CHARLOTTE:
DESIGN AND IMPLEMENTATION OF A DISTRIBUTED KERNEL

by

Yeshayahu Artsy
Hung-Yang Chang
Raphael Finkel

Computer Sciences Technical Report #554

August 1984

Charlotte:
Design and implementation of a distributed kernel

Yeshayahu Artsy
Hung-Yang Chang
Raphael Finkel

CONTENTS

Charlotte:
Design and implementation of a distributed kernel[1]

Yeshayahu Artsy
Hung-Yang Chang
Raphael Finkel

1. Introduction

This paper describes the architecture of the Charlotte distributed operating system designed for the Crystal loosely coupled multicomputer network [1]. Charlotte aims to provide a supporting environment to solve computationally intensive distributed problems. It includes unique inter-process communication mechanisms and process-management services. Applications running under Charlotte may have several closely coupled concurrent computational threads. Charlotte is responsible for assigning processors to processes and hiding details of inter-processes communication. Charlotte therefore fulfills several goals:

- to explore operating system design for multi-process applications.
- to study a new approach to communication paths and primitives.
- to maximize utilization of computational resourses while minimizing overhead. and
- to serve as a testbed for distributed algorithm design.

The network uses a token ring called *ProNet* [2], which currently connects 13 homogeneous node computers and several host computers. Each node is a VAX-11/750 with 1-2 MB main store. The hosts are VAX-11/750s and 780s running Berkeley Unix 4.2.[2] The network can be dynamically partitioned to run different operating systems and stand-alone application programs concurrently. Each node runs a basic software **nugget**, which provides low-level inter-node communication facilities [3].

The current version of Charlotte has many antecedents. Arachne (orginally called Roscoe) [4, 5] was first designed and built around a network of PDP-11/03 computers and subsequently rewritten for the network of PDP-11/23 computers connected by a broadband, 1 Mb/sec contention network. Inter-process communication in Arachne is based on the Demos operating system for Cray-1 [6]. Experience with Arachne resulted in the design and implementation of Charlotte version 1, which introduced different inter-process communication primitives to remedy perceived defects of Arachne. Full-duplex links replaced simplex links, synchronized message transfer replaced kernel buffering, and a more symmetrical send and receive replaced blocking receive and non-blocking send. This first version was written for the PDP-11/23 network. When Crystal became available in the summer of 1983, Charlotte underwent another change. Interprocess communication interfaces were modified, the kernel internal structure was redesigned, and a new inter-kernel protocol was built. We will present each of these new designs in detail.

This paper describes the general architecture of Charlotte version 2 while looking back to draw comparisons from previous versions. We will demonstrate how a simple but flexible inter-process communication design leads to complex kernel-level protocols. However, the complexity is measured by the number of compound conditions and anomalies that must be considered, not by time or space inefficiency. Section 2 provides an overview of Charlotte software, Section 3 describes kernel structure in general, and Section 4 elaborates on the kernel-level protocol design.

2. Overview of software architecture

Charlotte contains three levels of software (Figure 1). The innermost level is the kernel, which provides interprocess communication and simple process control. An identical copy of the kernel is running on each node. A process called the *KernJob* also exists on each node. The KernJob

[1] This work was supported in part by NSF grant MCS-8105904 and by DARPA contractr N00014-82-C-2087

provides process-control functions to other processes through the message-based inter-process communication interface. Failure of this level is considered nonrecoverable.

The second level consists of utility processes, which are part of Charlotte but need not reside on each node. For example, a *Starter*, which is the memory manager as well as the midwife of new child processes, may have several nodes under its purview. The following utility processes are currently available:

- Memory Manager (Starter)
- File Server
- Initial-linkup Server (Connector)
- Directory Server (SwitchBoard)
- Command Interpreter

A fully functional Charlotte installation needs at least one copy of each utility. Failures at this level are not considered fatal, so every effort is made to recover after loss of servers. We assume a server dies only because its node has failed. We can often reconstruct partial resource information governed by a failed server by querying kernel tables and can subsequently create a new server process to resume the service.

The third software level contains client processes enjoying the interprocess communication mechanism provided by the kernel and utility functions from various utility processes. Processes can be called into existence interactively through the command interpreter or through other processes.

```
+--------------------------------------------------------------+
|       CLIENT PROCESSES                                       |
|   +--------------------------------                          |
|   |                              |                           |
|   |   UTILITY PROCESSES          |                           |
|--------------------------------------------------------------|
|     Communication kernels:                                   |
|       +----------+               +----------+                |
|       | KERNEL 1 | ------...---- | KERNEL 2 | ---...         |
|       |          |               |          |                |
|       |   KJOB   |               |   KJOB   |                |
|       +----------+               +----------+                |
|            |                          |                      |
|       +----------+               +----------+                |
|       | KERNEL.. | ------...---- | KERNEL.. | ---...         |
|       |          |               |          |                |
|       |   KJOB   |               |   KJOB   |                |
|       +----------+               +----------+                |
+--------------------------------------------------------------+
```

Figure 1: **Layers of Charlotte**

2.1. Inter-process communication

2.1.1. Links

The central idea of Charlotte inter-process communication is the concept of a *link*. A *link* is a logical, full-duplex connection between two processes, each of which has a capability to access one end of the link. To own a link is to own the capability of communicating across the link, to give it away or to destroy it. A process never refers to other processes directly. Instead, it presents a *link*

[2] Unix is a trademark of Bell Labs.

identifier, which is a small integer local to the process. The kernel uses this identifier to index an entry in the per-process capability table, which contains an index into an entry in the global link table. If the request is to send a message to the process at the other end of the link, the kernels of the two ends of the link will cooperate to pass the message. If both processes are on the same node, only one kernel is involved. A process may send a message, receive a message, cancel a send or receive request, destroy a link, or enclose a link in a message. We will examine these operations shortly.

Each link end is owned by only one process. The kernel stores information about each link in a *link descriptor*, which contains:

(a) static information about the link: Type, state, local-end address (Source) and remote-end address (Destination). An address is a triple of node identifier, process identifier, and system link identifier.

(b) Information concerning a current Send operation: send state (S_State), send buffer.

(c) Information concerning a current Receive operation, like (b).

A process is born with one umbilical link to its parent. Other links acquired by a process are enclosed in messages received by that process. By convention, a process begins its life in a **linkup** phase in which its parent grants it a number of links. Links are created by the **MakeLink** system call, which constructs a new link with the caller holding both ends. A process can introduce two colleagues to each other by forming a new link and giving each colleague one end. For the sake of simplicity, the rest of the paper discusses only normal links; we will ignore special links provided to allow KernJob processes to talk with each other (half links) and to allow Charlotte processes to communicate with entities outside the Charlotte world (primitive links).

The link abstraction is supported by the kernel, which itself depends on lower-level software and hardware. Figure 2 demonstrates this dependency relationship.

2.1.2. Communication primitives

Process-level communication is

- *Non-Blocking*: A process can generally continue executing while the kernel is transmitting a message on its behalf,
- *UnBuffered*: a message is not transmitted until the receiver has provided a place to put it, and
- *Synchronous*: processes are not interrupted by the arrival of messages or the completion of sent messages.

The unit of communication is a *message*. A message is a package of information of any length. The kernel ignores any internal structure in messages. A *buffer* is an area in the addressing space of a process that contains or is expected to receive a message.

Send(transmission link, buffer, buffer size, enclosed link)
: initiates a transfer of data from the indicated buffer along the indicated link. This operation remains in progress until its completion is reported by the **Wait** system call. It is possible to enclose an end of a link in the message, in which case that end disappears from the sender's grasp. If **Send** should fail, the enclosed link, if any, is restored to the grasp of the sender.

Receive(transmission link, buffer, buffer size)
: allows a message to be received on the indicated link and placed in the buffer. The link identifier *AllLinks* permits the acceptance of a message on any link held by that process. **Receive** on *AllLinks* may not coexist with **Receive** on a specific link, because it may cause inconsistency (from user point of view) in choosing the buffer to place arriving messages. If the buffer is not large enough to hold the entire message, as much as fits is placed in the buffer, the tail is lost, and the completion event (discussed shortly) notifies both the sender and receiver about this lose of transmission.

```
+------------+                                  +------------+
|            |                                  |            |
| User Proc  | ----------LINK------------       | User Proc  |
|            |    a logical connection          |            |
+------------+    supported by kernel           +------------+
     | |                                             | |
     | |                                             | | system call
     | |                                             | |
+------------+                                  +------------+
|            |                                  |            |
|   Kernel   | ---------Reliable---------       |   Kernel   |
|            |    transmission service          |            |
+------------+     supported by nugget          +------------+
     | |                                             | |
     | |                                             | |  procedure
     | |                                             | |  interrupt
+------------+                                  +------------+
|            |                                  |            |
|   Nugget   | -------- datagram---------       |   Nugget   |
|            |     service of Pronet            |            |
+------------+           network                +------------+
     | |                                             | |
     | |                                             | |  interrupt
     | |                                             | |
+-------------------------------------------------------------+
|                                                             |
|               Pronet: a token ring network                  |
|                                                             |
+-------------------------------------------------------------+
```

Figure 2: **Comunication layers**

Wait(transmission link, direction, event)
queries the result of a previous communication request: incoming (**Receive**), outgoing (**Send**), or both; on a specific link or any link. An event descriptor is returned by the kernel. It contains the matched link, direction, result code, and number of bytes transmitted. In case of **Receive**, it tells whether a new link was acquired, and if so, its user-relative number. When **Wait** matches an operation that is still in progress, the user is blocked until that operation completes.

Cancel(transmission link, direction)
requests cancellation of a **Send** or a **Receive** operation on the specified link. **Cancel** returns an error indication when the operation does not exist, and a failure indication when the operation has progressed beyond the point where the kernel can stop it. This call blocks the process unless the kernel is able to report success or failure immediately.

Destroy(link)
Requests that the given link be closed. **Destroy** always succeeds unless the link does not exist or is being transferred. This call will abort all outstanding **Send** or **Receive** requests on that link. It blocks the user until any necessary cooperation by the other kernel has completed. The process at the remote end will get a failure code ''link destroyed'' from any call related to that link. That end is reclaimed once the remote process invokes **Destroy** on its end. A link being transmitted as an enclosure does not belong to the current process. Request to destroy a link being transferred is an error, since the link no longer belongs to the user.

2.1.3. Discussion

Charlotte takes stands on several controversial issues. We would like to explain these issues and why we prefer the Charlotte position.

Simplex and duplex links

Arachne, the ancestor of Charlotte, uses simplex links. No information at all is stored in the kernel serving the owner of the link (the process that can receive along that link). The effect is that holders (those that can send) cannot be found easily. Because of this asymmetry, a link cannot be revoked, owner failure cannot always be reported properly to the holder, and if a process is moved to a new node, it is awkward to inform all potential senders of new address. A server has no control over the number of clients to which it may be connected, nor does it have easy way to distinguish them. Demos/MP employs similar simplex links. During process migration, it leaves a forwarding address at the old location. With duplex links, holders of remote ends will receive address updates, so message forwarding is not necessary. We believe that this design leads to better efficiency in migrating processes. Another shortcoming of simplex links is the cost of creating a fresh once-only reply link to be enclosed in every request to a server. On the other hand, duplex links take extra time to move, consume more table space in the kernel, and require a more complex inter-kernel protocol, as we shall see. These costs are outweighed, in our opinion, by the improvements in ease of use from the point of view of processes.

Naming

The advantages of directing messages to *process names* is the simplicity of usage, the minimum kernel support, and the possibility of compile-time consistency checking. The disadvantages are its difficulty supporting a replicated process, awkwardness in modifying cooperating process without recompilation, the difficulties of managing process name space, and the lack of protection.

Languages like Ada [8], Distributed Process [9], and Communication Port [10] support symbolic names as message destinations. One difference of a language-based IPC (inter-process communication) mechanism and an operating system based IPC mechanism is the early binding of symbolic names and communication format.

Global names provide a different mechanism. Processes talk to each other through a commonly known name. For example, a fileserver in V kernel [11] may identify itself by a service call as a well-known fileserver. A stronger type of global naming is to use a "post office": Processes may send messages to mailboxes and receive messages from mailboxes. The post office may or may not require authentication for these transactions. The problem of global naming in general is its poor protection from unwanted messages. It also requires a centralized message server.

Capability-based names, such as Charlotte links, have the advantage of both protection and flexibility. Since links are bound dynamically to processes, it is easy to change communication patterns in order to reconfigure the system or to reallocate resources. The disadvantages of using this kind of naming are the lengthy initial setup episode and the cost of validating and associating the capability with the destination process in every communication.

Non-blocking and blocking communication

Interaction between a program and any service of I/O operation generally consists of two parts: initialization and completion notification. These can be combined, blocking the user until completion. On the other hand, they can be separated, allowing the user to continue execution until the operation completes. Notification can be asynchronous, by interrupt, or synchronous, by a blocking call or polling. Charlotte chooses non-blocking initialization and synchronous, blocking notification. (A synchronous, non-blocking notification has also been implemented, and we are currently considering asynchronous notification.)

We prefer non-blocking initialization for **Send** (or **Receive**) because it can be used to implement blocking **Send** (or **Receive**), but permits a higher degree of concurrency. This increased concurrency does have a price: It requires the programmer to use a style that may be unfamiliar. It also might require more kernel calls to achieve communication.

Buffering

Between initialization and completion, messages must be stored somewhere. Kernel buffers have the advantage of freeing buffers in user space, which can be used immediately for new messages. There is no need to lock users in main store during transfer, either. A disadvantage of kernel buffers is the allocation problem when kernel buffers are depleted.

Buffer-allocation policies must limit unrestrained production of messages. One policy is to limit the number of buffers a single process may use at one time. A per-link limit is also possible. Arachne preferentially provides buffers for replies over requests. Charlotte version 1 allows at most one message per link to be buffered; subsequent sends block the sender until the previous message has been received. This policy leads to poor utilization of buffers. Charlotte version 2 avoids the problem by providing no buffering at all for messages. It is up to each process to allocate enough space for outstanding **Send** and **Receive** requests and to avoid manipulating data that is in transit.

2.2. Utility processes

We present here only a brief discussion of utility processes, their functions and roles in Charlotte. A more complete description can be found elsewhere [1].

KernJob

The *KernJob* is logically part of the kernel. It supplements the process-kernel interface through calls made by ordinary messages. These messages may come from processes on any node, whereas kernel calls are directed to the kernel on the same node as the process. For example, the Starter, which may reside on a different node from the processes it controls, uses the KernJob as an intermediary for manipulating those processes. The links over which processes submit requests to the KernJob are called *control links*. They are the same as ordinary links; their special functions are only a convention established by the kernjob.

Starter

The Starter squad manages the creation of new child processes. Each Starter is responsible for a set of nodes. A parent process that wishes to create a child sends a message to a Starter naming a file containing the executable code for that child.

SwitchBoard

The SwitchBoard is a name server. Any server process can register itself with a SwitchBoard with a set of patterns, and any customer process can ask a SwitchBoard to locate a server described by a pattern. The SwitchBoard responds with a link to that server.

FileServer

The FileServer squad has two implementations, one converting file access requests to calls to a Unix process residing on the host, and the other converting requests to calls to the WiSS fileserver computer in the Crystal network [12]. Open files are represented by links connecting the client and the FileServer. If the file is opened for read, the FileServer immediately starts reading from the file and sending data across the link. If the file is opened for write, the FileServer starts accepting messages across the link and transferring them to the physical device. In each case, control information may flow in the other direction on the link, like logical seek commands.

Connector

The Connector is a tool to establish initial linkage within a group of processes. A parent process that wishes to institute such a group sends the file name of a *connection description file* to a Connector. This file contains the names of the object files to execute and their inter-relationships. The Connector asks the Starter to load these files. Each process should start with a call to the library routine *Linkup*, which communicates with the Connector to receive the initial link set.

2.3. Example

The following is a scenario of opening a file. We will use it to demonstrate the use of links, as well as to show the interaction between user process and utilities. Suppose initially client C only has a link to a SwitchBoard. It asks a SwitchBoard to locate a fileserver link. The SwitchBoard first searches its own tables. It returns the desired link to C if it can find it. Otherwise, it relays C's request to some other SwitchBoard. C accepts the link oblivious to the cooperation between SwitchBoards. Likewise, C's request to the FileServer to open a file may be relayed to another FileServer, invisibly to C. The diagram below ignores these intra-squad communications.

```
+------------------+  +---------------+  +------------------+
| user process C   |  | SwitchBoard   |  |    FileServer    |
+------------------+  +---------------+  +------------------+
      |__________________|                  |__________________|
         link A                                link B
       --(C1)----A-->
                         (S1)
                         |_______|
                          link C
                            -(S2+link C)-B-->
                               |__________________|
                                   link C
       <--A-(S3+link C)--
       |_________________________________________|
                       link C
       ---(C2)---------------------------------C-->
                                              (F1)
                                              |_______|
                                              link D
      <--C--------------------------(F2+ link D)---------
      |__________________________________________|
                     link D
```

C1 (C to SwitchBoard) Do you have a FileServer?

S1 (SwitchBoard) Finds a registered FileServer, makes a new link.

S2 (SwitchBoard to FileServer) Take this link to a new client.

S3 (SwitchBoard to C) Take this link to the FileServer.

C2 (C to FileServer) Open file ''Foo'' for reading.

F1 (FileServer) Creates a link to represent that file.

F2 (FileServer to C) Take this link to your new file.

As this example shows, links play a dynamic role in representing resources. A link to a server can be viewed as a capability to get a certain service.

3. Kernel design

3.1. Overview

The kernel, which resides on each node, must be efficient, concise, and ease to implemented. As we have seen, most Charlotte services are provided not by the kernel but by squads of utility processes that cooperate with each other to fullfill requests. The kernel needs to provide only two abstractions: *links* and *processes*. The process abstraction is kept quite simple; any control over processes is relegated to the KernJob, not the kernel.

In contrast to version 1, Charlotte version 2 tries to be both modular and expandible. The following decisions derive from these goals:

- Most of the kernel is written in Modula. It provides subscript-range checking, walkback upon failure, modular structure, and inter-module protection. Only about 600 lines of assembler and C code are needed for device interrupts, trap handling, context switching and other hardware-dependent activities.
- Modula's concurrency facilities are used for internal tasks. (Modula processes are called *tasks* here to avoid confusion with processes running on top of Charlotte.) For example, device interrupts are handled by waking up an appropriate task. The clarity gained by this decision is partially offset by the cost of task switching.
- We use finite-state automata to implement the kernel-to-kernel protocol. This approach allows a structural implementation, logical breakdown of complex situations, and relatively easy expansion of the protocol.
- Queues and monitors are used to synchronize tasks. Shared data are protected by Modula interface modules.

3.2. Task structure

The kernel is composed of four categories of tasks: the Envelope, the Finite-State Automaton, *n* device handlers, two nugget handlers, and two occasional servers. Small control packets called *work requests* (or simply *requests*) are enqueued to the finite state automaton and the nugget handlers.

Envelope
: This task encapsulates all user processes, which appear as subroutines to this task. The envelope calls a user, which returns when it submits a service call or when interrupted by the clock. Service calls are either performed directly or translated into work requests and enqueued for the finite-state automaton. Clock interrupts signal the end of a quantum; round-robin scheduling selects the next process to run. Long-term scheduling is the province of the Starter, using the KernJob to suspend and resume processes.

Finite-state automaton
: This task accepts work requests from its input queue (implemented using hardware queue instructions for efficiency). Requests are directed to one of four modules, each handling a different aspect of communication: sending, receiving, moving/destroying links, and communicating over special links (which we will not discuss further). The use of a queue serializes all requests and avoids race conditions. The internal structure of the automaton modules will be discussed in detail in Section 4.

Device handlers
: Charlotte aims to convert the raw hardware interface into a convenient uniform interface for user programs. Our devices include the nugget (a software communication device provided on all Crystal nodes), the clock, console, and optionally a disk. Each device is associated with a task awakened by the associated interrupt. Our current device handlers are rudimentary,

providing only low-level service for the console. The eventual goal is to provide a link-like interface for all devices accessible to users.

Nugget handlers

Nugget handler tasks invoke the **Send** and **Receive** functions of the nugget. There is a separate send handler for each node and one receive handler. (Revised nugget designs will allow us to simplify this situation considerably.) The Nugget interrupts upon completion, awakening the appropriate handler. Send handler accepts work orders from its queue. The messages are generally not interpreted by the handler; they are a mix of process-generated and automaton-generated traffic. (A few violations of this principle have been made for the sake of efficiency.)

Occasional servers

Two tasks awaken periodically to perform housekeeping chores. The *HeartBeat* task checks how long this node has not heard from each other node. If the silent interval is long (currently two seconds), the HeartBeat enqueues a work order to send that node a packet. If the nugget fails to send this (or any other) packet, we know that node has failed. (The sending nugget handler reports the failure to the finite-state automaton task, which will destroy all links to that node.) The *Statistics* task gathers performance information concerning CPU and network usage. This information is given to the KernJob, which presents it to the Starter as a guide for static and dynamic load balancing.

All kernel tasks are created at node initialization time and never terminate. We have found that the queue architecture leads to no deadlocks between tasks. Debugging has been relatively easy. (A few programmer-months built most of the function described in this paper.)

4. Protocol Evolution and Implementation

We are now ready for a careful examination of the communication protocol and its implementation. We shall concentrate on communication primitives and messages, mentioning other requests only briefly when appropriate. A more elaborate description of those can be found elsewhere [1].

We will focus on the finite-automaton task. It invokes (via procedure calls) specialized handlers for **Receive**, **Send**, **Destroy**, and **Move**. It also deals with **Wait**, **Terminate** and closing all connections to another Charlotte node. **Cancel** is treated by both the receive and send handlers.

In what follows, we shall view these handlers as specialized finite-state machines for Receive, Send, Destroy, and Move (or the R, S, D, and M machines, respectively). Finite-state machines are commonly used to formally specify communication protocols. Our method follows other work [13] in which ''context information'' is traded off for states, and work [14, 15] in which a state-machine model is mixed with a programming-language model, by using elaborated actions and a set of variables. We differ from these models in the complexity of situations we have to cope with and the interference between the state machines.

Roughly speaking, there are two classes of requests to these machines: *user to kernel* and *kernel to kernel*. (Users include utility processes as well as standard user processes.) The first class is invoked by service calls such as **Send**. The second class is usually the result of a user's request, transferred to the kernel at the other end of the link (possibly the same kernel).

We will first describe the four machines (S, R, D, and M) in a simplified fashion, isolated from each other's interference. We start with the S and R machines and then show how **Wait** interferes with them. We then show a simple D machine. Next, we add the complexities introduced by **Cancel** and the impact of binding **Send** and **Receive** requests across machines, to show complications that arise in the S and R machines' logic. Moving a link is discussed next; we present alternatives, show some possible (still simple) scenarios, and describe our simple M machine. The rest of this section will show more and more complex scenarios of communication and a stepwise evolution of the protocol to handle such possibilities.

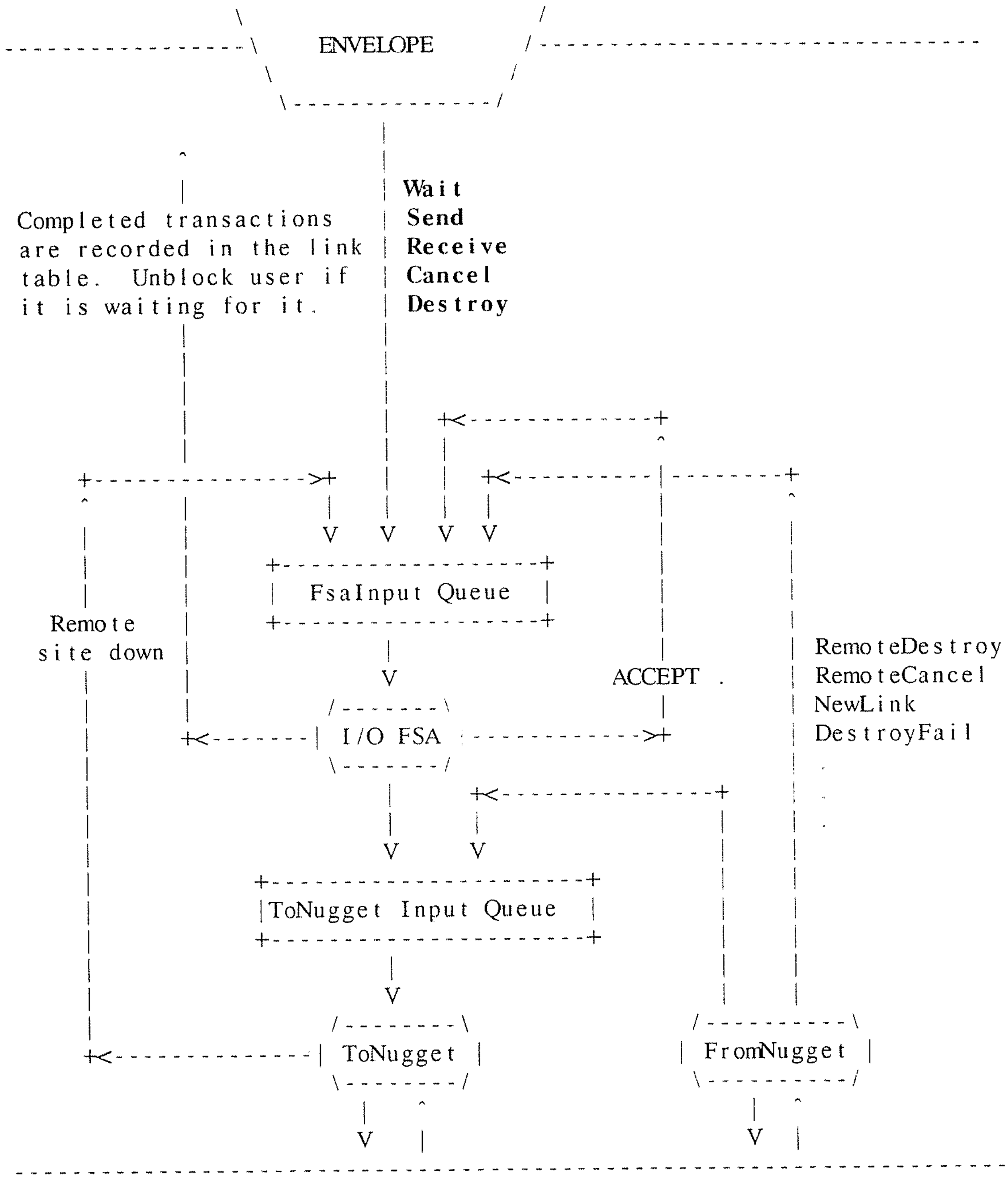

Figure 4: **Internal kernel structure**

Our goal in this order of presentation (which differs from the order in which we designed and tested the protocol) is to see why simple situations become complicated under our flexible communication rules. We wish to convince the reader that our solution is correct. It is also efficient in the sense that we perform simple cases in the simplest way and complex (or relatively rare) cases as simply as we can provided we don't spend enormous space for FSA tables.

As mentioned earlier, we shall present only the protocol for normal links. We end this chapter with discussion of alternatives and enhancements to our protocol. Some of these changes are currently under discussion; others have been rejected.

4.1. The Four Basic Machines

4.1.1. Send

As described earlier in Section 2.1, the **Send** request specifies a specific link and a buffer of data (unstructured to the kernel). **Send** is a non-blocking operation.[3]

A simple Send-FSA table is as follows:

S_State		Request	
		U_Send	K_Accepted
1	S_Idle	A:2	X
2	S_SendOut	B:2	C:3
3	S_Done	A:2	X

Table 4.1: **Simple Send-FSA table**

Actions:

A Out(cl,K_Send).

B KeepRequest(cl,NextSend); BlockUser(cu).

C StoreResult(cl,S,SendOk).

X impossible case (invoke fatal-error handler)

All our FSA tables are arranged in two dimensions. The first is the send, receive or link state; the second is the request type. Each cell specifies the current action and the next state. Each request refers to a link, denoted *cl* (CurrentLink) above. For now, we assume that the link is not concurrently being destroyed or moved. Information about the end of a link, such as its S_State, R_State, and L_State, is known to all FSA machines in the kernel of that end of the link. The owner of cl is denoted *cu* (CurrentUser).

The send state of a link can be S_Idle (nothing has been sent, or the last **Send** has completed and reported by **Wait**), S_Done (completed but not yet **Wait**ed) or S_SendOut (request sent to the receiver, but not yet acknowledged by its kernel).

This FSA can handle two inputs, U_Send (data only, without an enclosed link) and K_Accepted (acknowledges that the receiver has accepted the data). Our first example of interference between seemingly unrelated machines is that S_State will be changed from S_Done to S_Idle by the **Wait** mechanism, which is otherwise outside this FSA. The above protocol assumes that the user's buffer size can fit one packet, which is limited to 2KB by the hardware.

Actions taken by the S machine are described through primitive operations such as Out and StoreResult, which can be considered high-level machine instructions:

Out queues the request on a queue for the ToNugget task or FSA task (if the destination is the same node).

KeepRequest stores a pointer to a **Send** or **Receive** request, to be issued when the current in-progress request on the same link is completed.

StoreResult stores a completion or error code as an S_Result or R_Result in the link descriptor. A later **Wait** will find this result if it matches, return it to the user, clear the field in the link descriptor, and set S_State to S_Idle. If the user was already in a matching **Wait**, the completion information is passed immediately. If a new **Send** has already been submitted, then a new U_Send is issued, and the user is

[3] We allow a new **Send** on a link as soon as a previous one completes. When a user requests a **Send** before the previous one completes, we queue the new request and block the user. We rejected considering the new **Send** an error; we have deferred a design that allows multiple transactions per link.

unblocked. The completion code will indicate if the **Receive** buffer was too short.

4.1.2. Receive

Receive is similar to **Send**; it is non-blocking, synchronous and non-buffered. Messages arriving from other nodes are temporarily stored in a kernel buffer, which is copied to the receiver's buffer, if ready, and freed immediately. We present first the protocol and implementation for a simple **Receive**.

Two important decisions were made.

(1) **Receive** and **Send** are similar in the sense that a request could be delivered to the other side when the receiver is ready to receive. It turned out that our active Send with passive Receive protocol is more efficient.

(2) We decided to send data together with the K_Send header, instead of sending the header first and waiting for permission to send the data when the receiver is ready, because if both ends of a link are on the same node, a permission message is superfluous, since it is easy to see if a matching **Receive** has been submitted. We would like to have same protocol for inter- and intra-node communication. Furthermore, the time penalty for sending even full-size packets instead of short handshaking packets is relatively small in our environment.

We will temporarily ignore the problem of losing data if no **Receive** matches when K_Send arrives.

The simple Receive-FSA table is as follows.

R_State		**Request**	
		U_Receive	**K_Send**
1	R_Idle	A:3	D:2
2	R_SendIn	B:4	X
3	R_ReceivePend	C:3	E:4
4	R_Done	A:3	D:2

Table 4.2: **Simple Receive-FSA table**

Actions:

A **skip**.

B AcceptData(cl); StoreResult(cl,R,ReceiveOk); Out(cl,K_Accepted).

C KeepRequest(cl, NextReceive); BlockUser(cu).

D for now, same as A.

E for now, same as B.

The Receive-FSA table is quite similar to the Send-FSA table, except that we handle the case where the K_Send message arrives before a **Receive** request. Currently, it does not matter which comes first; actions B and E are equivalent. Soon we will show that the order matters.

The operations Out, KeepRequest and StoreResult are as described earlier.

AcceptData copies data from the incoming message (or the sender's buffer, if on the same node) into the receiver's buffer. If the receiver's buffer is too small, a warning notification is returned to both the sender and the receiver as the completion result.

R_Idle is the initial state; the **Wait** call resets R_Done to this state. The R_SendIn state follows either R_Idle or R_Done when K_Send is handled. If a **Wait** request comes when the link is in R_SendIn state, **Wait** needs to know whether this state came after R_Idle or R_Done. The distinction is that in the latter case, the R_Result stored in the link descriptor is valid.

The following two examples will demonstrate the protocol for a simple **Send**-**Receive** pair. Assume users **A** and **B** are communicating via link **L** (which they call La and Lb, respectively). Their kernels are **K(A)** and **K(B)**, respectively (possibly identical). Both ends of the link are initially in states R_Idle and S_Idle.

Example 4.1

(1) **A** calls **Send**(La), which causes a U_Send to be enqueued for S_machine(La).

(2) S_machine(La) outputs K_Send and sets S_State(La) to S_SendOut.

(3) R_machine(Lb) gets K_Send, stores the pointer to the buffer, and sets R_State(Lb) to R_SendIn.

(4) Some time later, **B** calls **Receive**, which enqueues a U_Receive request on R_machine(Lb).

(5) R_machine(Lb) takes action B to process this request, calling AcceptData, StoreResult (returning a special indication if the **Receive** buffer was too short), outputting K_Accepted (indicating how many bytes were accepted), and setting R_State(Lb) to R_Done.

(6) S_machine(La) gets K_Accepted, calls StoreResult, and sets S_State(La) to S_Done.

(7) At any point, **A** and **B** may submit **Wait** requests that match the event that has happened. If U_Wait is handled by the FSA before the result is stored, the caller **A** or **B** is blocked, to be awakened later by StoreResult. Otherwise, the FSA will find the result stored in the La (Lb) descriptor, and return immediately to the user.

Example 4.2

Assume events happen as in the previous example, but **A** invokes **Send** again before **B** calls **Wait**. In this case, K_Send is processed by R_machine(Lb) before U_Wait gets to FSA(Lb). R_State(Lb) is R_Done, so action D (for now, the same as A) is performed, and R_State(Lb) becomes R_SendIn. An eventual **Wait** by **B** will find the R_Result valid, report it to **B** without blocking, invalidate it, and *not* modify R_State(Lb).

Other event orders are possible. For example, R_machine(Lb) could encounter U_Receive before a K_Send arrives. In any case, the memoryless principle is maintained: We record in states only the current situation for each link and the result of at most the last **Receive** and/or **Send** request.

4.1.3. Wait

We mentioned in Section 2.1 that the **Wait** call specifies a link (either specifically or *AllLinks*) and the direction of communication (S, R, or All). These two parameters are orthogonal to each other. A matched **Wait** is one whose operation has completed or is still in progress.

Three cases occur:

(1) No match can be found. An error (''nothing to wait for'') is returned to the user.

(2) A match is found, but the operation is in progress. The user is blocked. For **Wait**(*AllLinks*), we wait for any action to finish. When the operation completes, the user will be awakened when StoreResult is called.

(3) A completed operation is matched. The result is returned without blocking the user.

We saw earlier that the S and R states are sufficient to distinguish these cases. In state S_Done (R_Done), case (3) holds, and state is set to S_Idle (R_Idle). In state R_SendIn, we have to check whether a valid R_Result exists; we do not modify R_State. If the user issues a new **Send** (**Receive**) before using **Wait** to insure the previous one is finished, the previous completion event is forgotten, and **Wait** will block on the new in-progress operation.

Wait is implemented as part of the FSA task. No race conditions can arise between it and the S or R machines, since these are inactive when the FSA is active. The following algorithms describe the logic to handle **Wait**:

Algorithm 4.1: Wait(AllLinks,D) where D = R, S, or All

```
var
        Match : Boolean;               (* true in case (3) above *)
        PossibleMatch : Boolean;       (* true in cases (2) or (3) *)
        PossibleMatch_L : Boolean;     (* temporary, for each link *)
begin
        PossibleMatch := false;
        forall L in user's links do
                (Match, PossibleMatch_L) := FindMatch (L, D)
                when Match exit;
                PossibleMatch := PossibleMatch or PossibleMatch_L;
        end;
        if not Match then
                if not PossibleMatch then (* case (1) *)
                        ReturnToUser (NothingToWaitFor)
                else (* case (2) *)
                        BlockUser;
                fi;
        else (* case (3) *)
                skip
        fi;
end
```

FindMatch sets PossibleMatch true only if the given link has an outstanding, in-progress operation in the specified direction. It sets Match true if the operation has finished. In that case, it also returns the link number, direction, and result to the user and invalidates the appropriate R_Result or S_Result in the link descriptor.

Before Algorithm 4.1 is called, we verify that the user has at least one link, or ReturnToUser(NoAnyLinks) is called immediately. If the user is blocked, the first operation matching the specified direction that completes will return the result via StoreResult.

Algorithm 4.2 : Wait(L,D).

```
var
        Match, PossibleMatch : Boolean;    (* as above *)
begin
        (Match, PossibleMatch) := FindMatch (L, D)
        if not Match then
                if not PossibleMatch then
                        ReturnToUser (NothingToWaitFor)
                else
                        BlockUser;
                fi;
        fi;
end;
```

Algorithm 4.2 is a special case of algorithm 4.1.

4.1.4. Destroy Link

Destroy is guaranteed to complete successfully, although it may be deferred for some time. We will clarify this statement later. When user **A** requests **Destroy**(La), the D machine will pass this request to D_machine(Lb), which will respond with K_DestroyOK. User **A** is blocked during this time.

When can the destroyed link be freed (operation DisposeLink)? Even though the link cannot be used again, information might still arrive from the other end of the link until both sides have agreed that it is destroyed. For example, assume **A** requests **Destroy**, which is granted by D_machine(Lb). La can be freed when the K_DestroyOK arrives, since **A** is precluded from any further operation on this link. As to Lb, there are few policies the D_machine might follow:

(1) Free Lb immediately when destroy is granted.

(2) Free after **B** is notified that link was destroyed at **A**'s request.

(3) Free only if there are no valid results stored.

(4) Free only when **B** explicitly requests **Destroy**.

Policy (1) doesn't allow **B** to examine the results of previous **Send** or **Receive** operations. If both a **Send** and a **Receive** have completed, then policy (2) will discard the results of one of them. The first three policies share another defect: The error that **B** gets for trying to **Send** or **Receive** again on Lb depends on whether or not the link has been destroyed, which should therefore be under **B**'s control. We have therefore chosen policy (4) to be consistent with our timing principle: Errors should depend on the user's behavior, not on circumstances beyond the user's control. Between its destruction by D_machine and **B**'s **Destroy** request, Lb will be in a L_Dead state, which allows operation **Wait**, but *not* **Send**, **Receive**, or **Move**.

When a user terminates, all its links are destroyed as if individual U_Destroy requests were made on each.

The Destroy-FSA table is:

	Link state	Request		
		U Destroy	K Destroy	K DestroyOK
1	L_OK	A:2	C:3	X
2	L_LocalDestroyed	X	D:4	D:4
3	L_Dead	B:4	X	X
4	L_Free	X	X	X

Table 4.3: **Simple Destroy-FSA table**

Actions:

A BlockUser(cu); Out(cl,K_Destroy)

B DisposeLink(cl)

C Out(cl,K_DestroyOK); ClearDeadLink(cl)

D ClearDeadLink(cl); DisposeLink(cl); UnblockUser(cu)

We have incorporated our decision to use method (4) above in the L_Dead state, in which the link has not yet been destroyed. Two optimizations were also incorporated:

(1) If both sides simultaneously send each other a K_Destroy request, neither needs to respond with K_DestroyOK. Instead, receiving a K_Destroy is equivalent to receiving K_DestroyOK; the link is closed and freed.[4]

(2) The user is not blocked unnecessarily, for example, when the link is already in state L_Dead.

---

[4] A difference exists only when the ends of the link reside in different node machines: receiving K_DestroyOK implies that the D_machine of the other end has seen our K_Destroy request; however, receiving K_Destroy doesn't; our request may be still in the queue of outgoing messages. Link disposal is therefore deferred until every message with respect to it is delivered.

The new operations introduced here:

DisposeLink clears link pointers and returns resources, such as buffers or descriptors for requests in progress.

ClearDeadLink terminates any in-progress **Send** or **Receive**. Since the K_Destroy message must be the last one in each direction on this link, **Send** or **Receive** will never complete. ClearDeadLink therefore sets the S_State and/or R_State to S_Done/R_Done and calls StoreResult to report S_Broken/R_Broken. Any pending NextSend or NextReceive is also removed.

These operations are examples of interference between FSA machines. We implement such interference by letting operations called by one machine produce side-effects in the states of other machines. Instead, we could let one machine enqueue internal messages to the other. In our example, the D_Machine could send a message to the S_Machine to announce that the link is broken. Although this solution is clearer, it is very inefficient. It can also lead to unreasonable growth of the FSA tables, since there are many more cases to consider.

There is an asymmetry between the Local-end and Remote-end request to destroy a link, since the former changes the link state until the other end approves the destruction, while the latter is accomplished immediately, and the link becomes L_Dead. For now, this solution is adequate. However, with **Move** algorithms, we will see cases when a K_Destroy request from the other end must be deferred.

Example 4.3

(1) User **A** calls **LinkDestroy**(La), and a U_Destroy request is enqueued.

(2) D_machine(La) blocks **A** and sends K_Destroy.

(3) D_machine(Lb) gets the K_Destroy. Assume Lb's state is L_OK, hence action C is taken: Operation ClearDeadLink(Lb) is performed, breaking each outstanding **Send** or/and **Receive**. Lb's state becomes L_Dead, and K_DestroyOK is sent.

(4) D_machine(La) receives K_DestroyOK, frees La and unblocks **A**.

(5) Further attempts by **A** to use La will be considered errors. **B** may request **Wait** on Lb (assuming some activity had not yet been **Wait**ed for). **Wait** will report ``broken'' and set R_State and S_State to R_Idle / S_Idle.

(6) If **B** attempts further actions (**Send**, **Receive**, **Wait**, or **Cancel**) on Lb, they will fail immediately and return ``Link is dead''.

(7) When **B** calls **Destroy**(Lb), D_machine gets U_Destroy request and will call DisposeLink by action B.

It is an easy exercise to check that the protocol works when both users call **Destroy** at the same time and when a single process holds both ends of a link.

4.1.5. Receive, Send and Buffer Management

Before introducing the **Cancel** mechanism, let us correct the Send/Receive FSAs by removing any assumptions concerning buffers. This correction will clarify the description of **Cancel** in the next section. We will continue to assume that messages are of a limited length and are transfered in one packet.

Intra-node communication employs only user buffers. The following discussion therefore deals only with inter-machine communication.

We allocate a small number of kernel buffers to incoming messages. When a data message arrives, it is handled by the R machine, which subsequently releases the buffer. If the R machine finds a matching **Receive** call pending, action E (Table 4.2) is taken, and the buffer is freed and placed at the head of the free list. Otherwise, action D is taken. As we mentioned earlier, we don't necessarily keep buffers until the matching **Receive**. Instead, we mark the buffer as free but still containing useful information, and place it at the tail of the free list. When new buffers are needed, they are taken from the head of the free list. There is a great chance that a full buffer placed at the end of the free list will survive intact to the time that its matching **Receive** occurs, in which case we copy its contents and complete the transaction. If the buffer does not survive that long, the R machine requests the sender to send the data again (K_SendAgain). The S machine is modified to handle this request.

Once the receiver has told the sender to repeat the data, we don't need a final K_Accepted message. (However, we will see that **Cancel** interferes with this optimization.) The modified Send/Receive FSAs are:

R_State		Request	
		U Receive	**K Send**
1	R_Idle	A:3	D:2
2	R_SendIn	B:4(5)	X
3	R_ReceivePend	C:3	E:4
4	R_Done	A:3	D:2
5	R_ReceiveAgain	C:5	F:4

Table 4.4: **Receive-FSA table with buffer management**

Actions:

A **skip**.

B **if** BufferIsAvail(cl) **then** (* as previously *)

AcceptData(cl);
StoreResult(cl,R,ReceiveOk);
Out(cl, K_Accepted)

else

Out(cl, K_SendAgain);
R_State := R_ReceiveAgain

fi.

C KeepRequest(cl, NextReceive); BlockUser(cu).

D FreeBuffer(full).

E Like **then**-part of B.

F AcceptData(cl); StoreResult(cl,R,ReceiveOk); (* with no K_Accepted *)

S_State		Request		
		U Send	**K Accepted**	**K SendAgain**
1	S_Idle	A:2	X	X
2	S_SendOut	B:2	C:3	D:3
3	S_Done	A:2	X	X

Table 4.5: **Send-FSA table with buffer management**

Actions:

A-C As in Table 4.1.

D A; C.

The Send FSA is only slightly changed. In the Receive FSA, a new action F differs slightly from E; D and E are no longer equivalent to A and B respectively. K_Send is used to send the initial message as well as to respond to the K_SendAgain request; it is clear from context which case applies.

The Receive FSA is no longer "pure", since action B can select a different next state. This sort of side effect was both useful and clear [5] in other cases too.

Some new buffer operations have been added:

BufferIsAvail returns true if the sender is local or the desired buffer is still intact on the free list.

FreeBuffer is ignored if the sender is local. Otherwise, it places the buffer either at the head or tail of the free list, depending on whether its data have been read.

AcceptData As previously, but it also frees the empty buffer (if the sender is remote).

4.2. A Second Level of Complexity

We will now introduce **Cancel** and **Move** and revisit **Destroy**. These calls are tightly coupled with each other, with mutual dependencies and interferences.

4.2.1. Cancel

Cancel is needed to let the user avoid deadlocks and to break **Sends** with no matching **Receive**. To make **Send** and **Receive** symmetric, each may be canceled. The **Cancel** call takes two arguments: a specific link number and a direction, which is limited to R (Receive) or S (Send). Blocking the user when Cancel cannot complete immediately prevents the user from requesting conflicting operations (like moving or destroying the link) while cancellation is in progress. We guarantee that **Cancel** will not delay the user longer than the time needed to pass a few messages between kernels.

Cancel(S) may be rejected immediately if there is no outstanding **Send** on the link (with error result "nothing to cancel" or "too late to cancel"). Otherwise cancellation is requested of the kernel at the other end of the link. This request is handled by the R machine, which approves cancellation (K_CancelOK) as long as the **Send** has not yet been matched by a **Receive**. If it is too late, the R machine denies cancellation (K_CancelFail).

We can improve this protocol by noticing that K_CancelFail means that a previous K_Accepted, sent earlier by the R machine, was not received by the sender's S machine by the time the **Cancel** was forwarded.[6] We therefore eliminate K_CancelFail: instead of sending it, the R machine ignores the **Cancel** request; the S machine will interpret K_Accepted to imply K_CancelFail. Another opportunity for improvement occurs in the case that the R machine requests K_SendAgain before it gets a K_Cancel request. We can let the R machine approve cancellation without sending K_CancelOK, since the S machine can interpret K_SendAgain to imply K_CancelOK.

Cancel(R) is easier to implement, since it requires only local actions. If there is no outstanding **Receive** on the link, cancellation is rejected immediately, as it was for **Cancel**(S); otherwise, the link can be in R_ReceivePend state, in which **Cancel** completes successfully, or in R_ReceiveAgain state, in which the user is blocked until the response from the Sender arrives. The response may be the requested data (in which case **Cancel**(R) fails), or a K_Cancel (Send) request (both **Cancels** succeed).

We chose to incorporate **Cancel** in the Send/Receive FSAs, not as a separate FSA, because it interacts so heavily with these two.

---

[5] To avoid letting an action determine the next state, we could introduce a state like R_BufferAvail, which would be checked and modified from other modules whenever a buffer is used for inter-machine communication. This solution introduces even more complicated side effects and inter-module messages.

[6] Otherwise, S_State would have been S_Done and the **Cancel** would have been rejected immediately.

The augmented Receive FSA and Send FSA are:

R_State	User Request		Kernel to Kernel	
	U_Receive	U_Cancel_R	K_Send	K_Cancel_Send
1.R_Idle	A:3	x1:1	D:2	N:1
2.R_SendIn	B:4(5)	G:2	X	J:1(4)
3.R_ReceivePend	C:3	H:1	E:4	N:3
4.R_Done	A:3	I:4	D:2	N:4
5.R_ReceiveAgain	C:5	L:6	F:4	N:3
6.R_Cancel_Recv	X	X	K:4	M:1

Table 4.6: **Receive-FSA table with Cancel**

Actions:

A-F As previously in Table 4.4.

G **if** R_Result is valid **then** I **else** x1 **fi**.

H ReturnToUser(CancelOK).

I ReturnToUser(TooLateToCancel).

J Out(cl, K_CancelOK);
if R_Result is valid **then** *R_State* := *R_Done* **fi**.

K F: I;* UnblockUser(cu).

L BlockUser(cu).

M H; UnblockUser(cu).

N **skip**.

x1 ReturnToUser(NothingToCancel).[8]

S_State	User Request		Kernel to Kernel		
	U_Send	U_Cancel_S	K_Accepted	K_SendAgain	K_CancelOK
1.S_Idle	A:2	x1:1	X	X	X
2.S_SendOut	B:2	E:4	C:3	D:3	X
3.S_Done	A:2	F:3	X	X	X
4.S_Cancel_Send	X	X	G:3	H:1	H:1

Table 4.7: **Send-FSA table with Cancel**

Actions:

A-D As previously in Table 4.5.

E Out(cl, K_Cancel_Send); BlockUser(cu).

F ReturnToUser(TooLateToCancel).

G C; F. UnblockUser(cu).

H ReturnToUser(Cancel OK); UnblockUser(cu).

[7] This state covers both R_ReceiveAgain and a pending U_Cancel_R.

[8] a non-fatal user error.

x1 ReturnToUser(NothingToCancel).

At first glance, The Destroy FSA needs no modification to incorporate **Cancel**, since **Cancel** and **Destroy** are mutually exclusive user requests (both are blocking). However, the kernel may issue one or both on behalf of the user. For instance, when a user terminates, its open links are closed by a simulated **Destroy** request on all links. We shall say more on this interaction after the Move machine is introduced.

4.2.2. Move link

A user is allowed to move its end of a link to any other user (including itself) to which it has a link. Link motion is performed by the **Send** call, whose last optional argument is the enclosed (moving) link. The link along which the transfer takes place will be called the transferring link. For the sake of example, we will call the active user **A**, with a link m (moving) to **B** and link t (transferring) to **C**. At the end of the transfer, **A** will only have link t to **C**, and **B** and **C** will be connected.

```
 ----------           ----------        ----------         ----------
|          |    m'   |          |      |          |       |          |
| User A   |<------->| User B   |      | User A   |       | User B   |
|__________|    m    |__________|      |__________|       |__________|
   /\                                      /\                  /\
    \  t                                    \  t               /  m'
     \                                       \                /
      \                                       \              /
       \                                       \            /
     t' \                                    t' \          /  n
        _\/_______                             _\/_____\/_
        |          |                           |          |
        | User C   |                           | User C   |
        |__________|                           |__________|

             (a)                                    (b)
```

Figure 5: **Link Moving - Before and After**

As with previous calls, the Envelope verifies that both links are owned by **A**. Two more requirements must be verified:

(1) The moving link should be acceptable (L_State = L_OK).

(2) The moving link should have no outstanding **Send** or **Receive**.

If one of the above fails, an appropriate error message is returned.

If an incoming **Send** (from **B**) is pending on the moving link (R_State = R_SendIn), it should be returned to **K(B)** (by kernel message K_SendLater). When the new destination (**C**) is introduced to **B**, **K(B)** can resend that message to **C**. However, this resend policy assumes that **Move** always completes successfully, which is incorrect.[9] If **Move** does not succeed, the moving link is re-installed, and the previous incoming **Send** (from **B**) should be re-activated by a K_NoUpdate message sent to **K(B)**. A similar situation occurs when an incoming **Send** (from **C**) arrives on a moving link.

We will ignore for the moment complicated scenarios like **B** or/and **C** moving m′ (or t′) concurrently, or trying to **Destroy** m′ or t′.

We divide the **Move** protocol into three stages:

---

[9] For example, the transferring link from **A** to **C** may be destroyed before the moving link was really accepted by **B**.

prologue: **A** introduces link m′ to **C**.

body: **K(C)** and **K(B)** install the link n-m′ between **C** and **B**.

epilogue: **K(A)** is notified that the **Move** succeeded and deletes m.

In what follows, we elaborate on these stages.

Prologue We considered three alternatives for the protocol:

(1) **K(A)** asks **K(B)** permission to move m′, then encloses m in a message to **K(C)**.

(2) **K(A)** notifies **K(B)** to freeze m′, then encloses m in a message to **K(C)**.

(3) **K(A)** encloses m (and the identity of m′) to **K(C)**.

Alternatives (1) and (2) would simplify conflicts such as an incoming **Send** on either link or a concurrent **Move** of both ends of a link. However, simplification is not always possible (for example, when both sides request permission or freeze concurrently). More messages might be required. We therefore chose alternative (3). This choice affects the body and epilogue as well. Hence the prologue is:

> **K(A)** sends K_Enclose request to **K(C)** across t, which like K_Send contains data, and additionally a description of both ends of the moving link (m and m′).

Body This part is performed by **K(C)** on receiving K_Enclose from **K(A)**, assuming **Receive** is already pending on link t′; otherwise, it is delayed until **C** calls **Receive**(t′,...).[10] Hence the body is:

> **K(C)** creates a new (tentative) link **n**, whose destination is m′, and sends K_Update to **K(B)** (including a description of n).

Epilogue **K(B)** grants the K_Update request (usually) by updating the new destination of m′ to n and sending K_Update_OK to **K(C)**. (These are transparent to **B**). In rare situations, complicating events may occur, such as when m′ is being destroyed by **B**. In that case, **K(B)** replies K_UpdateFail to **K(C)**.

We considered two alternatives for the protocol of the epilogue:

(1) **K(B)**'s response is routed back to **K(C)**, which accordingly installs n as a regular link, and responds K_Accepted to **K(A)**, or removes n and responds K_Rejected to **K(A)**.[11]

(2) **K(B)** responds directly to **K(A)**.

In both alternatives, **K(A)** finalizes by removing m, if the **Move** is accepted, or reinstalling m or retrying the **Move**, if it was rejected (provided t is not destroyed already).

Alternative (2) seems more attractive, since in a normal case only three messages are required for the entire (successful) **Move** protocol. Failure, such as when t or t′ is destroyed, allows rejection to be deduced from context, with no need for K_Rejected message. However, this alternative complicates concurrent moving and destroying the transferring/moving link(s), and it requires additional message types (such as forwarded-update for changed destination). Alternative (1) was selected, since it is simpler and had been studied and verified before we examined all implications of the second. Since the **Move** operation is relatively rare, and the complex situations much more rare, the slight inefficiency of alternative (1) is less important than simplicity. We may eventually switch to the other alternative. Hence the epilogue is:

---

[10] Links are accepted by users through ordinary **Receive** calls. It seemed less appropriate to have a special **ReceiveLink** call.

[11] A **Send** with enclosed link is atomic: Either both the data and the enclosed link are successfully transferred or neither is.

(i) **K(B)** sends K_Update_OK or K_UpdateFail to **K(C)**, and accordingly un/updates link m's destination; (ii) **K(C)** accordingly installs/deletes link n and replies K_Accepted or K_Rejected to **K(A)**; (iii) the latter accordingly deletes link m, or retries the **Move** from start, or returns a failure indication to **A**.

We introduce a new Move FSA to handle the update negotiations. The Send FSA and Receive FSA are augmented to handle the new messages described above, as well to recognize the fact that a link can be destroyed or moving. These FSAs are shown in Appendix A. Other modifications are discussed in the next sub-sections.

Example 4.4

(1) **A** calls **Send**(t,......,m); t and m are owned by it.

(2) **K(A)** verifies that m is L_OK with no outstanding **Receive** or **Send**.

(3) S_machine(**A**) sends K_Enclose across t, including a description of m and m'; S_State of t becomes S_EnclOut. L_State(m) becomes L_Moving.

(4) R_machine(**C**) gets K_Enclose. Assume that R_State(t') is R_ReceivePend. R_machine(**C**) copies the data, creates a new link **n**, sets its destination to m', and sends K_Update to **K(B)** including a description of n. R_State(t') becomes R_WaitUpdate. The linking situation at this stage is depicted in Figure 6 (a).

(5) Assuming that m' is OK, M_machine(**B**) changes the destination of m' to n and replies K_Update_OK to **K(C)**.

(6) M_machine(**C**) marks n as a regular link, adds it to links owned by **C**, and responds K_Accepted to **K(A)**. The situation at this stage is depicted in Figure 6 (b). When **C** calls **Wait** that matches this **Receive**, a new link number (user-relative, corresponding to n) is returned.

(7) On getting K_Accepted, S_machine(**A**) sets S_Result and S_Done of t appropriately, and frees link m.

```
 ----------          ----------         ----------           ----------
|          |   m'   |          |       |          |         |          |
| User A   |<------>| User B   |       | User A   |<--*     | User B   |
|__________| m      |__________|       |__________| m       |__________|
     /\  (L_Moving)                         /\  (L_Moving)        /\
      \                                      \                    /
       \ t (S_EncloseOut)                     \ t(S_EnclOut)     / m'
        \                                      \
         \          *                           \               /
          \        /                             \             /
(R_Wait t' \      / n (L_Fetus)       (R_Done) t' \           / n (L_OK)
Update)    _\/___\/__                             \/_________\/
           |         |                            |           |
           | User C  |                            | User C    |
           |_________|                            |___________|

              (a)                                      (b)
```

Figure 6: **Move - Intermediate Stages**

4.2.3. Destroy-Move conflicts

Many interactions between the **Destroy** and **Move** mechanisms shape both protocols. We examine four cases in which the moving or transferring link are requested to be destroyed by each owner. We will refer back to Figure 6, which defines the processes and links involved.

Case I: **A** calls **Destroy**(m). This request is an error, since m is no longer owned by **A** (though in rare cases **Move** may fail and m be reinstalled).

Case II: **B** calls **Destroy**(m′). Assume the **Destroy** request happens before **K(B)** has gotten K_Update from **K(C)**, as in Figure 6 (a). (Otherwise, it is a normal case and K_Destroy is sent to **K(C)**.) D_machine(**A**) can act in one of the following ways:

(1) Refuse the K_Destroy request from **K(B)**, so **K(B)** will send it instead to **K(C)** once it gets the K_Update request.

(2) Delay the K_Destroy request from **K(B)** until the **Move** fails, which it will, since **K(B)** will respond K_UpdateFail to **K(C)**. Then grant the destroy.

(3) Delay the request and try to **Cancel** the **Send**. When **K(C)** responds K_CancelOK or K_Rejected, grant the destroy.

(4) Grant the destroy with no delay.

Alternative (4) is least preferable, since **K(B)** may free link m′ before K_Update is received, at which time this latter message may confuse it. Alternatives (2) and (3) are preferable, since **Destroy** is guaranteed to be granted eventually. Both alternatives (1) and (2) delay until **C**'s **Receive**, which may not happen for a very long time. Hence, alternative (3) was chosen: D_machine(**A**) tries to abort the **Send** on behalf of **A** (and interferes with S_State). This abort is implemented as regular **Cancel**(S), except that on its completion, the user is not notified and unblocked [12]. K_Destroy_OK for m′ is sent to **K(B)**. Finally, **Send**(t...) fails with result "Send failed, moving link destroyed".

Case III: **A** calls **Destroy**(t). We considered four alternatives for D_machine(**A**):

(1) Consider it an error. We rejected this action, since **Destroy** should always succeed, even when outstanding **Send** exists.

(2) Delay it until **Move** completes. We rejected this action, since it must wait for **C**'s **Receive**.

(3) **Cancel** the **Send** as in the previous problem. Although this alternative is sufficient, the next is better.

(4) Transmit K_Destroy to **K(C)**, and let it decide whether to grant the request immediately or delay it. If **C**'s **Receive** hasn't occurred yet, or if the Update process has completed, K_Destroy_OK may be sent back to **K(A)**. Otherwise, **K(C)** will delay its response until K_Update_OK or K_UpdateFail is accepted from **K(B)**. Then **K(C)** grants the destroy request. This delay introduces a new link state in **K(C)**, L_RemoteDestroy. Link t′ remains in this state until the destroy can be granted.

Case IV: **C** calls **Destroy**(t′). As shown above, there are several alternatives. We show here only the solution adopted: At **C**'s end the **Update** protocol may be (i) completed, (ii) not yet started, or (iii) in progress. In the first two cases, there is no delay; they are handled as usual destroy cases. (In the second case, link m is reinstalled as **A**'s regular link.) In the third case, D_machine(**C**) should postpone the K_destroy request for t′ until the Update process (though involving different links) is completed, issuing that request together with the K_Accepted or K_Rejected response. A new link state arises, which we mark with an exception flag: LocalDestroyPend.

In rare situations both ends of a link might be moving. The protocol must be designed to cope with such situations. Up to four different kernels might be involved. Moreover, the situation at one end could change (for instance, a send may be cancelled), and we want the situation at the other end to adjust in minimum delay and minimum messages.

A solution based on *voting* protocols is unacceptable, since it might require too many messages (since not all kernels know what situation they and their peers are in.) A solution based on *exponential backoff* algorithms is unacceptable, since it complicates the protocol with timeouts, may incur too many messages and may lead to "livelock" situations. Our protocol (see Table A.4) breaks the tie

[12] More complex situations may arise, if **A** tries to **Cancel** the enclosure, or **Destroy** link t.

by letting one end to complete successfully while failing (temporarily) at the other end. When failure is reported (a K_Rejected message is received by the S machine. See Table A.2), the failed send is retried with the new destination for the moving link. The situation is thus reduced to one end moving, and will succeed (unless the new owner of the other end moves or destroys its end meanwhile.) Retry is not immediate when K_Rejected is processed, since the K_Update (from the link end that should succeed) might not yet been received. Both Move and Send FSAs need to be augmented to retry the Send when both K_Rejected and K_Update are heard. A deadlock may arise if K_Update is never heard, which can happen if the transferring link is destroyed. Our protocol prevents this hazard by reporting K_NoUpdate to tell the failing end to retry the enclosure. Of course, the users at all ends are oblivious to this negotiation between kernels.

These unfortunate situations may arise concurrently. It is an interesting exercise to verify that our protocol can cope with such extreme cases, with, what we believe, minimum or no penalty for a ''normal'' **Move** or **Destroy** mechanisms.

4.2.4. Receive / Send FSAs - the Third Dimension

The correct action for an FSA to take when a request arrives depends on three dimensions: the type of request (like U_Receive), the state of the link (like R_Idle), and the situation of the link (like *UpdatePend*). Usually, this third dimension is unnecessary; our tables have only shown the first two dimensions. In order to capture the third dimension, we could expand the table by placing entries for each possible link situation. This expansion is not usually needed, since most situations are either illegal or can be handled by the general case. Instead, we maintain a set of flags representing the situation for each link. The table entry describes what to do in the usual situation, that is, when no flags are set. If any flags are set (in particular, if the flag *NotOK* is set), then an alternative action may be taken. This alternative might require a careful investigation of the exact flags, so it might be inefficient. This use of alternatives is represented graphically in Appendix A (Tables A.1, A.2) by separating each box into two layers.

4.3. Augmented Send and Receive

We discuss here two features we implemented beyond the regular **Send** and **Receive** discussed so far: A **Receive**(*AllLinks*) and unrestricted buffer size. We will also touch on other features that were delayed for future consideration.

4.3.1. Receive(AllLinks)

The **Receive** call allows the link argument to be *AllLinks*. Such a call embraces all user's links and remains pending until it can be bound to one specific link. We disallow coexistence of **Receive**(*AllLinks*) with any other **Receive**, since otherwise semantic confusion may arise. For example, when data arrives on link L, and both **Receive**(*AllLinks*) and **Receive**(L) are in progress, we must decide which **Receive** to bind the incoming data to. Arbitrary semantics are unacceptable. Deciding in favor of the more restrictive **Receive** (that is, the latter one in this case) can be expensive. If the more general one is **Wait**ed for first, the data must be copied from the specific buffer back into the general one. Until that point, the sender cannot be informed of completion, because the eventual buffer may be too small, even though the preferred buffer is not. Similarly, a more restrictive **Receive** may be requested after the general one has accepted a message, which then has to be copied into the new buffer.

Therefore, **Receive**(*AllLinks*) first verifies that there are no other **Receive** calls in progress. (They are in progress until **Wait** succeeds.) If there is a choice, **Receive**(*AllLinks*) is preferentially bound to the link least recently bound. Otherwise, it may be bound to any link that shows any incoming activity (including notice of remote destruction). Once it is bound, **Receive**(*AllLinks*) is handled by the R_machine as a regular **Receive**.

4.3.2. Unrestricted buffer size

Programs sometimes want to transfer large chunks of data. This situation is particularly common in loading new programs. Even though our hardware limits packets to 2KB, Charlotte does not restrict buffer size. Instead, it breaks long messages into packets that are transferred one at a time.

No change is needed to the previously described protocol when both ends of the link reside in the same node. Otherwise, a two-phase protocol (three-phase in case of enclosure) is used: The first packet is as before, except that the only as many bytes as fit in a packet are sent. If the receiver's buffer can hold more data, the receiver's R machine sends a K_SendMore response (instead of K_Accepted, the normal response).[13] Both sides move to state R_/S_PartialDone.

The second phase is accomplished by the ToNugget task, delivering packet after packet as fast as it can. The receiving end replies K_Accepted only when the last packet is accepted correctly.

Destruction of the link may be requested by either side while the packet delivery is under way. In this case, the D_machine (of the sender) first tells ToNugget to abort the transfer and then carries on with link destruction.

Send with enclosure may also use large buffers, even though our experience shows that messages with enclosures are often quite short. If there is an enclosure, the update procedure starts after data are accepted. At the end of the second phase, the R_machine doesn't respond K_Accepted to the sender, but requests K_Update from the new link's destination, as described in Section 4.2.2. Alternatively, we could start the update procedure while data are being transferred. However, should the transferring link be destroyed before transmission completes, but after the moved link is installed successfully, we might need to undo the link movement (following the principle that the entire transaction should succeed or fail). At the end of this phase, K_Accepted or K_Rejected is delivered to the sender, as previously.

4.3.3. Other versions of Send/Receive

Asynchronous notification of **Receive** or **(Send)** interrupts the user upon completion. As long as the user is handling this interrupt, new interrupts are queued in FCFS order. Kernel calls may be disallowed while the user is in interrupt mode. For example, it may be blocked on **Wait**, **Cancel** or **Destroy** when interrupted. If we allow the handler to call these or **Send** or **Receive**, contradictions arise. Disallowing some calls introduces extra checking for every regular call. A compromise is to implement efficiently non-restrictive interrupt-handler calls on top of the existing FSAs. We have deferred this service for further re-evaluation.

Remote procedure call is not directly supported by Charlotte, which has no send-receive-reply paradigm. Instead, two pairs of **Send** and **Receive** are needed. Charlotte therefore incurs two extra acknowledgements (at the completion of each **Receive**). However, our **Send-Receive** paradigm is more general and serves better when no replies are needed or the replies should come in a different order from the requests. Implementing both paradigms is possible by adding new states and message types to the existing FSAs, but the added overhead may outweigh the gains. Since we cannot predict the frequency of usage of either paradigm, we defer this issue for further evaluation.

5. Conclusions

Charlotte is important for several reasons.

- It demonstrates the utility of the Crystal design.
- It addresses resource-allocation and inter-process communication issues that are central to distributed operating systems.

If the matching **Receive** is requested after the first packet has already been discarded, the receiver's R machine sends K_SendAgain, as always. No additional K_SendMore is needed.

- It allows distributed, computationally-intensive applications to be designed and tested.
- It packages a complex inter-machine protocol in a modular and elegant form.

We have learned several lessons from building Charlotte version 2. We were surprised at the subtle manner in which seemingly unrelated and simple communication facilities interact to produce complexity. Modularity, particularly the finite-state automaton design, allowed us to tackle that complexity, although at a significant performance cost. Modifications to improve performance were implemented after the finite-state automata were designed. Following and extending methods like [14, 16], we use states and simple actions for the normal (most frequent) cases, and a small number of flags for special cases. We are considering faster synchronization mechanisms than queues for common cases. Again, this change will improve performance at the expense of elegance.

In this paper, we have shown the development from a simple and restrictive protocol to one that covers all our inter-process communication semantics, including complex and timing-dependent scenarios. Building the kernel modularly aided in adding new features and services. For example, destroying the links of a terminating process or links to a failed node was a simple addition to normal link destruction.

The hardest part of Charlotte development was protocol definition. We defined and tested the protocol manually by drawing next-to-impossible timing diagrams and using both well behaved and crazy programs. Better (automated) verification techniques and an exhaustive simulation (or event genrator) are needed. We have started a project to examine automatic generation of finite state automata from protocol descriptions [17].

Charlotte is currently functioning as reported in this paper. Current research revolves around programming languages that make use of links, dynamic migration for load balancing, and distributed debugging [18, 19].

Acknowledgements

The Charlotte project owes its success to a great number of designers, implementers, and creative critics. Phil Krueger and Al Michael implemented Charlotte version 1. Bryan Rosenburg and Bill Kalsow were instrumental in designing the finite-state automata. Bryan also built the first versions of most of the utility processes. They have been modified and improved by Prasun Dewan, Vinod Kumar, and Cui-Qing Yang. Tom Virgilio implemented the Crystal nugget. Keith Thompson and Nancy Hall implemented the Modula compiler. Many helpful ideas and comments were provided by Aaron Gordon and Michael Scott. The idea of the connector is due to Tony Bolmarcich. Marvin Solomon is a principal designer of the entire Charlotte operating system and the Crystal project.

Appendix A: The four basic FSAs

The following is a complete version of the FSAs, including the **Cancel** and **Move** operations and Buffer Management, but without **Receive**(*AllLinks*) and multi-packet transfers. These latter are discussed in 4.3. For completeness, the reader is referred to the **Wait** algorithms described in 4.1. New operations, and some which are functionally extended from what was described earlier, are elaborated following the four FSAs.

R_State	User Request		Kernel to Kernel		
	U_Receive	U_Cancel Recv	K_Send	K_Enclose	K_Cancel[14] Send
1.R_Idle	A:3 Ax:1(3,4)	x1:1 x1:1	D:2 dx:1	D:2a dx:1	N:1 Nx:1
2.R_SendIn	B:4(5) X	G:2 X	X X	X X	J:1(4) Jx:1
2a.R_EncloseIn	P:7(5) X	G:2a X	X X	X X	J:1(4) Jx:1
3.R_ReceivePend	C:3 Cx:3	H:1 Hx:1	E:4 Dx:4	R:7 Dx:4	N:3 Jx:3
4.R_Done	A:3 ax:4(3)	I:4 Ix:4	D:2 Dx:4	D:2a Dx:4	N:4 Jx:4
5.R_ReceiveAgain	C:5 Cx:5	L:6 Lx:6	F:4 Fx:4	R:7 Dx:4	N:3 nx:3
6.R_Cancel_Recv	X X	X X	K:4 Kx:4	S:7 X	M:1 X
7.R_WaitUpdate	C:7 cx:7	I:7 ix:7	X X	X X	N:7 Tx:7

Table A.1: **Receive FSA with Cancel, Move and buffer management**

In each entry, the lower (Action, NewState) pair is selected on exceptional cases:

LinkState < > L_OK (* L_Dead, L_Moving, L_Fetus, L_Remote/Local Destroyed *)
or
LinkFlags < > 0 (* UpdatePend, UpdateCompleted, UpdateRefused, LocalDestroyPend, ReturnedToSender *)

Actions:

A **skip**.

Ax **if** LinkIsMoving(cl) **then** x2
elsif LinkIsDead(cl) **then**
StoreResult(cl,R,ReceiveFail_LinkDestroyed);
R_State := R_Done
elsif LinkUpdatePends(cl) **then** A; R_State := R_ReceivePend
else FatalError; **fi**.

ax As Ax, starting from **elsif**...

B **if** BufferIsAvail (cl) **then**
AcceptData (cl);

[14] This request might be obsolete, and hence is handled as a *hint*. It is considered obsolete when Link's current destination doesn't agree with requestor's address (because link has been moved meanwhile, for example), in which case it is ignored; otherwise, it is handled by the appropriate action.

StoreResult (cl,R,ReceiveOk);
Out (cl, K_Accepted)
else
Out (cl, K_SendAgain);
R_State := R_ReceiveAgain
fi.

C KeepRequest (cl, NextReceive); BlockUser (cu).

Cx **if** LinkUpdatePends(cl) **then** C **else** FatalError; **fi**.

cx **if** LinkIsRemoteDestroyed(cl) **then** C (*though it will fail*) **else** FatalError; **fi**.

D FreeBuffer (full).

Dx D; **if not** LinkIsLocalDestroyed(cl) **then** FatalError; **fi**.

dx D;
if LinkIsMoving(cl) **then**
Out(cl,K_SendLater);
Set L_Flag(ReturnedToSender)
elsif not LinkIsLocalDestroyed(cl) **then** FatalError; **fi**.

E Like **then**-part of B.

F AcceptData (cl); StoreResult (cl,R,ReceiveOk);

Fx **if** LinkUpdatePends[15] **then** F; PossibleUpdateOK(cl); **else** Dx; **fi**.

G **if** R_Result is valid **then** I **else** x1 **fi**.

H ReturnToUser(CancelOK).

Hx **if** LinkUpdatePends **then** H; **else** FatalError; **fi**.

I ReturnToUser(TooLateToCancel).

Ix **if** LinkIsDead(cl) **or** LinkUpdatePends(cl) **then** I **else** FatalError; **fi**.

ix **if** LinkIsRemoteDestroyed(cl) **then** I **else** FatalError; **fi**.

J Out(cl, K_CancelOK); **if** R_Result is valid **then** R_State := R_Done **fi**.

Jx **if not** LinkIsLocalDestroyed(cl) **then** FatalError; **fi**.

K F; I; UnblockUser(cu).

Kx **if** LinkUpdatePends(cl) **then** K; PossibleUpdateOK(cl); **else** FatalError; **fi**.

L BlockUser(cu).

Lx **if** LinkUpdatePends(cl) **then** L;[16] **else** FatalError; **fi**.

M H; UnblockUser(cu).

N **skip**.

Nx **if** LinkIsMoving(cl) **then**
if ReturnToSender $\in$ L_Flags **then** Reset L_Flag(ReturnToSender); **fi**.
else Jx; **fi**.

nx **if** LinkUpdatePends(cl) **then** N; PossibleUpdateOK(cl); **else** Jx; **fi**.

---

[15] This case may theoretically happen if this sequence of events occurs: (i) The local side requires K_SendAgain, (ii) which is responded with K_Send. (iii) The remote side encloses the link to a third party (iv) whose K_Update request we received before the above K_Send.

[16] Like the case in Fx, but the remote side could respond with K_Cancel.

P **if** BufferIsAvail (cl) **then**

```
        AcceptData (cl);
        StartNewLink(nl);   (* nl = New Link *)
        Out (nl, K_Update)
```

else

```
        Out (cl, K_SendAgain);
        R_State := R_ReceiveAgain
```

fi.

R Like the **then**-part of P.

S R; I; UnblockUser(cu).

Tx **if not** LocalDestroyPends(cl) **then** FatalError; **fi**.

x1 ReturnToUser(NothingToCancel).

x2 ReturnToUser(LinkIsMoving).

S_State	User Request		
	U Send	U Enclose	U Cancel Send
1.S_Idle	A:2 Ax:3(1,2)	I:2a(1) Ix:3(1,2a)	x1:1 x1:1
2.S_SendOut	B:2 Bx:2	B:2 Bx:2	E:4 Ex:1(4)
2a.S_EncloseOut	B:2a Bx:2a	B:2a Bx:2a	E:4a ex:1(4a)
3.S_Done	A:2 ax:3(2a)	I:2a(1) ix:3	F:3 Fx:3
4.S_Cancel_Send	X X	X X	X X
4a.S_Cancel_Enclose	X Sx:4a	X Sx:4a	X Tx:4a

S_State	Kernel to Kernel				
	K_Accepted	K_Rejected	K_Send Again	K_Cancel OK	K_Send Later
1.S_Idle	X X	X X	X X	X X	X X
2.S_SendOut	C:3 Cx: 3	X X	D:3 Dx:3	X X	J:2 Jx:2
2a.S_Enclose Out	M:3 Mx:3	N:2a Nx:2a	L:2a Hx:2a	X X	J:2a jx:2a
3.S_Done	X X	X X	X X	X X	X X
4.S_Cancel_Send	G:3 Gx:3	X X	H:1 Hx:1	H:1 Hx:1	H:1 Hx:1
4a.S_Cancel_Enclose	R:3 Rx:3	P:1 px:1(3)	P:1 Px:1(3)	P:1 px:1(3)	P:1 px:1(3)

Table A.2: **Send FSA with Cancel, Move and buffer management**

In each entry, the lower (Action, NewState) pair is selected on exceptional cases:

LinkState < > L_OK (* L_Dead, L_Moving, L_Fetus, L_Remote/Local Destroyed *)
or
LinkFlags < > 0 (* UpdatePend, UpdateRefused, UpdateCompleted, LocalDestroyPend, ReturnedToSender *)
or
SendFlags < > 0 (* SendLater, AbortEnclose, EncloseRejected *)

Actions:

A Out (cl, K_Send).

Ax **if** LinkIsMoving(cl) **then** x2; S_State := S_Idle;
elsif LinkIsDead(cl) **or** LinkIsRemoteDestroyed(cl) **then**
StoreResult(cl, S, SendFail_LinkDestroyed);

```
      elsif LinkUpdatePends(cl) then
              S_State := S_SendOut;
              Set S_Flag(SendLater)
      else FatalError; fi.

ax    Like Ax, starting from elsif...

B     KeepRequest (cl,NextSend); BlockUser (cu).

Bx    if LinkIsRemoteDestroyed(cl) then (*Discard this request*)
      elsif LinkUpdatePends(cl) or ToBeSentLater(cl) then B
      else FatalError; fi.

C     StoreResult (cl,S,SendOk).

Cx    if LinkIsLocalDestroyed(cl) or LocalDestroyPends(cl) then skip
      elsif LinkUpdatePends(cl) then C; PossibleUpdateOK(cl);
      else FatalError; fi.

D     A; C.

Dx    if not (LinkIsLocalDestroyed(cl) or LocalDestroyPends(cl)) then FatalError; fi

E     Out(cl, K_Cancel_Send); BlockUser(cu).

Ex    if LinkIsRemoteDestroyed(cl) then          ReturnToUser(CancelOK)
      elsif ToBeSentLater(cl) then
              ReturnToUser(CancelOK);
              Reset S_Flag(SendLater);
      elsif LinkUpdatePends(cl) then
              S_State := S_CancelSend;
              BlockUser(cu);
      else FatalError; fi.

ex    if LinkIsRemoteDestroyed(cl) then
              ReturnToUser(CancelOK);
              ReInstallLink(ml);
      elsif ToBeSentLater(cl) then
              ReturnToUser(CancelOK);
              Reset S_Flag(SendLater);
              ReInstallLink(ml);
      elsif EncloseRejected(cl) then
              ReturnToUser(CancelOK);
              Reset S_Flag(EncloseRejected);
              ReInstallLink(ml);
      orif LinkUpdatePends(cl) then (* last 2 situations may coexist *)
              S_State := S_CancelSend;
              BlockUser(cu);
      else FatalError; fi.

F     ReturnToUser(TooLateToCancel).

Fx    if LinkIsDead(cl) or LinkUpdatePends(cl) or LinkIsRemoteDestroyed(cl) then F
      else  FatalError; fi.

G     C; F; UnblockUser(cu).
```

```
Gx   if LinkUpdatePends(cl) then G; PossibleUpdateOK(cl); else FatalError; fi.

H    ReturnToUser(CancelOK); UnblockUser(cu).

Hx   if LinkUpdatePends(cl) then H; PossibleUpdateOK(cl); else FatalError; fi.

I    if OkToMove(ml) then
             ReturnToSender(ml);
             ml^.L_State := L_Moving;
             Out(cl, K_Enclose);
     else
             x3; S_State := S_Idle;
     fi.

Ix   if LinkIsMoving(cl) then x2; S_State := S_Idle;
     elsif LinkIsDead(cl) or LinkIsRemoteDestroyed(cl) then
             StoreResult(cl, S, SendFail-LinkDestroyed);
     elsif LinkUpdatePends(cl) then
             if OkToMove(ml) then
                     S_State := S_EncloseOut;
                     Set S_Flag(SendLater);
                     ReturnToSender(ml);
                     ml^.L_State := L_Moving;
             else
                     x3; S_State := S_Idle;
             fi.
     else FatalError
     fi.

ix   Like Ix, starting from the first elsif.

J    Set S_Flag(SendLater);

Jx   J;
     if LinkUpdatePends(cl) then
             PossibleUpdateOK(cl);
             if it succeeds then Out(cl, K_Send); fi.
     else  Dx;  fi

jx   Like Jx, but Out(cl, K_Enclose).

L    Out(cl, K_Enclose);

M    StoreResult(cl,S,SendOK);
     if LinkUpdatePends(ml) then Out(ml, K_UpdateFail); fi.
     DeleteLink(ml).

Mx   M;
     if LinkUpdatePends(cl) then PossibleUpdateOK(cl)  else Dx; fi.

Q    if LinkUpdateCompleted(ml) then RetryMove(cl); else Set S_Flag(cl, EncloseRejected); fi.

Qx   if LinkUpdatePends(cl) then
             PossibleUpdateOK(cl); if it succeeds then Q;  fi.
             if LinkUpdateCompleted(ml) then RetryMove(cl);
             else Set S_Flag(cl, EncloseRejected); fi.
     else  Dx;  fi

P    ReInstallLink(ml); H.
```

```
Px    if LinkIsRemoteDestroyed(ml) then
              StoreResult(cl, S, SendFail-EnclosedLinkDestroyed);
              S_State := S_Done;
              GrantDestroy(ml);
      else ReInstallLink(ml);
      fi.
      if not AbortEnclose ∈ S_Flags then H else Dx; fi.

px    if LinkUpdatePends(cl) then PossibleUpdateOK(cl); else Px; fi.

R     M; F; UnblockUser(cu).

Rx    M; Dx.  (* ml cannot be RemoteDestroyed *)

Sx    if AbortEnclose ∈ S_Flags then B else FatalError; fi.

Tx    if AbortEnclose ∈ S_Flags then
              Reset S_Flag(AbortEnclose); BlockUser(cu);
      else FatalError; fi.

x1    ReturnToUser(NothingToCancel).

x2    ReturnToUser(LinkIsMoving).

x3    ReturnToUser(LinkNotEnclosable).
```

Link State	Request		
	U Destroy	K Destroy	K Destroy OK
1.L_OK	A:2(1)	C:3(4)	X
2.L_LocalDestroyed	X	D:6	D:6
3.L_Dead	B:6	X	X
4.L_RemoteDestroyed	E:4	X	X
5.L_Moving	x1:5	F:4	X
6.L_Free	X	X	X

Table A.3: **Destroy FSA**.

Actions:

```
A   BlockUser(cu);
    if R_State = R_WaitUpdate then (* Delay request *)
            L_State := L_OK;
            Set L_Flag(LocalDestroyPend);
    else Out(cl,K_Destroy); fi.
    if LinkUpdatePends then
            Reset L_Flag(UpdatePend);
            Out(to-updating-destination, K_UpdateFail);
    fi.

B   DisposeLink(cl)

C   if R_State = R_WaitUpdate then L_State := L_RemoteDestroyed;
    else GrantDestroy(cl);
    fi.

D   ClearDeadLink(cl); DisposeLink(cl): UnblockUser(cu)

E   Set L_Flag(LocalDestroyPend);

F   if tl^.S_State = S_CancelEnclose then (* tl = Transferring Link *)
            (* no cancellation needed *)
    elsif ToBeSentLater(tl) or EncloseRejected(tl) then
            tl^.S_State := S_Done;
            StoreResult(tl, S, SendFail-EnclosedLinkDestroyed);
            GrantDestroy(cl);
    else
            tl^.S_State := S_CancelEnclose;
            if not LinkIsLocalDestroyed(tl) then Out(tl,K_Cancel_Send); fi.
            Set tl^.S_Flag(AbortEnclose);
    fi.

x1  ReturnToUser(LinkIsMoving).
```

Link State	Request			
	K_Update	K_UpdateOK	K_UpdateFail	K_NoUpdate
1.L_OK	A:1	X	X	F:1
2.L_LocalDestroyed	B:2	X	X	N:2
3.L_Dead	X	X	X	N:3
4.L_RemoteDestroyed	X	X	X	N:4
5.L_Moving	C:5	X	X	H:5
6.L_Free	X	X	X	N:6
7.L_Fetus	G:7	D:1	E:6	N:7

Table A.4: **Move FSA.**

A **if** R_State = R_ReceiveAgain **then**
  Set L_Flag(UpdatePend);
 elsif S_State ∈ {S_Idle,S_Done} **then**
  Set new destination of link cl.
  Out(cl, K_UpdateOK);
 else PossibleUpdateOK(cl);
 fi.

B Out(updateing-destination, K_UpdateFail);

C **if** MyPriority()[17] **then**
  Set L_Flag(UpdateRefused);
  Out(cl, K_UpdateFail);
 else
  Set new destination of link cl;
  Set L_Flag(UpdateCompleted);
  Out(cl, K_UpdateOK);
  if EncloseRejected(tl) **then** RetryMove(tl); **fi**. (* tl = Transferring Link *)
 fi.

D InstallNewLink(cl);
 Find the receiving link (rl);
 Out(rl, K_Accepted);
 rl^.R_State := R_Done;
 StoreResult(rl, R, ReceiveOK); (* with indication of the new accepted link *)
 Perform d.

d **if** LinkIsRemoteDestroyed(rl) **then**
  Out(rl, K_DestroyOK);
  ClearDeadLink(rl);
  if LocalDestroyPends(rl) **then**
   DisposeLink(rl);
   UnblockUser(rl);
  else rl^.L_State := L_Dead; **fi**.
 elsif LocalDestroyPends(rl) **then**
  rl^.L_State := L_LocalDestroyed;
  Reset rl^.L_Flag(LocalDestroyPend);
  Out(rl, K_Destroy);

[17] This is a simple function which should return **T** for one end of the link, and **F** for the other (eg: comparing the pair (MachineId, LinkId) of both ends.)

fi.

E DisposeNewLink(cl);
Find the receiving link (rl);
Out(rl, K_Rejected);
rl^.R_State := R_ReceivePend;
Perform d.

F **if** (Request is not obsolete)[18] **then**
if S_State = S_SendOut **then** Out(cl, K_Send)
elsif S_State = S_EncloseOut **then** Out(cl, K_Enclose)
else skip;
fi.
Reset S_Flag(SendLater);
else skip
fi.

G Set L_Flag(UpdatePend).

H **if** (Request is not obsolete) **then**
if EncloseRejected(tl) **then** RetryMove(tl); **else** Set L_Flag(UpdateCompleted); **fi**.

N **skip**.

We have introduced some new operations:

LinkIsMoving(link) **return**(link^.L_State = L_Moving).

LinkIsDead(link) **return**(link^.L_State = L_Dead).

LinkIsLocal/RemoteDestroyed(link)
return(link^.L_State = L_Local/RemoteDestroyed).

LocalDestroyPends(link)
return(LocalDestroyPend ∈ link^.L_Flags).

LinkUpdatePends(link) **return**(UpdatePend ∈ link^.L_Flags).

LinkUpdateCompleted(link)
return(UpdateCompleted ∈ link^.L_Flags).

LinkUpdateRefused(link)
return(UpdateRefused ∈ link^.L_Flags).

ToBeSentLater(link) **return**(SendLater ∈ link^.S_Flags).

EncloseRejected(link) **return**(EncloseRejected ∈ link^.S_Flags).

StartNewLink(nl) opens a new entry (nl) in link table, sets its destination end appropriately, keeps mutual pointers w/ the receiving link (current link).
nl^.L_State := L_Fetus.

PossibleUpdateOK(link)
check whether all conditions to postpone an K_Update request are resolved, namely (R_State < > R_ReceiveAgain) **and** (S_State = {S_Idle **or** S_Done} **or** SendLater ∈ S_Flags)

OkToMove(link) **return**(L_State = L_OK **and** S_State = S_Idle
and R_State ∈ {R_Idle,R_SendIn,R_EncloseIn}

---

[18] This request reflects failure to move the link by the other side, but theoretically it may come after the link has been consecutively moved (or been failed to move) even more than once. Therefore, it is handled as a *hint*, rather than as an *absolute*. This request is <u>not obsolete</u> when: (1) Link's Destination agrees with the address of this request sender, and (2)

```
                    and R_Result is not valid.
```

RetryMove(link)

```
Reset link^.S_Flag(EncloseRejected);
Reset ml^.L_Flag(UpdateCompleted); (* moving link *)
Out (link, K_Enclose);  (* with new destination address of ml *)
```

ReturnToSender(link)

```
if R_State ∈ {R_SendIn, R_EncloseIn} then
        Out(link, K_SendLater);
        Set L_Flag(ReturnedToSender);
        R_State := R_Idle;
fi.
```

InstallNewLink(link)

```
add the link to user's set of links.
if LinkUpdatePends(link) then Out(to-new-destination, K_UpdateOK); fi.
```

ReInstallLink(link)

```
if LinkIsMoving(link) then
        if LinkUpdateCompleted(link) then Reset L_Flag(UpdateCompleted); fi.
        if LinkUpdateRefused(link) then
                Reset L_Flag(UpdateRefused);
                Out(link, K_NoUpdate);
        elsif ReturnedToSender ∈ L_Flags then
                Reset L_Flag(ReturnedToSender);
                Out(link, K_NoUpdate);
        fi.
        L_State := L_OK;
elsif LinkIsRemoteDestroyed(link) then
        GrantDestroy(link);
fi.
```

GrantDestroy(link)

```
        Out(link, K_DestroyOK); L_State := L_Dead;
        ClearDeadLink(link);
```

ClearDeadLink(link) In addition to breaking any outstanding **Send** or **Receive** (see Section 4.1.4), break outstanding **Cancel**. If that **Cancel** was user-initiated, unblock the user with CancelOK. If enclosure is broken, ReInstallLink(moving link).

6. References

1. R. Finkel, M. Solomon, D. DeWitt, and L. Landweber, "The Charlotte Distributed Operating System: Part IV of the first report on the crystal project," Technical Report 502, University of Wisconsin--Madison Computer Sciences (July 1983).
2. Proteon Associates, "Operation and Maintenance Manual for the ProNet Model p1000 Unibus," *Waltham, Mass*, (1982).
3. R. Cook, R. Finkel, D. DeWitt, L. Landweber, and T. Virgilio, "The crystal nugget: Part I of the first report on the crystal project," Technical Report 499, Computer Sciences Department, University of Wisconsin (April 1983).
4. R. A. Finkel, M. H. Solomon, and R. Tischler, "Arachne User Guide, Version 1.2," Technical Summary Report 2066, University of Wisconsin Mathematics Research Center (April 1980).
5. M. H. Solomon and R. A. Finkel, "The Roscoe distributed operating system," *Proc. 7th Symposium on Operating Systems Principles*, pp. 108-114 (December 1979).

SendLater ∈ SendFlags (Hence S_State ∈ {S_SendOut, S_EncloseOut}).

6. F. Baskett, J. H. Howard, and J. T. Montague, ''Task communication in Demos,'' *Proc. 6th Symposium on Operating Systems Principles*, pp. 23-31 (November 1977).

7. M. L. Powell and B. P. Miller, ''Process migration in DEMOS/MP,'' *Proc. 9th Symposium on Operating Systems Principles*, pp. 110-119 (December 1983).

8. J. D. Ichbiah et al., ''Preliminary Ada reference manual,'' *Sigplan Notices* **14**(6)(June 1979).

9. P. Brinch-Hansen, ''Distributed processes: A concurrent programming concept,'' *CACM* **21**(11) pp. 934-941 (November 1978).

10. T. W. Mao and R. T. Yeh, ''Communication port: A language concept for concurrent programming,'' *IEEE Trancactions on Software Engineering* **SE-6**(2) pp. 194-204 (March 1980).

11. D. R. Cheriton and W. Zwaenepoel, ''The distributed V kernel and its performance for diskless workstations,'' *Proc. 9th Symposium on Operating Systems Principles*, pp. 110-119 (December 1983).

12. H-T Chou, D. J. DeWitt, R. Katz, and T. Klug, ''Design and Implementation of the Wisconsin Storage System (WiSS),'' Technical Report #524, Computer Sciences Department, University of Wisconsin (November 1983).

13. A. Danthine and J. Bremer, ''An Axiomatic Description of the Transport Protocol of Cyclades,'' *Professional Conference on Computer Networks and Teleprocessing*, (March, 1976).

14. G. V. Bochmann and J. Gescei, ''A Unified Method for the Specification and Verification of Protocols,'' *Information Processing*, IFIP, North-Holland, (1977).

15. G. V. Bochmann, ''Finite State Description of Communication Protocol,'' *Computer Networks* **2** pp. 361-372 (1978).

16. A. Danthine and J. Bremer, ''Modelling and Verification of End-to-End Transport Protocols,'' *Computer Networks* **2** pp. 381-395 (1978).

17. B. Rosenburg, ''Automatic generation of communication protocols,'' Thesis proposal, University of Wisconsin--Madison Computer Sciences (March 1984).

18. A. J. Gordon and R. A. Finkel, ''The use of timing graphs for distributed program debugging,'' *Distributed processing technical newsletter*, (Fall 1984).

19. M. L. Scott and R. A. Finkel, ''LYNX: A Dynamic Distributed Programming Language,'' To appear in the *1984 International Conference on Parallel Processing*, (August 21-24, 1984).