* [TICH82], or some other revision control system would be necessary. Some special Unix software was also used to simplify the cross development process. For this purpose Multiprocessor Toolsmiths Inc. (MTI) provided *CopyCat,* a host-based Unison simulator, *Remedy,* a multiprocessor debugger, *hcc,* a special C compiler front end, and *edl,* an Ethernet downloader.

C was chosen as the programming language for a number of reasons. First, C is the language currently supported by the MTI development environment. C is well supported in the Unix environment. C is generally sufficient as a systems programming language, and it is rarely necessary to resort to assembly language. Messages, communicated between tasks, are often polymorphic, hence C's ability to cast types is an advantage. Ada was not considered since an Ada compiler was not available and the target software system does not yet support Ada.

The simulator, *CopyCat,* is a host-based software package that runs on a Unix

---

3 A linker and assembler are required with the C compiler.

4 The program *mkinc* is described in Appendix D.

workstation and simulates the Unison multi-tasking runtime environment. Execution under host debuggers is supported, in particular, *dbx* and *dbxtool* [SUN86a] can be used to debug application programs running under CopyCat. The simulator supports Unison system level event tracing so interaction between tasks can be carefully monitored. CopyCat also supports host-based interrupt simulation and device modeling which permits application software to execute under CopyCat. Unix and Unison terminal I/O facilities are also provided. CopyCat only provides uniprocessor simulation; multiprocessor configurations must be re-configured to run under CopyCat. The advantage of using CopyCat rather than Remedy (discussed below) is the increased control provided. The disadvantage is that real-time goals cannot be achieved, and hence timing related bugs are difficult or impossible to expose when running under CopyCat.

Remedy is the real-time multitasking multiprocessor debugger. It is different from most other debuggers in that it uses interactive graphical displays to aid debugging real-time multiprocessor applications at the source level. During debugging, the application program executes on the target hardware along with minimal target resident debug support. The debugger itself executes on the host hardware, making use of interactive graphical devices. The source-level debugger has features similar to *dbx* for displaying task code, manipulation of task variables, and setting breakpoints. Source-level interaction can be focused on any instance of any task.

Other support tools were supplied by MTI. Two *downloaders* were provided for transferring executables from the host to the target. One downloader, *dl*, uses a serial RS-232 link and the other downloader, *edl*, uses Ethernet as the communication medium. The latter mechanism is considerably faster, typical downloads taking from one to two seconds

per processor. A compiler front-end, *hcc*, provides a simplified approach to using standard Unix C compilers as cross compilers, producing target code images. This front-end links the appropriate Unison libraries and informs the linker of the correct addresses for the target code.

5.3.2. Runtime Software

Runtime software is the software used during runtime of the application. All application software other than the user interface software executes on the target hardware. The user interface software executes on the host hardware. The Unison operating system provides support for real-time target-based application software. The Unix operating system and NeWS provides support for the user interface software. Pertinent details of Unison, Unix and NeWS are described below.

5.3.2.1. The Unison Kernel

History

Thoth [CHER79] was one of the first operating systems to use the Send-Receive-Reply task communication primitives. A few years later Port [MALC81] at the University of Waterloo and Harmony [GENT83] at the National Research Council (NRC) appeared, both of which use the Send-Receive-Reply primitives. Port is a commercial system that makes use of inexpensive networked microcomputers while Harmony was targeted for embedded real-time applications. A significant difference between Harmony and the others was its transparent support for multiprocessors. Harmony is a simple, elegant operating system with a small set of well-defined primitives. QNX [QUAN88] is another

commercial derivative of Thoth, providing real-time support in a networked PC environment. Cheriton followed Thoth with Verex at the University of British Columbia. This was followed by the V-Kernel [CHER84] which currently provides a research environment at Stanford University.

A few attempts have been made to commercialize NRC Harmony to make it more accessible to industry. A modified version of Harmony has been used by Vertigo [VERT86] in a multiprocessor graphics workstation to provide high-speed production quality rendering. This version allowed an arbitrary number of processors to co-exist in a single system. Recent systems with up to 48 processors have been built. P-Can [TANN87] of Toronto is currently providing commercial support for NRC Harmony. The University of Carleton's Advanced Real-Time Toolset (ARTT) [ROWE86] project used NRC Harmony as the core operating system and built upon it a graphical debugger and a Unix based simulator. Unison evolved from the ARTT research and the commercial efforts by MTI. MTI has tailored Unison to make it more suitable as the multi-tasking kernel for Ada programs.

The complete family tree of Send-Receive-Reply operating systems is illustrated in Figure 5.2.

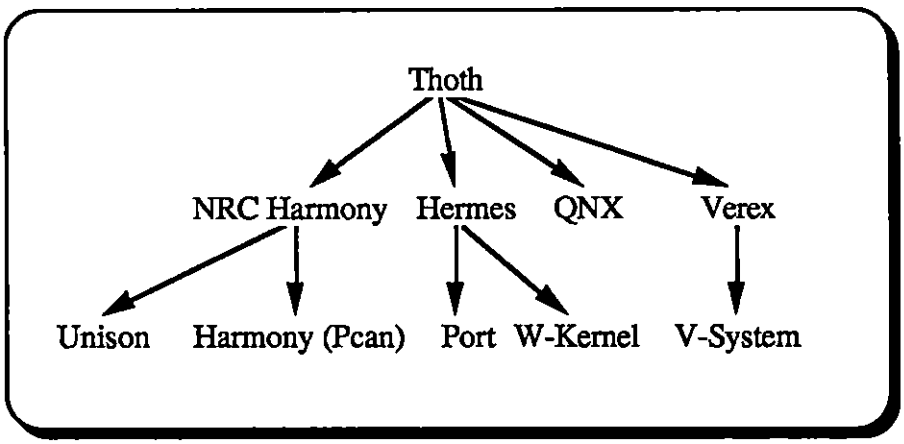


Figure 5.2 — Send-Receive-Reply Family Tree

Features

Unison is a derivative of NRC Harmony. It is a portable, real-time, multitasking, multiprocessor operating system using synchronous Send-Receive-Reply messages for inter-task communication. Unison provides transparent multiprocessing: a task does not know (or care) on which processor it is running. These key characteristics of Unison are discussed below. Further Unison details can be found in the Unison documentation [MTI87].

Unison can be *ported* to different machine types. Currently, only the Motorola MC680x0 family of processors is supported. Unison can be ported to almost any MC680x0 based computer, however, if multiple processors are required, there are a few special requirements. First, there must be some shared memory in the system that can be accessed by test-and-set instructions. Second, there must be a mechanism to generate processor specific interrupts. Unison uses these interrupts for inter-kernel communication.

Unison provides support for *real-time* activities. Real-time programs must interact

with the real world and must deal with time itself as a critical resource [GENT83]. Interaction with the real world requires dealing with asynchronous events, unanticipated events, and recovering from errors. Tasks and interrupt service routines are the mechanisms supporting asynchronous activities. Error recovery and coping with unanticipated events is simplified as a result of dynamic resource allocation. Resources such as tasks can be created and destroyed, memory can be allocated and freed, and connections to device servers can be opened and closed.

Time can be a critical resource in three ways. *Response* time is the most obvious case of time as a critical resource. An asynchronous external event may need to be handled within a guaranteed time. The *throughput* of events may also be critical. Finally, *periodic* activities may need to be performed at the appropriate times within specified tolerances. Unison uses fixed priority tasks and interrupt service routines to guarantee response time. Since tasks execute until they block or a higher priority task becomes ready, it is possible (although quite tedious) to examine all the tasks in a system and determine the maximum latency. In general, only the higher priority tasks need to be considered. For example, to determine if a time critical task will meet its goal, the worst-case execution times and periodicity of all tasks of equal or higher priority must be examined. Often it is only feasible to schedule less than ten time critical tasks on a single processor. If it becomes too difficult or impossible to schedule all events on a single processor, more processors can be added to the system and the tasks and external events distributed amongst them. Once the tasks are distributed, only tasks on the same processor need be considered. Of course, periodic activities and activities requiring a guaranteed throughput can also be distributed amongst the processors. To ensure minimum response time, it is also possible to dedicate an entire processor to handling a specific event.

Interrupt routines and tasks are statically assigned to processors at configuration time. For each processor a set task templates is defined indicating the particular tasks can be instantiated on that processor. A similar set of templates exists for interrupt routines. The assignment of interrupt service routines and tasks to processors can be difficult with large systems involving numerous external events. Currently, there is no simple way to decide upon an optimal configuration. Ad hoc configuration techniques are used and as such are rarely documented. This can lead to maintenance problems since the reasons behind a particular configuration are rarely documented and if changed can cause runtime errors. MTI is developing tools to help automate and document the configuration process.

Tasks have been defined in [PARR86, RIES86 and many others] but in the context of Unison "a task is best thought of as like a subroutine, except that an instantiation of it must be explicitly created, and once it has been created, it executes independently of, and in parallel with, the task that created it. Each task itself executes sequentially and synchronously, like a conventional program — the asynchronism and parallelism are strictly *between* tasks" [GENT83].

Transparent multiprocessing is an important characteristic for evolving software systems. Many software systems begin operating at a minimal level and as more development is performed, provide more functionality. Increased functionality often implies increased computational requirements; therefore, it is advantageous to have the ability to add processors to a processing node.[5] Allocation of tasks to processors is a configuration decision, not a software decision. This transparency allows simple migration

5 In the context of this work, a processing node is a computational unit within which operations can be performed in real time. Processing nodes are typically connected via a non-real-time communication link (e.g. Ethernet).

to more powerful computing nodes since more processors can be added at any time, requiring only the configuration tables to be changed. No software needs to change.

5.3.2.2. Application Software

A prototype control system for a proposed KAON Factory was implemented as an example application. This software is comprised of three main components. The KAON Factory Simulator (KFS) simulates various activities that would exist in the actual KAON Factory. The KAON Factory Automatic Control System (KFACS) controls the activities of the KFS. The KAON Factory User Interface (KFUI) allows user interaction with KFACS and KFS and the display of various data. This software system is discussed in more detail in Chapter 6.

5.3.2.3. User Interface Software

A variety of window systems were used for various phases of the development of the KAON Factory Application. This section outlines the various characteristics that were of interest and how the window system was used within the scope of this project. This section does not comment on the window systems.

NeWS[6] is a distributed, extensible window system [SUN87b]. It is an attempt to step up to a new level of technology, utilizing the more powerful workstations and graphical input and output devices that are now readily available. Key characteristics of NeWS are 1) it is distributed: client programs may execute on machines different from the NeWS server;[7] 2) the client-server language is turing equivalent; 3) the basic imaging

---

6 NeWS refers to version 1.1 of NeWS from Sun Microsystems Inc.

model is based on the stencil/paint model [WARN82]; 4) the event based input system is user-extensible. NeWS does **not** provide a model for user interaction, instead it provides the tools with which user interfaces can be constructed. NeWS was used as the basis for the KAON Factory User Interface.

Although NeWS can be used as a graphical interface to Unix for development purposes, other window systems are more suited to this task. Both Suntools [SUN85] and Version 10 of the X window system were used during development. Suntools is a well known, but older window system for Sun Workstations. It was used to support the Remedy debugger and dbxtool in conjunction with CopyCat. The X window system is a server-based distributed window system based on the raster imaging model. This system was evaluated for suitability as a basis for the user interface in place of NeWS, but was rejected due to lack of user interface support software. Version 11 of the X window system was not evaluated and may prove more useful than NeWS.

---

7 The NeWS server must run on the machine that is physically connected to the display device.

Chapter 6

The Prototype Control System

6.1. Introduction

This project is an attempt to satisfy the requirements of the KAON Factory Control System (KFCS) as outlined in Chapter 4 and detailed by Dawson, Dobinson, Gurd and Serre in [DAWS87]. In this chapter various aspects of the design and implementation of the KAON Factory Application (KFA) are discussed. A more detailed description of the design is given by the author in [BAKE88b]. The KFA is a prototype system whose goal is to experiment with the programming paradigm in a real application and also determine the feasibility of various hardware and software architectures that may be used in the KAON Factory Control System.

Many of the long-term maintenance and expandability requirements were met by using off-the-shelf products adhering to published standards. The microprocessor bus in the real-time node[1] is based on the VMEbus, a standard 32-bit microprocessor bus. The interconnection between the real-time node and the host computer is a via Ethernet using the IEEE 802.3 protocol. The boards used in the real-time node are off-the-shelf boards

---

1 The real-time node is the target system described in Chapter 5. It is a crate housing a VMEbus and a number of processors and I/O devices.

designed and manufactured by a Canadian company. The host computer is a Sun Workstation™ running Berkeley Software Distribution (BSD) 4.2 Unix, a quasi-standard operating system. The user-interface software is based on NeWS, a Network extensible Window System from Sun Microsystems. It was hoped that this system would provide a good basis for user-interface development. The X window system was also investigated as to its applicability in this environment.

The performance, distributed intelligence, and expandability requirements are satisfied in two ways. First, within the real-time node, processors can be added for increased processing power at the real-time level. Adding processors here can greatly simplify the scheduling problems associated with servicing time-critical devices since processors can be dedicated to servicing these devices. Adding processors with on-board I/O hardware increases the I/O handling capability of a real-time node with minimal effect on existing software executing on other processors. Additional real-time nodes and other types of processing complexes can also be added to the network. Note that communications across the network are not in real-time and have no guaranteed response time.

Language and operating system requirements are also addressed. The C programming language [KERN78] is used for all software components of the KAON Factory Application. C provides the essential portability and flexibility required for this type of system software. While the existing BSD4.2 C is a weakly-typed language, many enhancements described in [HARB84] are adopted for the ANSI standard for C that permit the programmer to fully specify types. In particular, the new standard allows full typing of functions, hence these ANSI C compilers will perform stronger type checking than their

predecessors. C also allows enough control to directly access memory-mapped hardware devices without resorting to assembly language programming.

The Unix BSD 4.2 operating system is used for both development of the target software and for the host system supporting the user interface software. While Unix is still evolving, it is a quasi-standard operating system. There are two main Unix groups, the BSD group and the System V group. An ANSI committee is working towards a recognized standard operating system called POSIX. It is hoped that POSIX will be a combination of the best features of each operating system. Unison, described in Chapter 5, is the real-time operating system executing on the real-time node. Since there is no generally accepted real-time operating system, a modern message-based multitasking, multiprocessor operating system was chosen.

6.2. Overview of the KAON Factory Application

The KAON Factory Application (KFA) consists of three major components: the KAON Factory Simulator (KFS), the KAON Factory Acquisition, and Control System (KFACS), and the KAON Factory User Interface (KFUI). The KFS simulates the proposed synchrotron, the KFACS controls the KFS, and the KFUI allows the operator to affect the KFACS and KFS. Figure 6.1 shows the "uses" relationship [PARN79] for these modules.

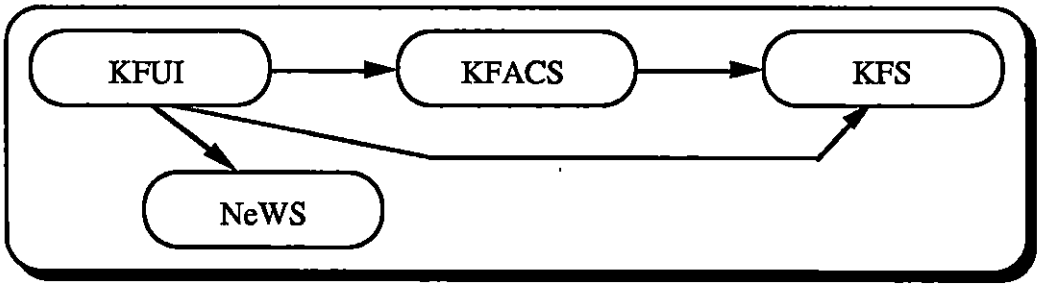


Figure 6.1 — Software Configuration - Uses Relationship

The KAON Factory Simulator executes on the target hardware (the real-time node). This component is responsible for simulating a variety of synchrotron activities such as beam generation, beam acceleration, beam deflection, beam focusing, and beam detection. It is possible to connect real or virtual devices to the simulator. For example, an analog I/O module may be attached and influence the beam according to external stimuli. Alternately, a virtual device may be connected such that the operator has a slider control that influences the beam. In a real system, the KFS would be replaced by the actual synchrotron and hardware input/output devices.

The second component, the KAON Factory Acquisition and Control System, also runs on the target hardware. It is responsible for controlling and monitoring the KFS. The KFACS monitors various KFS sensors in order to maintain appropriate beam conditions. Various KFS alarm conditions (such as out of bounds beam position, deflection magnet overheating, etc.) are monitored. Normally, this sub-system will run according to predefined control settings, however, the KAON Factory User Interface may change operating parameters or request sensor readings.

The third KFA component is the KAON Factory User Interface. This sub-system

executes partially on the host and partially on the target. The host and target parts communicate via Ethernet sockets. The KFUI allows the operator to change any KFS operational parameter (i.e. accelerating potential or frequency, deflection field strength, etc). The KFUI also performs continuous monitoring of KFS activities. Finally, the KFUI gives operator control over the KFACS. The operator controls are adjusted by means of a graphical display and a pointing device (e.g. mouse). The WIMP (Window, Icons, Mice and Pop-up menus) user interface model is used. It was hoped that a Macintosh™ style user interface [APPL87] could be implemented, however this proved too time consuming given the existing NeWS tools.

6.3. The KAON Factory Simulator

The KFS consists of five of relatively independent components, the deflection magnets (DM), the focusing magnets (FM), the accelerator stations (AS), the ring or beam container (BC), and the beam generator (BG). The general sequence of operations is as follows:

- the ring requests a new beam bunch
- the beam generator produces a beam bunch
- the beam bunch enters the ring which repeatedly performs the following operations:
    - the ring gives the beam bunch to an accelerator station which in turn returns a modified beam bunch to the ring
    - the ring gives the beam bunch to a deflection magnet which in turn returns a modified beam bunch to the ring
    - the ring gives the beam bunch to a focusing magnet which in turn returns a modified beam bunch to the ring

It is possible for external control tasks to query the various components as well as provide control input. For example, a trending task may query a focusing magnet (which has beam position sensors) for the most recent beam position, or an operator may set a new value for the field strength of a particular magnet.

Each type of physical station, such as an accelerating station or deflection magnet, is represented by a task. If there are 48 deflection magnets around the ring, then there are 48 instances of the task that simulates a deflection magnet. The same for accelerating stations and focussing magnets. This simple model allows the representation of each physical station to have independent state, as do the real stations. The overall task structure is shown in Figure 6.2.

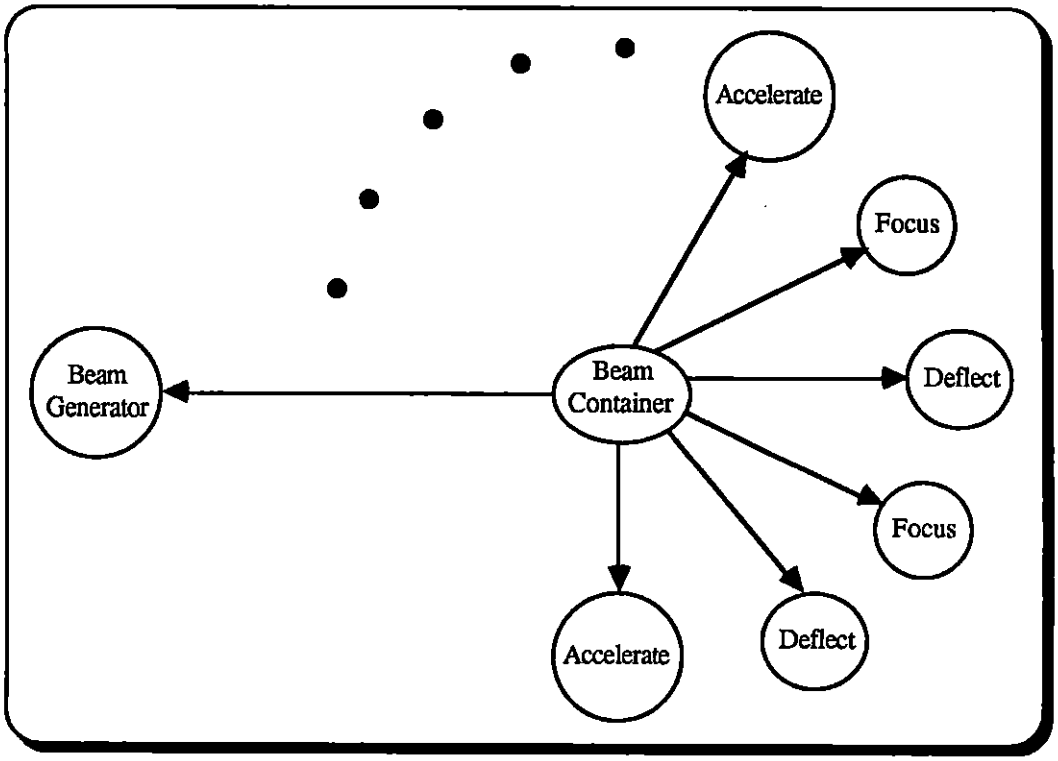


Figure 6.2 — Task Communication Structure

The *beam container* (the ring) is a client task. It is the logical medium for the beam and houses the beam in its various stages. Initially, the ring is empty and sends a message to the beam generator requesting a beam bunch. The ring sends this new beam bunch to an accelerator to increase its kinetic energy. Next the beam is sent to a deflection magnet (we don't want it getting out of the ring) to maintain its circular path. Next, the beam is sent to a focusing magnet. The accelerators, deflectors, and focusers exist at many intervals around the physical ring, hence the beam is sent to a number of instances of these tasks in sequence. There need not be an equal number of accelerator stations, deflection magnets, and focusing magnets.

While there is only one beam container depicted, a simple extension to this model could include a number of beam containers, representing rings A to E [DAWS87] of the KAON Factory, each obtaining beam bunches from the preceding ring. Only ring A would obtain a beam bunch from the beam generator. For the KAON Factory, the beam generator would represent the existing TRIUMF cyclotron.

The *beam generator* is a connectionless server task, an administrator, which creates a beam bunch that has properties such as static and relative mass, velocity, direction, vertical and horizontal oscillation, etc. The beam generator produces beam bunches at periodic intervals. A ring, requesting a beam bunch, is blocked between intervals.

The *accelerating stations, deflection magnets,* and *focusing magnets* are also administrator tasks. A number of instances of each type of task is created by the beam container. The accelerating stations increase the energy of a beam bunch. The deflection magnets change the direction of the beam bunch as well as affecting the dispersion and

shape of the beam bunch. The focusing magnets modify the dispersion of the beam bunch. These stations also house the beam position sensors.

The interfaces to each component of the KFS allow various parameters to be set, such as beam generation frequency, and operating conditions to be sensed, such as beam position or magnet temperature. Each station also has an interface to block the invoking alarm notifier until some alarm condition arises within the station. This allows the information necessary to determine alarm conditions to be encapsulated within the individual components. External alarm notifiers do not need to know anything about the type of component that is being monitored. When an alarm condition arises, a message indicating the type of alarm is replied to the notifier that is blocked awaiting an alarm.

Within any real-time node, the interfaces to the KFS are processor independent so that a task invoking one of the KFS interfaces does not need to know on which processor the associated simulator task is executing. The KFS interfaces also hide the existence of the actual simulator tasks from the invoking task. The invoking task only needs to know that it will be *sending* a request. This knowledge is necessary in order to establish blocking diagrams when designing applications that use the simulator.

6.4. The KAON Factory Acquisition and Control System

The KFACS is responsible for controlling the KAON Factory simulator and maintaining optimal operation. The KFACS consists of three main modules. The Control Module is responsible for controlling system variables such as beam position and focus, controlling and monitoring various hardware operating conditions such as magnet coil temperatures. The Alarm Module provides an alarm capability that can be attached to any (simulated or real) physical process. The Trend Analysis Module performs trend

monitoring operations in response to KFUI requests. These operations provide the raw data used for trend display in the KFUI.

The Control Module consists of a control administrator task and zero or more control worker tasks as shown in Figure 6.3. The control administrator task is responsible for the general operations of the KAON Factory and maintains operational state information. The control workers perform actions on behalf of the control administrator to free the administrator during potentially long operations. During startup, the administrator creates a control worker for each KFS station. Each worker is then responsible for maintaining operations at its specified station. The administrator is responsible for coordinating the efforts of all workers to ensure that inter-station adjustments are consistent.

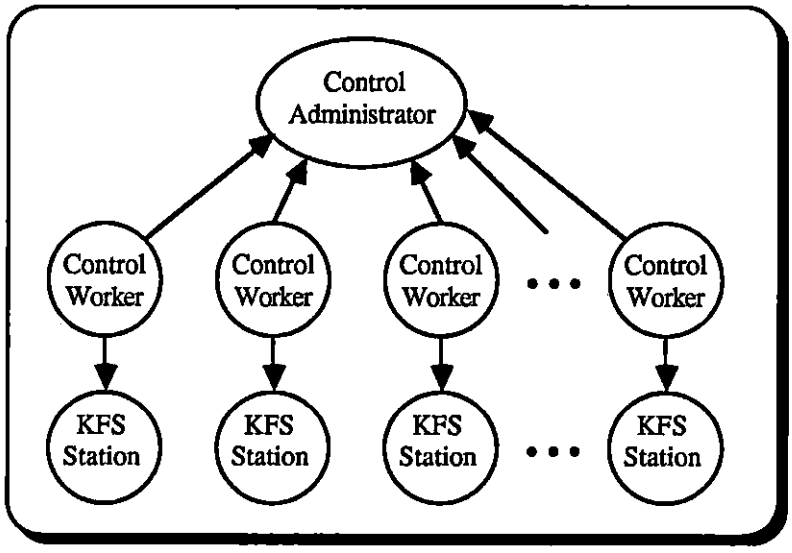


Figure 6.3 — Control Module Sub-system Task Communication Structure

The Alarm Module consists of an alarm administrator task and zero or more alarm notifier tasks as shown in Figure 6.4. The alarm administrator task is responsible for creation and management of alarm notifier tasks. When a client needs some activity monitored for alarm conditions, it sends a **create** message to the alarm administrator which creates the alarm notifier. The client then sends an **initialization** message to the notifier, indicating the type of activity to monitor. The alarm notifier task sends an **alarm query** message to the designated KFS station. The KFS station only replies to the alarm task when an alarm condition arises. At this time, the alarm notifier sends the **alarm message** to the KFUI alarm agent. In addition, if the alarm is severe, such as the beam becoming too far out of alignment, then the alarm notifier can shut down the beam generator or take other appropriate action (i.e. run quickly from the building, dragging its real-time node behind it).

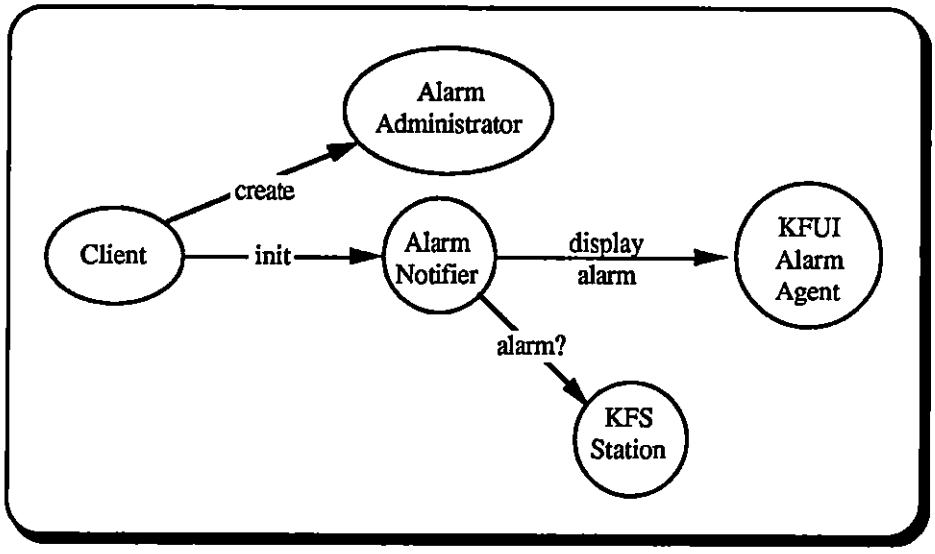


Figure 6.4 — Alarm Sub-system Task Communication Structure

The Trend Analysis Module consists of a trend administrator task and zero or more trend worker tasks as shown in Figure 6.5. The structure of this module is similar to the alarm module in that a task is created to monitor a specified condition (e.g. magnetic field strength, beam position). The client sends a **create** message to the trend administrator. The administrator creates a trend worker for the client. The client then sends an **initialization** message to the trend worker, indicating the frequency and activity to be monitored. The trend worker task collects data at appropriate intervals and subsequently sends the data to a KFUI trend agent at appropriate times. The agent then transfers the data to the display device.

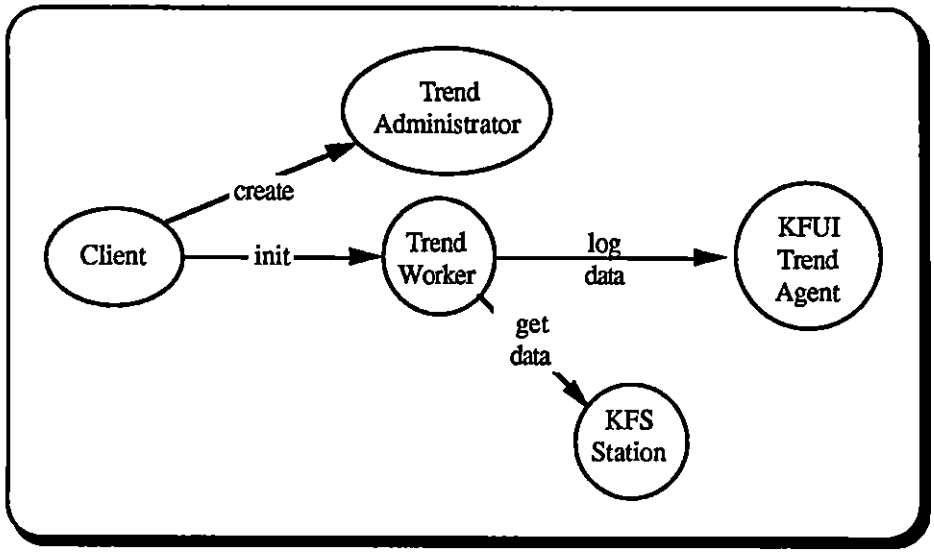


Figure 6.5 — Trend Analysis Sub-system Task Communication Structure

6.5. The KAON Factory User Interface

The KFUI directs user input to the appropriate control modules and displays graphical and textual information on the screen. The KFUI is capable of adjusting both KFACS and KFS parameters. A user interface for a large system such as the KAON Factory will require a significant amount of research to determine an optimal interaction model. This project makes no attempt at this research so the KFUI is a simple 'bare bones' user interface.

Pop-up menus[2] are used for selection from a number of alternatives. The KFUI provides four categories[3] of menu items. The KFACS can be started and stopped, KFS stations can be monitored for alarms, trending can be performed on a KFS station, or KFS stations can be *adjusted*. The menu items, shown in Figure 6.6, have the following meaning.

---

2 Pop-up menus appear under the cursor whenever a particular mouse button is pressed.

3 Interactive assignment of trend and alarm displays to KFS stations proved too awkward under NeWS so these displays are attached automatically during system startup. Adjustment sliders are also assigned automatically.

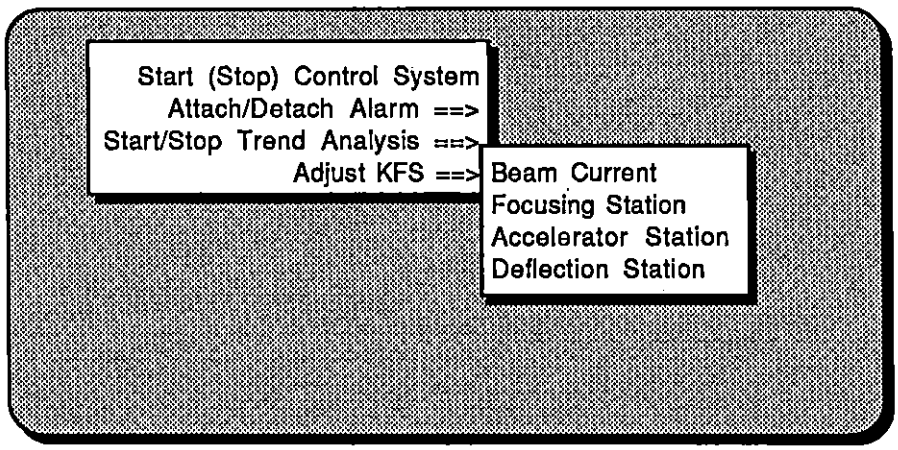


Figure 6.6 — KFUI Main Menu and Sub-menu

1) The Control System can be started if it is not already running or stopped if it is currently running. No adjustments to the KFACS are implemented.

2) An alarm monitor can be attached to any KFS station or detached from a station. After selecting the *Attach/Detach Alarm* menu item, a sub-menu is displayed to allow the user to select which type of station is to be monitored. When a station type is selected, the operator must enter the number of the particular station to be monitored. A more sophisticated method of selecting the station would be a graphical presentation of all stations and clicking on a station would cause it to be selected. For this implementation, the alarm monitors are attached automatically, at startup. Figure 6.7 shows four possible alarm messages that can occur upon attaching an alarm.

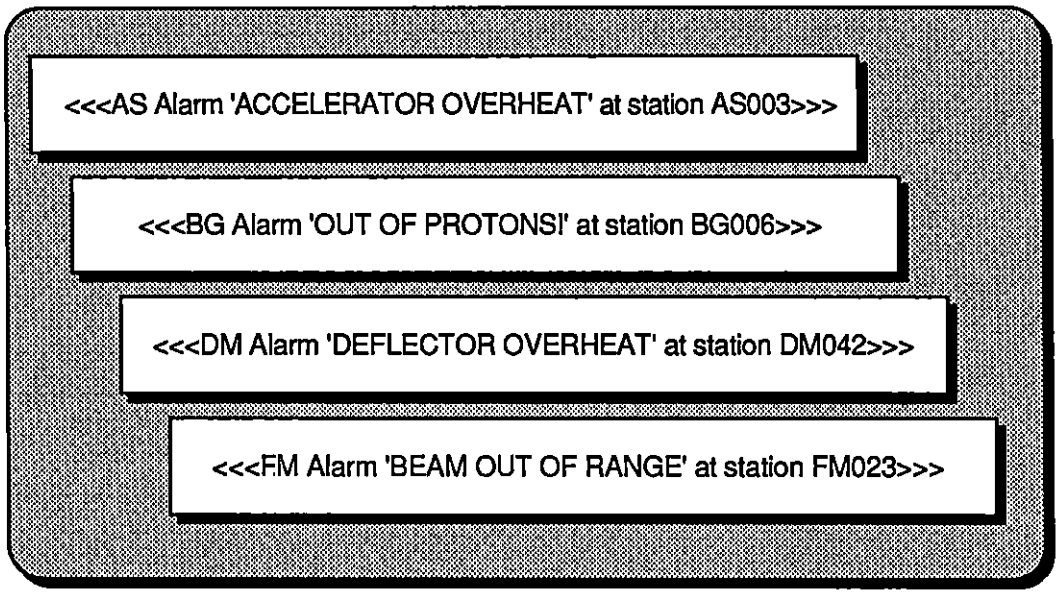


Figure 6.7 — Active Alarm Windows

3) Trending can be attached to any 'readable' characteristic of any KFS station. The station selection procedure is identical to that for alarm monitors. Figure 6.8 shows a trend window for the field strength of deflection magnet DM017 (the 17^{th} deflection magnet around the ring). For this implementation, the trend displays are attached automatically, at startup.

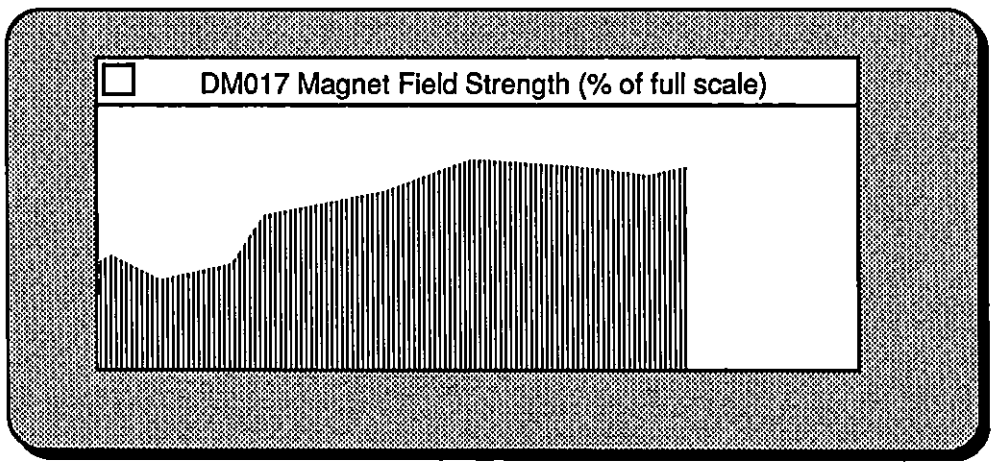


Figure 6.8 — Trend Display Window

4) Sliders can be attached to or detached from any 'writeable' KFS characteristic. Station parameters such as beam current setpoint, centre of focus setpoint, acceleration potential setpoint, and deflection field strength setpoint can be adjusted with these sliders. Figure 6.9 shows five sliders, the top two attached to deflection magnets DM000 and DM001, the next attached to beam generator BG000, and the last three attached to accelerating stations AS002, AS001, AS000. For this implementation, the sliders are attached automatically, at startup.

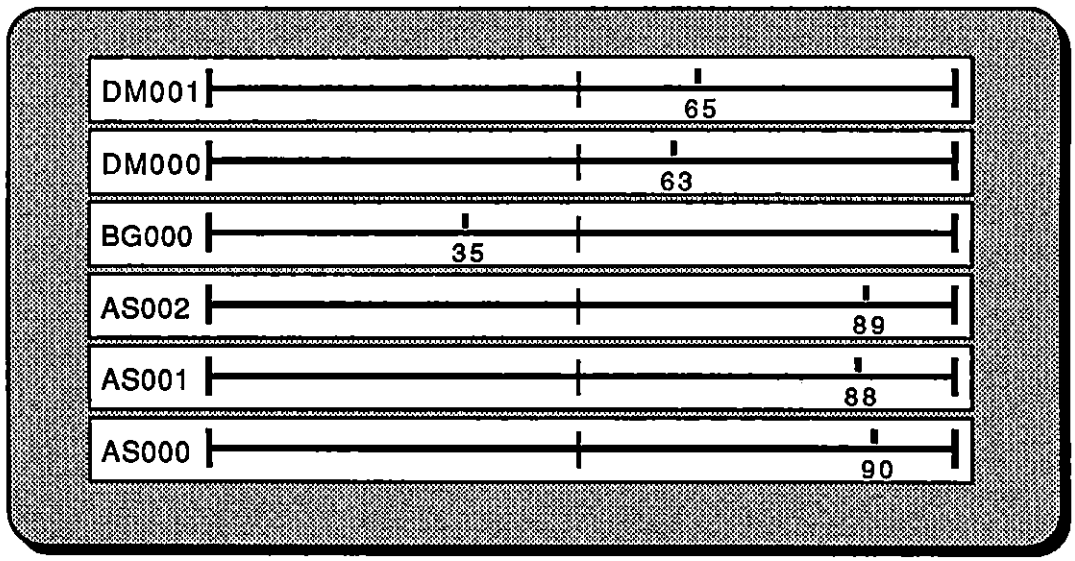


Figure 6.9 — KFS Station Setpoint Sliders

KFUI Task Structure

The KFUI task structure is somewhat more complex than either the KFS or the KFACS. This is partly due to the complexity of Unix to Unison communications needed for distributing the user interface software across machines and partly because of the need to completely isolate the KFUI from the KFACS and KFA. It is desirable that the KFACS and KFS software is independent of the particular user interface software being used so that the user interface software can change without affecting either the KFACS or KFS.

The KFUI software is distributed across three different environments. First, there is the PostScript based software running under NeWS which runs under Unix. Second, there are NeWS clients, a combination of C and PostScript™ programs, that run under Unix and communicate with both the NeWS server and some Unison tasks. Finally, there are Unison based tasks that communicate with the NeWS clients as well as with the

KFACS and KFS.

Because it is difficult (due to lack of appropriate NeWS documentation) for Unison tasks to communicate directly with the NeWS server it is necessary to create Unix processes to act as intermediaries. Three such Unix processes are used. The Unix Event Courier gets an event from a lightweight[4] process in the NeWS server. The courier then sends the event to either the Unison Control Agent or the Unison Menu Agent which then sends appropriate requests to KFS tasks. The Unison Alarm Agent receives alarm messages from Unison Alarm notifiers whenever an alarm condition arises. The Unison Alarm Agent then sends the alarm message to the Unix Alarm Agent which then displays an appropriate alarm window on the operator console via a NeWS lightweight process. The Unison Trend Agent receives trend data messages from Unison Trend notifiers. The Unison Trend Agent then sends the trend message to the Unix Trend Agent which then updates (or creates) the appropriate trend display window via a NeWS lightweight process.

The communication blocking diagram shown in Figure 6.10 omits details such as vultures and watchdogs attached to the agents that are not pertinent to the blocking characteristics of the KFUI itself. Figure 6.11 illustrates the details of these agent tasks.

---

4 Lightweight processes are sub-processes within a Unix process. Lightweight processes have very low creation and context switch overheads.

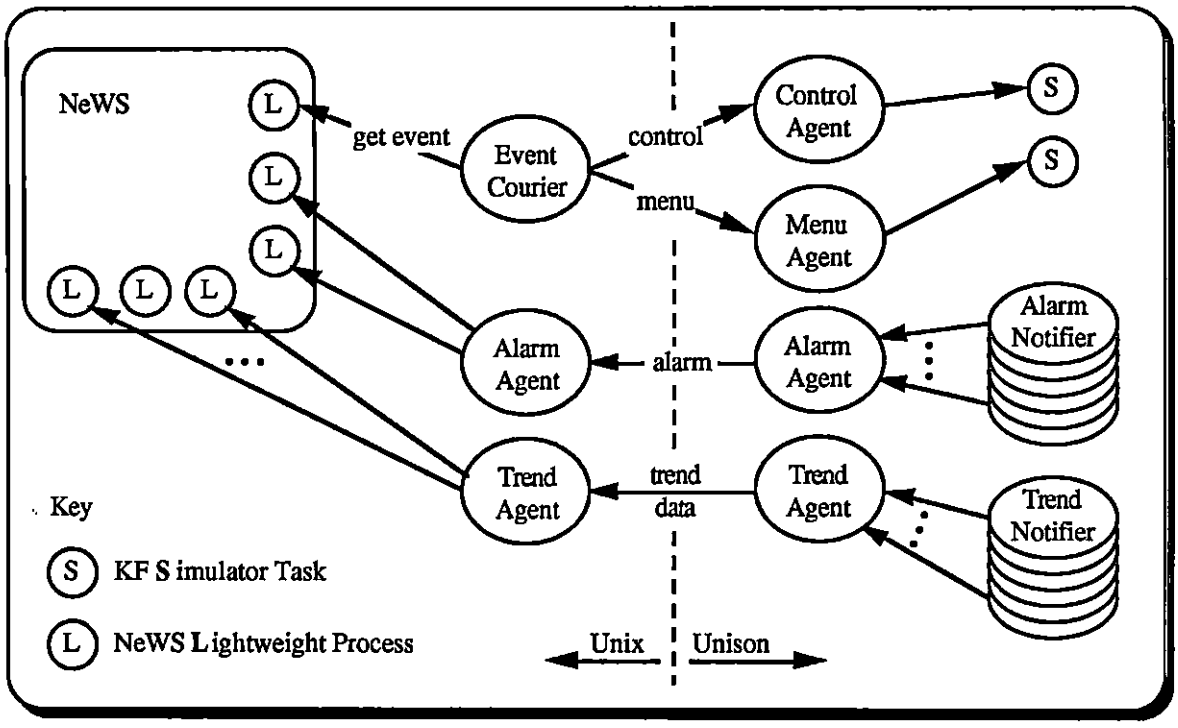


Figure 6.10 — KFUI Communication Blocking Diagram

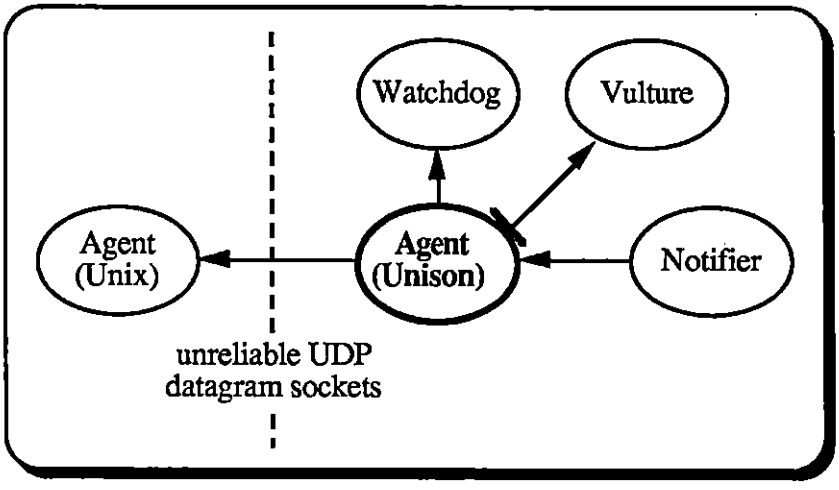


Figure 6.11 — Trend or Alarm Agent Blocking Diagram

6.6. Summary

The use of the S-R-R programming paradigm was used throughout the design. The paradigm can also be used when real I/O devices are integrated into the application. Deadlock detection was not necessary since the system is composed of components (i.e. servers) that are free of internal deadlock and communication protocol rules prevent deadlock between components. Due to the small number of processors used in the system, measures of concurrency are not particularly meaningful, however, it should be noted that the ease with which tasks can be reassigned to another processor within the real-time node is a significant advantage.

The connectionless servers of the KFS are examples of how data abstraction and encapsulation can be achieved by the use of tasks. The advantage of this implementation is its ease of expansion (one can refine the simulator as needed) and the straightforward increase in computational power through the addition of processors. This software and hardware configuration may prove more powerful and versatile than simulations performed on larger general purpose timesharing computers. Furthermore, the use of this model may prove useful in the actual simulations that will be necessary for the installation, testing and general operation of the KAON Factory.

6.7. Future Work

It is possible for the simulator to be extended to provide more realistic modeling of the KAON Factory. In particular, each of the components of the simulator can be individually enhanced and tested. It is also possible to integrate real I/O devices into the simulator so that the simulator is a mixture of simulation and a physical system. This

approach may be useful during the installation and testing phases of the KAON Factory. Process control algorithms can be developed and tested using the KFACS and KFS as building blocks.

A graphical display of the real or simulated KAON Factory would be useful, especially if the operator could select a station by clicking on it, then have information about that station displayed and made accessible for change.

Research into user interface techniques for process control is necessary to achieve a good interaction with the final control system. The use of NeWS was unsatisfactory in many respects. Other distributed window packages must be investigated to determine their value in this type of project. The user interface for the control system should include a simple mechanism for dynamic attaching and detaching of control devices to system processes.

Chapter 7

Conclusions

Due to the complexity of designing systems based on the communicating task model, it is clearly desirable to have a standard structuring mechanism to avoid re-inventing task relationships for each application. It is also desirable that this mechanism intrinsically avoid pitfalls such as deadlock and race conditions due to uncontrolled access to shared objects. Furthermore, the mechanism should permit the design of highly concurrent programs that can exploit the many powerful multiprocessor architectures available today. A programming paradigm based on the Send-Receive-Reply task communication primitives presented in this thesis is proposed as such a mechanism.

The Send-Receive-Reply model is good basis for a programming paradigm because of its simple and elegant solutions to many of the problems that plague other inter-task communication mechanisms. The model provides synchronous bi-directional transfer, permits static deadlock detection, does not require message buffering, and naturally supports the object-oriented client-server model. Commercial operating systems that provide the S-R-R model are readily available for a variety of processors.

The S-R-R programming paradigm is a small but sufficient set of related task types

that can be used as the building blocks for a large class of multitasking applications, including real-time and multiprocessing systems. The role of each task type has been described, along with their allowable inter-task communication semantics. Enough detail has been provided in the form of code templates and other system support mechanisms to allow a software developer to directly use the paradigm. A significant amount of the work involved in designing a system is therefore done, allowing the developer to concentrate on the particular problems of the application.

An example application using the S-R-R programming paradigm has been presented illustrating how the paradigm can be used to solve problems presented by a large real-time system. Each of the task types is exploited in a prototype control system for a proposed KAON Factory.

The programming paradigm, in various stages of its evolution, has also been used in a number of significant real-time and non-real-time applications. A three dimensional graphics renderer was developed to exploit a high performance multiprocessor. A real-time videotape controller, using the same hardware as the renderer, is also based on the programming paradigm. More recently, a real-time, multiple channel, digital voice messaging system was developed using key concepts from the paradigm. Each of these applications was non-trivial, however the paradigm greatly simplified the design and made each system significantly easier to maintain.

It is hoped that, through this thesis, others will recognize the usefulness of the Send-Receive-Reply primitives and perhaps successfully apply the programming paradigm to their particular applications.

Suggestions for Further Research

It is possible that a real-time response can be achieved across distributed real-time nodes using IEEE 802.3/Ethernet [ANSI85] as the communication medium. A method using *urgent* messages to achieve real-time response of selected messages over Ethernet is described by Ciminiera, DeMartini, and Valenzano in [CIMI88]. Using urgent messages for inter-processor communication, a distributed real-time system consisting of many tightly-coupled real-time multiprocessors is conceivable. At least one project is underway in which a distributed version of Harmony is under development [GENT88]. Such a system, together with urgent messages and the programming paradigm, would allow a real-time application to be distributed across multiple nodes.

Object-oriented environments such as Smalltalk-80 and C++ are interesting test beds for developing reusable, extensible software components. Message-based operating systems such as Harmony are similar in that tasks can be considered objects. Inheritance, however, is not provided. By standardizing messages and simulating the Forward primitive as discussed in Appendix B, inheritance could be added to Harmony in a straightforward manner. Implications of extensible servers or other task types via inheritance should be investigated.

Message timeouts are a contentious area of discussion. Purists claim that timeouts are unnecessary since they can be implemented on top of a simpler system. However, overhead involved in their implementation may impact the rest of the application. Further research is necessary to determine if timeouts are necessary, and if so, where they should be placed and for what primitives. A detailed analysis of both the performance and software engineering tradeoffs would be beneficial.

Dynamic load balancing could be achieved using a simple scheme involving servers and workers. An analysis of various scheduling algorithms based on the server-worker relationship could be contrasted with other load balancing schemes that are integrated into an operating system.

A detailed performance comparison between this programming paradigm and shared memory programming paradigms could be performed. Such a study should investigate and compare both initial performance characteristics and performance behavior over the life of the software. An analysis of the tradeoffs between ease of software maintenance and long term performance would be useful.

References

ANSI85 ANSI/IEEE, Standard 802.3, "Carrier Sense Multiple Access with Collision Detection", 1985.

APPL87 Apple Computer Inc., "Human Interface Guidelines: The Apple Desktop Interface," Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

ATKI85 Atkins, M.S., "The Role of ???? Mechanisms in Software System Design", Ph.D. Thesis University of British Columbia, 1985.

BAKE87a Baker, P.J.M., "Digital Voice Messaging System — Design Document," Working Paper, Island Software Ltd., Victoria, 1987.

BAKE87b Baker, P.J.M., "An Overview of Computer Bus Issues," Working Paper, University of Victoria, September 1987.

BAKE88a Baker, P.J.M., "Standard Computer Buses for Multiprocessing," submitted for publication, University of Victoria, May 1988.

BAKE88b Baker, P.J.M., "A Prototype Control System for the Proposed KAON Factory — Design Document," Working Paper, University of Victoria, July 1988.

BASK79 Baskett, F., Howard, J.H., Montague, J.T., "Task Communication in Demos," *Proceedings of the Sixth ACM SIGOPS Symposium on Operating System Principles,* November 1979.

BOAR84 Boari, M., Crespi-Reghizzi, S., Daprá, A., Maderna, F., Natali, A.,

"Multiple-Microprocessor Programming Techniques: MML a New Set of Tools," *IEEE Computer*, January 1984.

BOOC86 Booch, G., "Object-Oriented Development," *IEEE Transactions on Software Engineering*, Vol SE-12(2), February 1986.

BORI85 Borrill, P.L., "A Comparison of 32-bit Buses," *IEEE Micro*, December 1985.

BSD84 "UNIX Programmers' Manual," 4.2 Berkeley Software Distribution, University of California, Berkeley, California, March 1984.

BYTE81 Special Issue on Smalltalk-80, BYTE Magazine, August 1981.

CASH80 Cashin, P.M., "Inter Process Communication," *Working Paper*, Bell-Northern Research, 1980.

CHER79 Cheriton, D.R., Thoth, "A Portable Real-Time Operating System," *Communications of the ACM*, Vol 22(2), February 1979.

CHER84 Cheriton, D.R., "The V Kernel: A Software Base for Distributed Systems," *IEEE Software*, Vol 1(2), April 1984.

CIMI88 Ciminiera, L., DeMartini, C., Valenzano, A., "Industrial IEEE 802.3 Networks with Short Delivery Time for Urgent Messages,", *IEEE Transactions on Industrial Electronics*, Vol 35(1), February 1988.

CLEA86 Cleaveland, J.C., "An Introduction to Data Types," Addison-Wesley 1986.

CORN86 Cornhill, D., Sha, L., Lehoczky, J.P., Rajkumar, R., Tokuda, H., "Limitations of Ada for Real-Time Scheduling," CMU ART Project, 1986.

DAWS86 Dawson, W.K., Dobinson, R.W., "Buses and Bus Standards," *IEEE Spectrum*, 1986.

DAWS87 Dawson, W.K., Dobinson, R.W., Gurd, D.P., Serre, Ch., "A Conceptual Design for the TRIUMF KAON Factory Control System," Technology

Division of TRIUMF, July 1987.

DIJK65 Dijkstra, E.W., "Co-operating Sequential Processes," Programming Languages, ed. Genuys, F., London, Academic Press, 1968.

DIJK68 Dijkstra, E.W., "The Structure of the 'THE' - Multiprogramming System," *Communication of the ACM,* Vol 1(5), May 1968.

DVME86 DY-4, "DVME-750 Intelligent Local Area Network Controller Operations Manual, "DY-4 Systems Inc., Nepean, Ontario, Document number OM-VME750-999, issue A, December 1986.

DVME87 DY-4, "DVME-134 Single Board Computer Operations Manual," DY-4 Systems Inc., Nepean, Ontario, Document number OM-VME134-999, issue A, June 1987.

DYME85 Dyment, J.D., Personal communications, 1985.

GAMM85 Gammage, N., Casey, L., "XMS: A Rendezvous-Based Distributed System Software Architecture," *IEEE Software,* Vol 2(3), May 1985.

GENT81 Gentleman, W.M., "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept," *Software—Practice and Experience,* Vol 11, 1981.

GENT83 Gentleman, W.M., "Using the Harmony Operating System," National Research Council of Canada, ERB-966, NRCC No. 27469, December 1983, revised March 1987.

GENT88 Gentleman, W.M., Personal communications, 1988.

GOLD83 Goldberg, A., Robson, D., "Smalltalk-80, The Language and its Implementation, "Addison-Wesley 1983.

GREN83 Green, D., "Chorus — A Multiprocessor Architecture for Real-Time Control Applications," National Research Council of Canada, ERB-964, NRCC No. 23031, December 1983, revised April 1984.

GUST84 Gustavson, D.B., "Computer Buses — A Tutorial," *IEEE Micro,* Vol 4(4), August 1984.

HANS78 Hansen, P.B., "Distributed Processes: A Concurrent Programming Concept," *Communications of the ACM,* Vol 21(11), November 1978.

HARB84 Harbison, S., Steele, G. Jr., "C: A Reference Manual", Englewood Cliffs, NJ, Prentice-Hall, 1984.

HAUS87 Hausmann, G., "Implementing VRTX on the series 32000," *Microprocessors and Microsystems,* Vol 11(7), September 1987.

HOAR74 Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," *Communications of the ACM,* Vol 17(10), October 1974.

HOAR78 Hoare, C.A.R., "Communicating Sequential Processes," *Communications of the ACM,* Vol 21(8), August 1978.

ICHB79 Ichbiah, J.D. et al, "Rational for the Design of the Ada Programming Language," *ACM SIGPLAN Notices,* Vol 14(6), Part B, June 1979.

KERN78 Kernighan, B.W., Ritchie, D.M., "The C Programming Language," Englewood Cliffs, NJ, Prentice-Hall, 1978.

LISK88 Liskov, B., "Distributed Programming in Argus," *Communications of the ACM,* Vol 31(3), March 1988.

LOCK85 Lockhart, T.W., Personal communications, 1985.

MALC81 Malcolm, M.A., Staffort, G.J., Didur, P.A., "The Port Programming System," Department of Computer Science, University of Waterloo, 1981.

MAO80 Mao, T.W., Yeh, R.T., "Communication Port: A Language Concept for Concurrent Programming," *IEEE Transactions on Software Engineering,* Vol SE-6, March 1980.

MOKH84 Mokhoff, N., "Parallelism Makes Strong Bid for Next Generation Computers," *Computer Design*, Vol 23(9), October 1984.

MOTO85a "The VMEbus Specification - Rev C.1," PRINTEX Publishing Inc, October 1985.

MOTO85b "MC68020 Microprocessor User's Manual," second edition, Motorola Inc., 1985.

MTI87 "Integration Toolkit Manual," Multiprocessor Toolsmiths Inc., Kanata, Ontario, 1987.

MTOS84 "MTOS-UX/68K User's Guide," Industrial Programming Inc., 1984.

NIEL87 Nielsen, K.W., Shumate, K., "Designing Large Real-time Systems with Ada," *Communications of the ACM*, Vol 30(8), August 1987.

PARN72 Parnas, D.L., "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, Vol 15(12), December 1972.

PARN79 Parnas, D.L., "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, Vol SE-5, March 1979.

PARN88 Parnas, D.L., Faulk, S.R., "On Synchronization in Hard-Real-Time Systems," *Communications of the ACM*, Vol 31(3), March 1988.

PARR86 Parr, R.K., "Real-time Multitasking Multiprocessing Operating Systems: A Detailed Comparison of VRTX, MTOS, pSOS and Harmony," National Research Council of Canada, Ottawa, Ontario, 1986.

PSOS82 "pSOS-68K User's Manual," Software Components Group, Inc., Santa Clara, California, 1982, revised May 1987.

QUAN88 "QNX User's Guide," Quantum Software Systems Ltd., Kanata, Ontario, 1988.

RIES86 Riese, H.M., "Towards Harmony on Sylvan," M.Math Thesis University of Waterloo, 1986.

RITC78 Ritchie, D.M., Thompson, K., "The UNIX Time-Sharing System", *The Bell System Technical Journal*, Vol 57(6), Part 2, July-August 1978.

ROTH75 Rothkind, M.J., "The Source Code Control System," *Research Directions in Software Technology*, ed. Peter Wegner, M.I.T Press, 1979.

ROWE86 Rowe, P.K., Vishnubhatla, B., Wilson, J., Boucouris, S., Pagurek, B., "Melody: A Multiple Process, Multiprocessor, Symbolic Debugger," ARTT Project, Carleton University, Ottawa, 1986.

SCHA86 Scharf, W., "Particle Accelerators and Their Uses - Part 1," Harwood Academic Publishers, Chur, Switzerland, 1986.

SIMP86 Simpson, D., "Multiprocessors us Radical Architectures," *Mini-Micro Systems*, May 1986.

STRO86 Stroustrup, Bjarne, "The C++ Programming Language," Addison-Wesley, 1986.

SUN85 "Programmer's Reference Manual for SunWindows," Sun Microsystems Inc., 1985.

SUN86a "Debugging Tools for the Sun Workstation", Sun Microsystems Inc., Mountain View, California, February, 1986.

SUN86b "Inter-Process Communication Primer", Sun Microsystems Inc., Mountain View, California, February, 1986.

SUN87a "NeWS Technical Overview," Sun Microsystems Inc., Mountain View, California, 1987.

SUN87b "NeWS 1.1 Manual," Sun Microsystems Inc., Mountain View, California, 1987.

TANN87 Tanner, P.P., Personal communications, University of Waterloo, Ontario, 1987.

TICH82 Tichy, W.F., "Design, Implementation, and Evaluation of a Revision Control System," *Proceedings of the 5th International Conference on Software Engineering,* IEEE, 1982.

TRIU85 "KAON Factory Proposal," TRIUMF: Canada's National Meson Research Facility, 1985.

TRIU88 Personal communications with TRIUMF personnel, 1988.

VERT86 Private communications with employees of Vertigo Systems International (while the author was a Vertigo employee), Vancouver, B.C., 1986.

VRTX87 "VRTX32/68020 User's Guide," First edition; release 1.04, Ready Systems, Palo Alto CA, April 1987.

WARN82 Warnock, J., Wyatt, D., "A Device Independent Graphics Imaging Model for Use with Raster Devices," *Computer Graphics,* Vol 16(3) July 1982.

WILS84 Wilson, P., "Thirty-Two Bit Micro Supports Multiprocessing," *Computer Design,* Vol 23(6), October 1984.

Appendix A

Differences Between the Harmony and Unison Kernels

A.1. Introduction

This appendix describes the key differences between the Unison[1] and Harmony[2] kernels with respect to the Send-Receive-Reply Programming Paradigm. These differences are important since the paradigm is affected by implementation details not inherent in the S-R-R model. The differences between the development environments and device support are not described since these issues do not impact the paradigm. The differences addressed are message passing semantics, task creation mechanisms, interrupt mechanisms, and server support.

A.2. Message Passing Semantics

Harmony was designed as a real-time, multiprocessor operating system. Unison is a commercialization of Harmony that has been modified to provide straightforward support

---

1 Unison Software Release 1.1.

2 Harmony Release 2.0..

of Ada tasks. These different goals have lead to considerably different message passing semantics, even though both kernels are based on the Send-Receive-Reply model.

A.2.1. Message Addressing

Message addressing is the mechanism by which message destinations are specified. Both Harmony and Unison specify a many-to-one mapping between sending tasks and receiving tasks. Harmony messages are addressed to task IDs. The Harmony send primitive always sends to a specific task ID. The Harmony receive primitive may receive from *any* task or from a specific task ID, thus Harmony also allows a one-to-one mapping. Unison messages are addressed to a combination of ports and tasks. The Unison send primitive specifies a particular port and a particular task as the destination of a message. The Unison receive primitive merely specifies on which port(s) it is willing to receive messages: the receiver cannot specify a particular sending task. Unison does not support one-to-one mapping since, even if only one port is specified, many tasks may still send to this port. This presents difficulties in the implementation of vultures as explained in Chapter 3. A one-to-many mapping, as in broadcast communications, is not supported by either Harmony or Unison but is not relevant to the programming paradigm.

A.2.2. Message Screening

Message screening is the ability of a task to only accept messages from a specified set of other tasks. Harmony has no message screening other than its ability to receive specifically from a single designated task, temporarily ignoring[3] other sending tasks.

---

3 The *ignored* messages are queued.

Since Unison is intended to support Ada tasks, it provides message screening. The receiver of a message specifies the set of ports on which it is willing to accept messages while messages on other ports are temporarily ignored. While this feature is very useful when implementing Ada's *select* operation, the programming paradigm does not require message screening. The ability for Harmony and Unison tasks to reply to messages out of order is in many ways similar, but more general, than Ada's 'select' mechanism.

A.2.3. Message Timeouts

Message passing primitives may include timeouts which specify a maximum elapsed time before the communication is considered a failure. Harmony has no timeouts on its message passing primitives. Unison defines timeouts for the send and receive primitives, however timeouts are only implemented[4] for the receive primitive. A timeout of FOREVER can also be specified. In the programming paradigm, timeouts are useful but not necessary for the implementation of watchdogs. In the absence of timeouts, a more awkward mechanism using a timing task (and possibly couriers) is required to send a timeout message to the watchdog when the timeout expires.

A.2.4. Error Handling

The only error of concern in the context of the programming paradigm is when one correspondent is terminated during a message transaction. This can occur if a receiving task is terminated while a corresponding sending task is blocked. In both Harmony and Unison the sending task becomes unblocked and a returned error code indicates that the recipient task does not exist (as if the sender had tried to send to an invalid task ID). In

---

4 Timeouts on the *send* primitive may be implemented in a subsequent release.

Harmony, a similar sequence occurs when a task executes a receive-specific and the correspondent terminates. In Unison, a task receives on a particular port and a port cannot be destroyed so this situation never occurs. Since the reply primitive is non-blocking a terminated correspondent does not affect the blocking characteristics and the primitive merely returns an error code in both Harmony and Unison.

A.2.5. Other Primitives

Unison provides an extra primitive, *transfer*, which provides no additional functionality. The transfer primitive provides bi-directional data transfer between the invoking task and a specified task. The primitive can only be invoked when the specified task is reply blocked on the caller. This facility is sometimes used when the size of a message varies significantly from transaction to transaction and the receiving task cannot predict how large a message will be received. The same effect can be achieved with an additional message. Debuggers can also make use of this type of primitive. A task being debugged sends a debug message to the debugger which then has access, via the transfer primitive, to the entire address space of that task. A more effective debugger would distribute low level debugging facilities across processors, each of which has direct access to all tasks on its processor. The debugger interface would reside on a separate processor and communicate with the low level debugging facilities via messages. The programming paradigm does not utilize the transfer primitive to encourage simpler, more general solutions to the above problems.

Both Harmony and Unison support a non-blocking receive primitive. Harmony provides the *try-receive*, which is equivalent, in Unison, to a receive with a timeout of zero. When try-receive is used, an error code is returned to the receiving task if no

message is ready. The programming paradigm does not make explicit use of the try-receive primitive or its Unison equivalent since it often creates a polling situation. However, the mechanism may be used to priorize incoming messages.

A.3. Task Creation and Destruction

Harmony provides a simple **create** primitive that creates a new instance of the specified task. Unison provides an extended create primitive that allows initial parameters to be passed to the newly created task. This mechanism is useful for passing initialization information to sub-tasks that would otherwise require explicit communication as in Harmony. Explicit communication is necessary for task synchronization if the parent task needs to know when the child has finished initializing. In this case the initialization message in Harmony serves a dual purpose and a synchronization message is required in Unison. Differences in task creation mechanisms do not affect the programming paradigm, they merely affect the implementation of task initialization.

Harmony and Unison both support recursive, synchronous task destruction. When a task is destroyed, all its descendent tasks are destroyed first. This feature is particularly useful for implementing task groups, such as servers, in which all children tasks must be destroyed when the parent is destroyed.

A.4. Interrupt Mechanisms

Tasks can be considered either send-blocked or receive-blocked on an interrupt. The transaction between an interrupt service routine (ISR) and a notifier can be mapped onto the Send-Receive-Reply message passing model as follows. The notifier *logically* sends a

request-for-interrupt to the ISR and blocks. The ISR *logically* replies to the notifier when the event occurs and does not block. The ISR *implicitly* receives the request-for-interrupt but is never receive-blocked. This approach is appealing since the ISR can then be considered a special form of task. Alternatively, the notifier can be thought of as receiving events from an ISR. In this case the notifier never replies to the ISR, violating the Send-Receive-Reply model. Both Unison and Harmony consider tasks receive-blocked on interrupts but differ in their implementation.

In Harmony, interrupts are handled by a first level interrupt handler then passed on to an awaiting notifier task. The notifier must be blocked on the _Await_interrupt() primitive to accept an interrupt event. This mechanism is best suited for situations where the notifier handles the interrupt and the ISR merely passes the interrupt to the notifier. The difficulty with a separate primitive for receiving messages and receiving interrupts is that a complex task structure is needed to implement a buffered notifier as shown in Figure A.1. Furthermore, a message is required for each interrupt[5] and if interrupts occur faster than messages can be passed to a buffer administrator, events may be missed.

---

5 A message is not strictly required if the programmer is willing to use shared memory to communicate between the courier and the notifier.

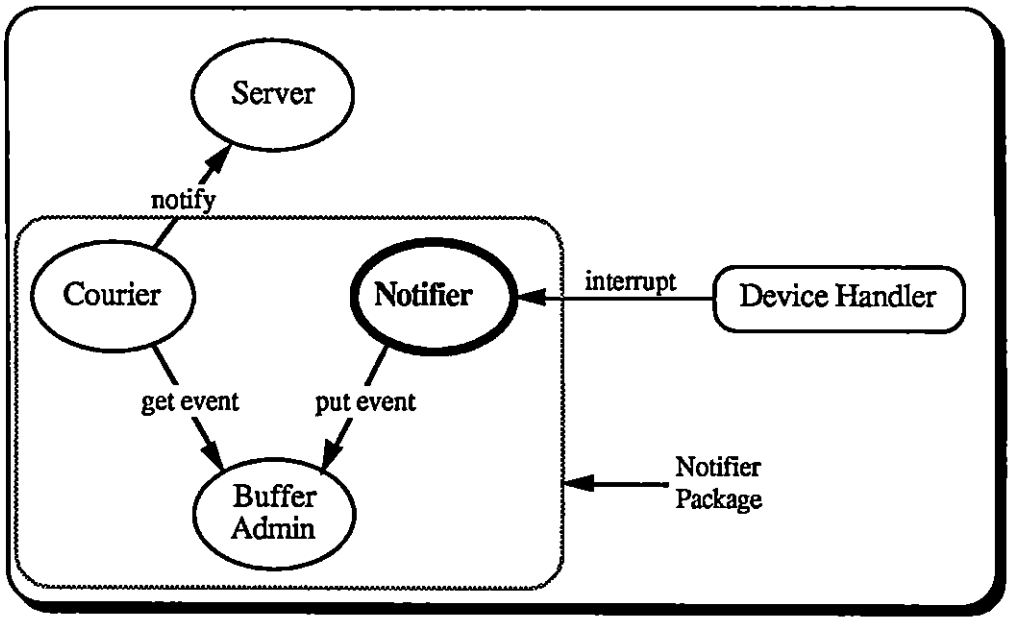


Figure A.1 — Buffered Notifier in Harmony

Unison tasks receive events from ISRs via the receive primitive. The ISR signals the task when an event occurs. This differs from the Harmony implementation in that the single receive primitive allows the notifier to receive *either* messages or events. A buffered notifier is simplified as shown in Figure A.2. The simpler task structure of this scheme enhances interrupt performance since there are no actual messages involved in the process of buffering events which may be critical when handling devices that have high burst rates.

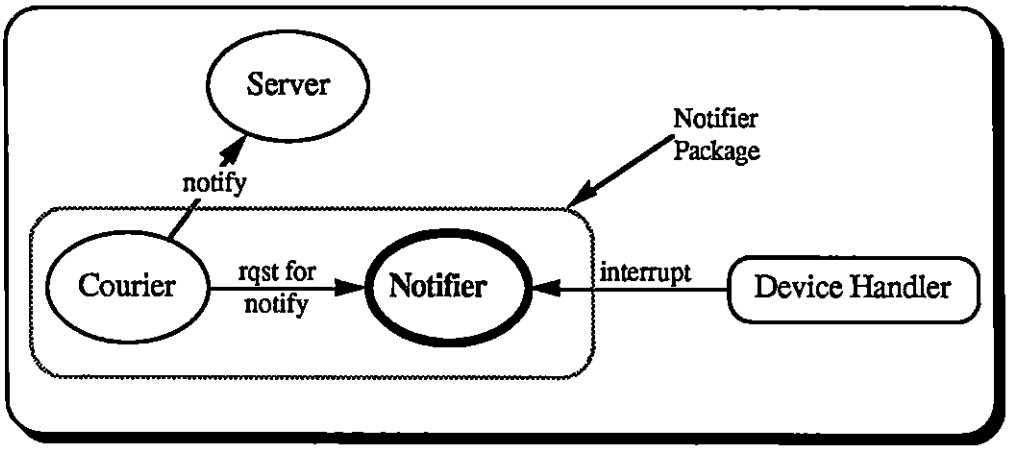


Figure A.2 — Buffered Notifier in Unison

A.5. Server Support

Servers are an integral part of the programming paradigm and operating system support influences their ease of implementation. Connection-based servers in particular require client-server connections, automatic close messages for connections, and a Name Server.

Harmony provides explicit support for connection-based servers. Connection managing library routines standardize and simplify the manipulation of server-resident connection tables. Each open connection is guaranteed to be closed if the client owning the connection is destroyed. A *directory server* provides mapping from symbolic names to server task IDs. The directory server is tightly bound to the _Open() primitive which is used by a client to open a connection with a server. When a connection is *opened,* a message is sent to the directory server to determine the ID of the named server. If the ID is found, then a second message is sent to the server to open the connection.

Unison does not currently provide support for connection based servers[6], however

support is expected to be available in the near future. Unison does provide a general name server that allows any task to register a symbolic name in conjunction with a value, usually its task ID. This can be used by servers or other tasks to make their IDs available to other tasks.

A.6. Syntax of the Send-Receive-Reply Primitives

The syntax of the Send-Receive-Reply primitives for Harmony and Unison are given here to clarify some of the issues discussed in this appendix. The code templates given in Chapter 3 also assume knowledge of the Unison syntax.

A.6.1. Harmony Syntax

Harmony has four primitives for message passing. A request or reply message is a variable length contiguous block of storage, the first two bytes of which are an unsigned integer specifying the length. Pointers to the message variables are passed to the primitives. A task ID is an operating system defined type whose contents are unavailable to the programmer.

TaskId **_Send**(rqst, rply, id)

Sends the message *rqst* to the task specified by *id*. If the message transaction succeeds, then *rply* is updated with the replied message and the ID of the correspondent is returned. If the transaction fails, zero is returned.

---

6 MTI indicates that Unison *does* provide support for connection-based servers, however the support routines are as yet undocumented.

TaskId **_Receive**(rqst, id)

Blocks the invoking task until a message is received from the task specified by *id*. If *id* is zero, the invoking task blocks until a message arrives from any task. If the message transaction succeeds, *rqst* is updated with the received message and the ID of the correspondent is returned. If the transaction fails, zero is returned.

TaskId **_Try_receive**(rqst, id)

This is a non-blocking receive primitive. If a message is pending and is from the task specified by *id*, or an *id* of zero is specified, then *rqst* is updated with the received message and the ID of the correspondent is returned. If a message is not pending, or no pending messages match the specified *id*, zero is returned.

TaskId **_Reply**(rply, id)

Replies the message *rply* to the task specified by *id* which must be reply-blocked on the invoking task. If successful, the ID of the correspondent is returned, otherwise zero is returned. Reply never blocks the invoking task.

A.6.2. Unison Syntax

Unison has four primitives for message passing. Messages and task IDs have similar properties to their counterparts in Harmony. Unison defines, for each task, a set of ports which can be used to receive only selected messages. Unison also implements timeouts for the receive primitive.

TaskId **_send**(rqst, rply, id, port_number, timeout)

Sends the message *rqst* to the task specified by *id*. *Port_number* specifies the port on the receiver's end through which the message will be routed. Timeouts are not implemented in Unison release 1.1 and must have a value of FOREVER. If the message transaction succeeds, *rply* is updated and the ID of the correspondent is returned. If the transaction fails, zero is returned.

TaskId **_receive**(rqst, portset, sigset, rec_portsig, timeout)

This primitive is used either to receive a message from a task or a signal from an interrupt handler. Messages are screened according to the mask *portset*, which indicates on which ports the task is willing to receive messages. Similarly, *sigset* indicates which signals the task is willing to accept. The *timeout* argument specifies the maximum time that the invoking task is willing to wait for a message or signal. A timeout value of FOREVER indicates that the task is willing to wait until a message or signal arrives, regardless of how long it tasks. A timeout value of zero is equivalent to Harmony's _Try_receive. If the message transaction succeeds, *rec_portsig* is updated by the operating system to reflect which port or signal was used during the transaction, *rqst* is updated to the received message, and the ID of the correspondent is returned. If the transaction fails, zero is returned.

TaskId **_reply**(rply, id)

Replies the message *rply* to the task specified by *id* which must be reply-blocked on the invoking task. If successful, the ID of the correspondent is returned, otherwise zero is returned. Reply never blocks the invoking task.

TaskId **_transfer**(src, dest, count, id)

This primitive provides bi-directional data transfer between the invoking task and the task specified by *id*. The task specified by *id* must be reply-blocked on the invoking task. *Count* bytes are transferred from the area pointed to by *src* to the area pointed to by *dest*. If the transaction is successful, the ID of the corresponding task is returned, otherwise zero is returned.

A.6. Summary

While the essential characteristics of Harmony and Unison are the same, there are minor differences that affect the implementation of various task types. Overall, neither operating system is inherently *better* than the other, since they each have their advantages. Harmony's task-based message addressing is better suited to the programming paradigm than the port-based addressing of Unison since the latter scheme does not permit one-to-one mapping between senders and receivers. Message screening is neither an advantage nor a deficiency with respect to the programming paradigm, however, it adds unnecessary complexity to the send and receive primitives of Unison. Unison's message timeouts simplify the implementation of watchdogs and reduce the task and message overhead of timing operations. Unison's task creation mechanism provides slightly easier and more

consistent communication between a parent and child task if synchronization at initialization is not required. Unison's ability to receive both interrupts and messages using a single primitive simplifies the implementation of buffered notifiers and improves their potential response. Finally, Harmony's server support is crucial to the development of connection-based servers. Both Unison and Harmony are continually evolving according to customer demand and any deficiencies with respect to the programming paradigm may be remedied in future releases.

Appendix B

The Forward Primitive

B.1. Introduction

B.1.1. Motivation

This appendix is an attempt to answer some questions about the necessity of the forward primitive. The questions are: do we *need* Forward in theory or practice, what are the benefits of Forward, what are the problems associated with Forward (in theory and in practice). It appears that Forward is an emotional issue, not unlike that of *your favourite editor*. However, Forward involves more serious theoretical and practical issues than does an editor since it affects not only the person using it, but all those that need to understand the design and implementation of a system using Forward.

B.1.2. Why is Forward Interesting

Forward is interesting because it is an issue that straddles the boundaries between theory and practice. In theory, it seems Forward is not necessary. In practice however, Forward appears very useful and may be an important efficiency consideration.

B.1.3. Why is Forward Not Discussed

The literature contains little discussion of the issues related to Forward. It may be that it does not warrant paper (i.e. potential authors firmly believe their own opinions and think that others must obviously agree); it may be that it does not seriously affect either system design or application design; it may be that it is a less theoretical issue and of concern only to areas of industry that are too busy to publish. It is also true that there is not a large number of published systems that use the Send-Receive-Reply communication primitives, where Forward is a simple extension.

B.1.4. Scope

This appendix does not evaluate, in depth, systems using Forward and does not judge the actual performance and other pragmatic issues. This appendix is an attempt to evaluate the usefulness of Forward from a personal point of view in light of the author's experience, discussions with others, and from the literature.

B.2. The Forward Primitive

B.2.1 What is Forward

Forward is an operating system primitive that acts on messages. There are a few variations of Forward implementation. Although the basic meaning is similar, the manner in which Forward is used may change depending upon the semantics. This section describes two possible semantics.

1) **FORWARD-1**(task_A, task_B)

A task has received a message from task **A** and wishes to forward that message to task **B**. The overall effect is the same as if the sender had performed a Send to task **B**. Note that the message cannot be altered by the forwarding task and the forwarding task does not send-block on the destination task. Figure B.1 illustrates the *effect* of task A sending to task F, then task F forwarding the message to task B.

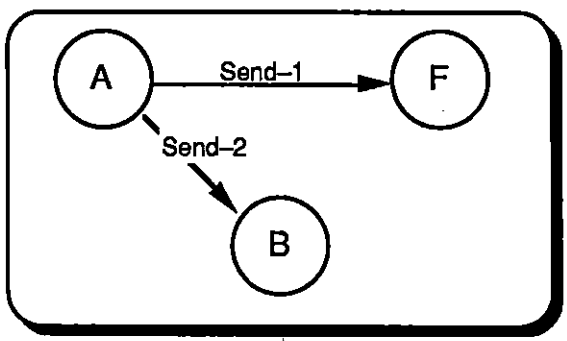


Figure B.1 — Forwarding Using FORWARD-1

2) **FORWARD-2**(message, task_A, task_B)

This type of Forward (as used in Thoth [CHER79], Verex and the V-Kernel [CHER84]) is similar to the FORWARD-1 with the exception that the forwarding task may change (or completely replace) the message contents. The blocking diagram for FORWARD-2 is shown in Figure B.2.

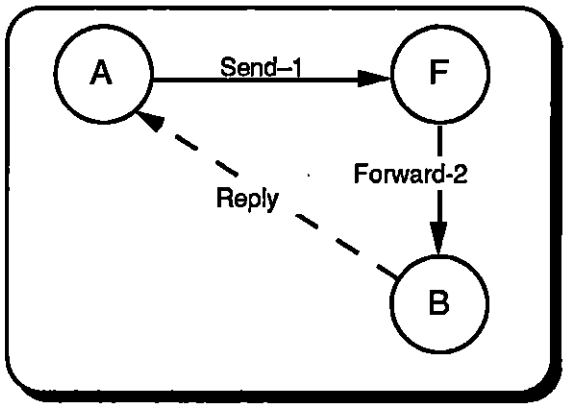


Figure B.2 — Forwarding Using FORWARD-2

B.2.2 Is Forward Necessary?

From a theoretical viewpoint, Forward is unnecessary since the same effect can be achieved via the existing S-R-R primitives. See section B.5.3 for an example of how this can be done. From a practical standpoint, it may also be undesirable to include Forward in an operating system since a) it adds complexity to the operating system, and b) it adds an extra primitive. The second may seem insignificant, however if one examines the systems available today, it is not at all apparent that a large number of system primitives is a good thing.

B.3. Forward in Context

B.3.1 Classes of Operating Systems

Only synchronous message-based systems using the Send-Receive-Reply model are considered. Asynchronous Send-Receive message-based systems cannot use Forward (as described in this appendix) since the sender does not remain blocked after the *receive*

primitive has been executed, thus cannot become blocked on the forwardee. Forward has no meaning in the context of asynchronous message-based systems since there is no blocking on messages. Obviously, Forward is not an issue in non-message-based systems[1].

B.4. Utility of Forward

B.4.1 How Forward is Used

In a typical message based system with blocking message primitives it is necessary to ensure that certain types of tasks (e.g. servers) do not block while they are processing client requests. These server tasks may periodically require services of some other task in order to satisfy the client request. If they send a request to another task, they would block, possibly delaying other important requests. The forward primitive allows a task to shift the blocking from itself to the original sender (e.g. the client).

The most fundamental class of problems solved by the forward primitive is that of hierarchical task structures communicating via messages (i.e. task A sends to task B which sends to task C which performs some operation and replies a result). This situation can arise in relatively simple servers, where a server requires the services of another server in order to satisfy a request. A courier can be used to prevent a server from blocking on another server as described in section B.5.2, however this approach increases complexity as well as message and task overhead. Alternately, if tasks are used to implement the objects of a class hierarchy (such as Smalltalk-80 [GOLD83]), then it is desirable for a task

---

1 In the scope of this discussion, Ada is considered to be message based.

receiving a request to perform the request if possible and if not, Forward the request to its super-class (parent task) where the same operation occurs. Using couriers in this situation can drastically increase overhead.

B.4.2 Problems with Forward

Forward does have some difficult aspects. The first problem arises when drawing task blocking diagrams. How should a forwarded message be depicted?. The arrow typically used for *send* is not appropriate since it implies blocking. A good notation is required here if Forward is to be depicted in blocking diagrams. A second problem (related to the first) is the static detection of deadlock. Normally, deadlock can be detected by searching the blocking diagrams for cycles, however, when Forward is involved, it is not as obvious. A third concern, is that Forward introduces the potential for livelock, that is, a pair (or sequence) of tasks can continually forward messages between themselves, each expecting the other to deal with the request. These three problems make it difficult to design a reliable system when it is composed of many interacting tasks forwarding messages.

From an implementation viewpoint, Forward presents more difficulty. In a system that supports task destruction or one in which message communication media is unreliable, the operating system must take care of many special cases when messages are part way through a Forward transaction and one of the correspondents is destroyed.

B.5. Alternatives to Forward

B.5.1 Avoidance

It is often possible, through careful design, to avoid the need for the forward primitive. In particular, when a server-worker model is used, a simple and effective method of avoiding Forward is presented in [GENT81]. The following two sections describe specific techniques for avoiding using Forward when hierarchies of communicating tasks are required.

B.5.2 Using Couriers

It is possible for a task to have a message sent on its behalf without causing the task itself to block. A server can create (any number of) **couriers** which perform the send on behalf of the server. The courier sends a message to the server requesting some message to be forwarded. The server **replies** the message it wishes forwarded, along with target information, to the courier. The courier then **sends** this message to the target server. Thus, the courier remains blocked on behalf of the server. When the target server replies, the courier **sends** the reply to the server. See Figure B.3 below.

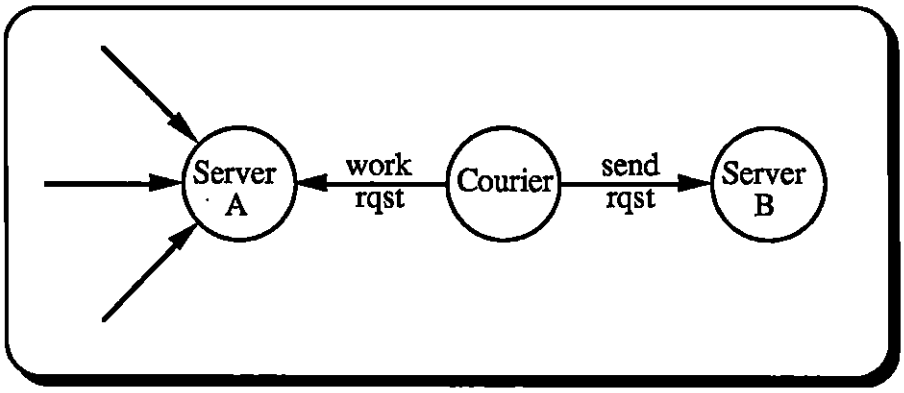


Figure B.3 — Using Couriers to Avoid Blocking

Although couriers solve (in theory) the problem of blocking servers, in practice they tend to introduce unnecessary complexity. The designer must decide exactly how many couriers should be created and whether couriers are created at initialization or on demand. The allocation strategy is operating system and hardware dependent, thus rendering the server code less portable. The implementation also requires that significantly more state information be maintained (state of couriers, courier queues, etc.). Couriers consume memory resources and incur more message overhead than the forward primitive. The advantage of couriers is that the operating system can be simpler than one supporting Forward.

B.5.3 Simulated Forward

It is possible to simulate Forward in a system that implements the Send-Receive-Reply model. The primitives SEND, RECEIVE, REPLY and FORWARD can be implemented on top of the primitives _Send, _Receive and _Reply. The semantics of the primitives SEND, RECEIVE and REPLY are identical to those of the original operating

system with the restriction that all messages must have a message result field (MSG_RESULT) and that the value MSG_FORWARD is unique and not used for other purposes. The semantics of FORWARD(senderID, receiverID) are: the forwarding task has received a message from task 'senderID' and forwards this message to task 'receiverID'; the forwarding task is not blocked, the sending task becomes blocked on the task 'receiverID'. The difference between FORWARD and an operating system based Forward is that the message forwarded cannot be changed by the forwarding task. The C source code for the simulated Forward is shown in Listing B.1. Figure B.4 depicts the blocking diagram for simulated Forward.

Listing B.1 — Simulating Forward

```
#define RECEIVE( rqst, tid )    _Receive( rqst, tid )
#define REPLY( rply, tid )      _Reply( rply, tid )

TaskId SEND( rqst, rply, tid )
StdRqst *rqst;
StdRply *rply;
TaskId  tid;
{
    TaskId  correspondent;

    correspondent = _Send( rqst, rply, tid );

    /* Allow for many possible forwards */
    WHILE( rply->MSG_RESULT == MSG_FORWARD &&
           correspondent == tid )
        /*
         * send the UNCHANGED rqst to the new correspondent
         * task
         */
        correspondent = _Send( rqst, rply, rply.MSG_FWD_TID );
    ENDWHILE

    return( correspondent );
}

TaskId FORWARD( from_tid, dest_tid )
TaskId  from_tid, dest_tid;
{
    StdRply rply;

    /* Set up reply fields as necessary */
    rply.MSG_RESULT = MSG_FORWARD;
    rply.MSG_FWD_TID = dest_tid;

    return( _Reply( rply, from_tid ) );
}
```

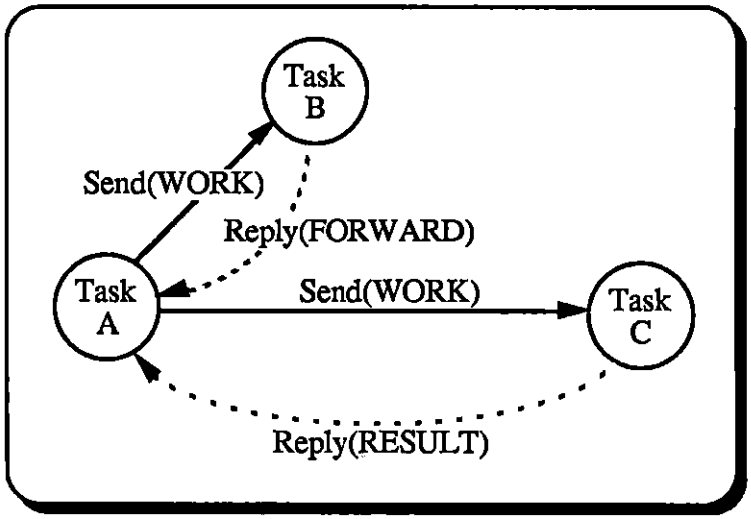


Figure B.4 — A Sends Request to B, B Forwards to C

Figure B.4 illustrates the following sequence of events: Task A sends a request to task B. Task B realizes that task C is really the task that is capable of dealing with the particular messages and so replies a FORWARD message to task A. Task A then re-sends its message to task C. Task C performs the requested work and replies the result to task A.

B.6. Performance Issues

For the purposes of this performance discussion, a message is defined to be a single transfer of data. Thus for a system that implements a send-receive-reply cycle, two messages are required. The first is the request message and the second is the reply message. In the above blocking diagrams, each arrow represents two messages (based on the S-R-R model).

The performance cost of couriers may be significant compared to Forward. Six messages are required (plus one during initialization) as follows:

courier sends to server A /* during initialization */

client sends to server A

server replies to courier

courier sends to server B

server B replies to courier

courier sends reply to server A

server A replies to client

In addition, if many couriers are required in the system the memory overhead of the courier tasks may be significant. Note that each task requires at least a stack and task descriptor. Consider a system of 100 servers, 50 of which require couriers. If each server uses on average 10 couriers at a time, then a total of 500 couriers are used. A conservative memory estimate for the size of a courier is 100 bytes for the task descriptor and 400 bytes for the stack. Thus 500 tasks x 500 bytes/task or 244 Kilobytes of memory will be consumed by couriers.

Simulated Forward exhibits better performance than couriers. A simulated Forward requires four messages as follows:

client sends request to server A

server A replies 'FORWARD' to client

client sends forwarded request to server B

server B replies to the client

In addition to requiring two less messages, the simulated Forward has the advantage of requiring no extra task resources.

In order to compare the above two methods to a real Forward, we must consider the type of system involved. If the system is a tightly coupled multiprocessor (e.g. Harmony [GENT83]), then it may be possible to realize Forward in three message transactions. This assumes that the sending task descriptor can be manipulated directly by the processor doing the Forward to ensure that the sender is blocked on the correct task. On the other hand, if the system is truly distributed and the kernels cannot directly modify each other's task descriptors, it is necessary to send a message to the kernel of the sending task to ask it to make the sender send-blocked on the new task. Now we must consider the message passing semantics available to the kernel itself. If the kernel uses the S-R message model (as opposed to the S-R-R model), then the kernel can perform the Forward in four messages (best case) as follows:

client sends to server A
A forwards to B:
    kernel sends update to client processor
    kernel sends request to server B
server B replies to client

A problem with the above analysis is that we have to make certain assumptions about the kernel implementation. It may well be that messages between kernels are significantly faster than regular messages. It may also be that the kernel is also required to use S-R-R in which case an implementation dependent analysis is necessary.

As a result of this analysis, the simulated Forward is the best approach, since it does not increase the complexity of the operating system and has minimal message overhead.

B.7. Conclusions

It should be apparent from the preceding discussion that Forward is unnecessary and indeed, undesirable, to incorporate into a message-based operating system. Forward complicates the detection of deadlock, introduces unnecessary complexity, and the same effect can often be achieved using couriers. In the event that a system actually benefits from using Forward, then the simulated Forward should be sufficiently powerful and efficient for most situations.

Appendix C

A Suggested Standard C Format
or
"C'ing Clearly"

Programming style is important in all commercial software projects since the style directly affects the readability of the code. Unfortunately, C programming style is often the subject of heated religious debate. One of the key issues is the placement of curly braces. Since everyone seems to have their favorite format for curly braces, enforcing a standard format is difficult. A desirable solution would be to rid the language of curly braces entirely. The following C coding convention was employed throughout the KAON Factory Application and was found to lead to more compact and readable code.

To eliminate the curly braces from C without changing the compiler, a number of macros are required to re-define the standard C constructs. Each of these macros is a direct replacement for one of the C control structures. For example, the **while** statement is replaced by the **WHILE** macro. Although there are some minor limitations of this convention, the readability benefits outweigh the disadvantages. Listing C.1 shows a short piece of regular C code and Listing C.2 shows the same code using this convention.

Listing C.1 — The Author's Previous Coding Style

```
/* Regular C Code */
int foo()
{
    for( j = 0, i = 0 ; i < 100 ; i++ )
    {
        while( bar( i ) < MAX_BAR )
        {
            j = process( i, foobar( i ) );
        }

        switch( j )
        {
            case 1 :
                /* processing for this case */
                break;
            case 2 :
                /* processing for this case */
                break;
            default :
                /* default processing */
                break;
        }
    }
}
```

Listing C.2 — C'ing Clearly Coding Style

```
int foo()
{
    FOR( j = 0, i = 0 ; i < 100 ; i++ )
        WHILE( bar( i ) < MAX_BAR )
            j = process( i, foobar( i ) );
        ENDWHILE

        SWITCH( j )
            CASE( 1 )
                /* processing for this case */
            ENDCASE
            CASE( 2 )
                /* processing for this case */
            ENDCASE
            DEFAULT
                /* default processing */
            ENDCASE
        ENDSWITCH
    ENDFOR
}
```

Listing C.1 is just one of the many ways that the code fragment could be formatted. The convention used in Kernighan and Ritchie's "The C Programming Language" [KERN78] places opening curly braces on the same line as the control structure as in Form-1 shown below.

```
if( expr ) {
    statement;
}
```
Form-1

```
if( expr )
    statement;
```
Form-2

```
if( expr )
    stmt1;
    stmt2;
```
Form-3

Kernighan and Ritchie also use curly braces only when necessary, omitting them when there is only a single statement within the control structure, as shown in Form-2. Although this form is often used, it leads to inconsistent format since sometimes a control structure will be followed by curly braces and sometimes not. This creates a readability problem, due to the inconsistency and error-prone programming. Errors can easily creep into a program when a programmer decides to add statements and forgets to add curly braces as in Form-3. Programmers complain that always including curly braces, even when they are not required wastes vertical whitespace and spreads out the program too much.

C without curly braces produces code that does not suffer from these types of problems. There is only one form for the above **if** statement:

```
IF( expr )
    statement;
ENDIF
```

If statements are added within the **if** statement, then the new code looks like:

```
IF( expr )
    statement1;
    statement2;
ENDIF
```

The two code fragments are consistent and readable.

The macro definitions are contained in a single include file. This file is included in each C program using the macro definitions. The macro definition file is listed entirely in Listing C.3.

Listing C.3 — Macros for C'ing Clearly Constructs

```
/*
 * Re-definition of C constructs for better readability
 */

/*
 *  FOR ( expr ; expr ; expr )
 *      stmt;
 *      stmt;
 *  ENDFOR
 */
#define FOR(x)  for(x) {
#define ENDFOR  };

/*
 *  IF ( expr )
 *      stmt;
 *      stmt;
 *  ELSEIF ( expr )
 *      stmt;
 *      stmt;
 *  ELSE
 *      stmt;
 *      stmt;
 *  ENDIF
 */

#define IF(x)       if(x) {
#define ELSE        } else {
#define ELSEIF(x)   } else if(x) {
#define ENDIF       };

/*
 *  WHILE ( expr )
 *      stmt;
 *      stmt;
 *  ENDWHILE
 */
#define WHILE(x)    while(x) {
#define ENDWHILE    };
```

```
/*
 *  SWITCH ( expr )
 *      CASE ( constant )
 *          stmt;
 *          stmt;
 *      ENDCASE
 *
 *      CASE ( constant )
 *          stmt;
 *          stmt;
 *      <<drop through to next case>>
 *
 *      CASE ( constant )
 *          stmt;
 *          stmt;
 *      ENDCASE
 *
 *      DEFAULT
 *          stmt;
 *          stmt;
 *      ENDCASE
 *  ENDSWITCH
 */
#define SWITCH(x)          switch(x) {
#define ENDSWITCH               };
#define CASE( const )     case const :
#define ENDCASE              break;
#define DEFAULT             default :

/*
 *  REPEAT
 *      stmt;
 *      stmt;
 *  UNTIL ( expr )
 */
#define REPEAT      do {
#define UNTIL(x)    } while( !(x) );
```

The semantics of the **repeat** statement have changed from a repeat-while loop to a repeat-until loop. This is because there are two different meanings for the **while** statement in C. **While** can indicate the start of a while loop or the end of a **repeat** loop. The repeat-until disambiguates the use of the macro **WHILE.** This change is considered an improvement over regular C.

In the switch statement, the constant in each case selector is enclosed in parenthesis. While this deviates from regular C, it was considered necessary to maintain consistency. Regular C permits the selector constant to exist with or without surrounding parenthesis. This seems inconsistent with control structures such as **if** and **while** which require parenthesis. This change is considered an improvement over regular C.

The macro statements have an **ELSEIF** construct to multiple if statements in a sequence. The example in Listing C.3 shows the use of ELSEIF. This change is neither an advantage nor a disadvantage.

The main disadvantage of this mechanism is that errors, related to parameters to macros, seem difficult to track down. One difficulty that arose a couple of times was a missing closing parenthesis for an IF (or other) macro. While this doesn't seem so hard to track down, the Sun C pre-processor complains about the actuals (the parameters to the macro) being too long and the error message indicates that error occurred somewhere long after the start of the erroneous macro. The difficulty is searching for the unterminated macro. If the compiler indicated the line number on which the macro started, then it would be simple to fix. This type of problem is restricted to the C compilers that use a BSD based C pre-processor. Other compilers, such as the LightSpeed C compiler for the Macintosh report the macro start-line as the source of the error.

The end result is that a few minor difficulties are introduced but the language is more consistent and readable. Curly braces can still be used to indicate nested blocks and the scope of C functions. Additionally, regular C code can be mixed with these macros. While this may seem unusual, it is necessary to support both since software projects may have considerable regular C code that does not warrant conversion and software may be obtained from external sources that do not use the macros.

A tool to convert from the regular format to the macro format would be quite useful. This tool could be an enhancement of the BSD 'indent' program. Additionally, a C parser that directly understands the new format could be implemented, however the cost to benefit ratio of this program would be low.

Appendix D

Makefiles Without Fault-Prone Redundancy

There is a fundamental problem with the way the standard makefile dependencies relate to the *#include* directives in a C source file. All files that are included in the C source file are dependencies of the C source file. That is, if any one of those include files changes, the C source file must be re-compiled. Because of this, it is necessary to specify all the dependencies in the makefile. Now we have a problem. If the makefile is generated manually, which is often the case, there is redundant information. If the *#include* directives change in any way, it is vital to reflect these changes in the makefile. However, since these changes are being made by a human programmer, the possibility of error exists. An errors such as this is extremely difficult to detect, especially in a large system with many programmers. The cost of this type of error can easily lead to work-weeks of lost programmer time since bugs that have been fixed may still appear in the resulting executable if one particular compilation unit has not been re-compiled.

Many Unix systems provide an automatic mechanism for generating makefiles. One method is via a program called 'makedepend' or something similar. This program scans C programs for *#include* directives and creates dependency lists that are acceptable for

'make'. Some BSD Unix systems have an option to the C compiler that performs exactly the same function. With this mechanism, it is common to have a 'depend' target in the makefile.

While these two mechanisms are often adequate and do indeed solve the problem for single programmer projects, a problem exists for multi-programmer projects. The question arises: when should the system builder invoke 'makedepend'? The best solution is to **always** invoke make depend, since there is no way for the system builder to know if any programmers have changed any *#include* directives. This has the undesired effect of performing a lot of work since *all* source files must be scanned for *include* directives. This can take a long time, even if recompilation is not necessary. This solution also creates new makefiles (or modifies the existing makefile) each time 'make depend' is executed. This can cause problems for version control mechanisms since the makefile has changed, but not necessarily due to a change in the source files. This new makefile should be placed under version control even though there may have been no changes to the source code.

A second solution[2] makes use of makefile macros instead of an external dependency list creation program. For this approach, the C source files contain no *#include* directives at all. Instead, the makefile has a macro for each compilation unit that lists all C files and all header files (the constituent files) that make up the compilation unit. A trivial program, 'mkinc', is used by the macro to create a single C file that contains nothing but include directives, listing the constituents that make up the compilation unit. The resultant C file is then compiled to produce one component of the system.

In this solution the list of include files exists only in the makefile. When make examines the constituent macro, it generates a new include file only if the target is older

---

2 Resulting from discussions with [LOCK85].

than one of the constituent files and needs to be re-compiled. There are no new files created or updated if all targets are up to date, and the makefile can be under source code control with no difficulties.

An additional benefit of this solution is that the project manager can be in charge of the makefiles. This allows the project manager to ensure that programmers do not *#include* header or source files that are intended to be hidden from their modules. This controlled isolation is important to ensure module independence, encapsulation, and other software engineering principles.

Another key feature of the mkinc approach is that it extends the existing relationship between files and modules. When using make or make depend, the compilation unit is a single C source file. By definition, this file defines the scope of static variables defined outside any functions. This forces programmers to keep all functions that need to share module data in the same file. If the functions are divided into separate files, then the module data must be made global, and becomes part of the global name space. This is a very undesirable situation. When using mkinc, a module is a compilation unit, comprised of possibly many C source files. The programmer is free to separate the module functions appropriately, yet still share module data. A hidden feature of this division is that revision control schemes are often much faster when smaller files are revised.

An apparent deficiency of this technique is that the makefile must undergo many small changes when software projects are in their infancy, causing the project manager excessive 'trivial' work updating the makefiles. This can be avoided in two ways. Include stubs can be created in advance, forming a skeleton framework of include files before the modules are implemented or completed. Alternately, each programmer can be working on a branch

of the source tree that has its own makefile. The programmer will 'checkout'[3] the makefile for the particular branch of the source tree and will use it as an evolving makefile until that part of the tree becomes stable.

Following are two listings. Listing D.1 is the 'mkinc' program that generates the C #include files from the constituents and the Listing D.2 is a sample makefile, showing the use of 'mkinc'.

---

3 When version control software is used, it is common to *checkout* a file for use by a single person, preventing others from making concurrent changes.

Listing D.1 — The 'mkinc' Program

```
/*
 * mkinc - make a file of includes that can be compiled in
one
 * go.
 * Usage:  mkinc files... >IncludeFileName.c
 */
int main( argc, argv )
int argc;
char **argv;
{
    /* Skip argv[0] */
    argc--;
    argv++;

    /* print argv[1] .. argv[argc] */
    while( argc-- > 0 )
    {
        /* output a '#include' line */
        printf( "#include \"%s\"\n", *argv++ );
    }
    return( 0 );
}
```

Listing D.2 — A Sample Makefile

```
#
# Makefile for Producer-Consumer Sub-system
#

# Directory Macros
SYSINC          = /usr/include
SUBSYS          = /usr/BigSystem/Development/ProducerConsumer
H    = $(SUBSYS)/h
SRC = $(SUBSYS)/src

# Producer constituents
PRODUCER = \
        $(SYSINC)/stdio.h \
        $(H)/prod_com.h \
        $(H)/producer.h \
        \
        $(SRC)/producer.c

# Consumer constituents
PRODUCER = \
        $(SYSINC)/stdio.h \
        $(H)/prod_com.h \
        $(H)/consumer.h \
        \
        $(SRC)/ consumer.c

# Main target
main:           producer consumer

producer:       producer.o
    $(CC) producer.o -o producer

consumer:       consumer.o
    $(CC) consumer.o -o consumer

producer.o:     $(PRODUCER)
    $(MKINC) $(PRODUCER) > producer.c
    $(CC) producer.c

consumer.o:     $(CONSUMER)
    $(MKINC) $(CONSUMER) >consumer.c
    $(CC) consumer.c
```

Appendix E

Parameterized Types in C

Parameterized (or polymorphic) types have one or more parameters [CLEA86]. A parameterized type with arguments is called an *instance* of the parameterized type. Parameterized types are often useful when a standard type is required, for which there are well-defined operations, but additional application specific information needs to be attached.

For example, consider a list element type that has the operations *addToHead* and *removeFromTail*. The list operations are independent of the application specific fields of the data type, and only require access to the fields necessary for list manipulation. We may wish to have a list element type that has the appropriate list manipulation data as well as an array of 100 characters. The list operations are not interested in the application-specific array and the application is not interested in the list data manipulation data. One method of achieving this is to force the programmer to explicitly include the list information in each data type that is to be used as shown in Listing E.1.

Listing E.1 — Explicit List Type

```
/* List Library Header */
typedef struct { ... } List;
typedef struct { ... } ListNode;

/* C ProtoType (only available in ANSI C Compilers) */
ListReturnCode      addToHead( List list, ListNode *node );

/* Application Code */
typedef struct
{
    ListNode          listHeader;
    char              string[100];
    double            someBigNumber;
    int               someSmallerNumber;
} myListNode;

main()
{
    List              list;
    myListNode        node;

    ListInit( &list );
    getString( node.string );
    node.someBigNumber = getBigNumber();
    node.someSmallerNumber = getSmallerNumber();
    addToHead( &list, (ListNode *) &node );
    ...
}
```

A better method is to use the macro facility in C to realize parameterized types. With this approach the builder of the library routines (such as the list routines in the previous example) would provide macros in a header file to enable application programmers to build a type without being forced to place the proper types in their proper places or see the internal representation of the list nodes. Listing E.2 illustrates this approach.

Listing E.2 — Parameterized List Type

```
/* List Library Header */
typedef struct { … } List;
typedef struct { … } ListNode;

#define standardListNode( applDataType ) \
    struct {                             \
        ListNode        listHeader;      \
        applDataType    applData;        \
    }

/* Application code */
typedef struct
{
    char    string[100];
    double  someBigNumber;
    int     someSmallerNumber;
} applicationData;

typedef standardListNode( applicationData ) myListNode;

main()
{
    myListNode  node;
    List        list;

    ListInit( &list );
    getString( node.applData.string );
    node.applData.someBigNumber = getBigNumber();
    node.applData.someSmallerNumber = getSmallerNumber();
    addToList( &list, (ListNode *)&node );
    …
}
```

The KAON Factory Application described in Chapter 6 makes use of this form of parameterized types for message structures. Unison requires the first two bytes of every message to contain the message length. By convention, following the message length field of a request message is an enumerated type which is the **MSG_TYPE** (or operand) field.

The rest of the request message is application specific. Also by convention, following the message length field of a reply message is an enumerated type which is the **RESULT** field. The message passing primitives are only interested in the length of a message and not the contents. The macros given in Listing E.3 satisfy both operating system requirements and convention requirements.

Listing E.3 — Message Definition Macros

```
#define STD_RQST( type ) \
    struct { uint_16 MSG_SIZE; type MSG_TYPE }
#define STD_RPLY( type ) \
    struct { uint_16 MSG_SIZE; type RESULT }
```

The macro **STD_RPLY** in Listing E.3 defines a structure that has an unsigned 16-bit integer field for the message size, as required by the operating system and an application specific field of type **type** for the message type, as required by convention.

VITA

Name:	Peter J.M. Baker
Place and year of birth:	Comox, B.C., 1959
Education:	University of Victoria, 1979-1984 B.Sc. in Computer Science University of Victoria, 1987-1988 M.A.Sc. in Electrical and Computer Engineering
Honours and Awards:	DEC Award of Merit, 1983 NSERC Postgraduate Scholarship I, 1987 NSERC Postgraduate Scholarship II, 1988 Advanced Systems Institute of B.C. Scholarship, 1987 President's Special Research Scholarship, 1987 President's Special Research Scholarship, 1988
Publications:	"An Overview of a Send-Receive-Reply Programming Paradigm", presented at the *Canadian Conference on Electrical and Computer Engineering*, November, 1988.

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis (the title of which is shown below) to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis

"A Programming Paradigm Based on the Send-Receive-Reply Task Communication Primitives"

Author [redacted]

PETER JOHN MARK BAKER PETER JOHN MARK BAKER

(Name in block letters)

DEC 28/88

(Date)

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-50155-3