The Parallel Protocol Framework

by
Murray W. Goldberg
Gerald W. Neufeld
Mabo R. Ito

Technical Report 92-16
August 1992

Department of Computer Science
University of British Columbia
Rm 333 - 6356 Agricultural Road
Vancouver, B.C.
CANADA V6T 1Z2

The Parallel Protocol Framework

By

Murray W. Goldberg, Gerald W. Neufeld, and Mabo R. Ito

University of British Columbia, Vancouver, British Columbia, Canada

Contents

1 Introduction

1.1 What Is a Protocol Framework?

PPF (Parallel Protocol Framework), like most other protocol frameworks (eg. [HP88]), defines an implementation and execution environment for communication protocols. There are two parts to the service provided by the framework. The first part is a set of structural guidelines which determine protocol implementation details. These are enforced[1] through written documentation and library modules. Common examples of structural guidelines include the format of communication between protocol modules or layers, and the structure of the protocol state machines. The second part of any protocol framework service is a set of library routines to perform common protocol functions[2]. These routines include buffer management, timer management, and inter-layer communication primitives.

There are several advantages in using such a framework when implementing communication protocols. One advantage is code reuse. As indicated, primitives are provided by the framework for common protocol functions. The usual code-reuse arguments apply well to communication protocols due to their complexity and performance sensitive nature. Protocol implementation is faster because the common functions are provided by the framework. Also, it is worthwhile to expend significant effort on library efficiency since the libraries are used by several protocol implementations. A final advantage comes from the consistent format imposed on the protocol implementation. Coding consistency enhances maintainability, readability and extensibility. A third advantage of using a well designed protocol framework (somewhat related to the first advantage) is structural efficiency of the protocol implementation. For example, most frameworks impose a certain method of inter-layer communication. If the communication structure and primitives are designed to execute efficiently, then the performance of the implementation benefits. Another advantage is protocol portability. If the protocol implementation relies on the services of the framework and avoids the O/S interface to the greatest extent possible, porting protocol implementations is simplified. Optimally, once the framework is ported, protocol porting is free.

[1] Actually, *enforced* is far too strong a word. It would be trivial (however counterproductive) to avoid the libraries and circumvent the guidelines set out by any protocol framework.

[2] There is significant overlap between the two parts of the framework service in that the library routines help dictate protocol implementation conventions.

PPF has been used to implement several protocol sets. These include a connection-oriented, full ISO stack [GNI92], and a light-weight, connectionless request/response protocol [NG90].

1.2 Overview of Services Supported By PPF

The PPF framework contains several modules to guide the structuring and aid in the implementation of communication protocols. This section provides an overview of some of the services offered by these modules.

PPF defines a technique for protocol parallelization and provides a set of routines to support parallel protocol execution. These routines include mutual exclusion management for critical sections, ordering mechanisms for protocols which expect implicit event ordering, and sequence number generation routines to support the ordering mechanisms. These guidelines and routines allow the parallel execution of normally serial protocols with minimal impact on the protocol implementation.

PPF provides guidelines and routines for inter-layer communication. It defines the interfaces (called boundary routines) that must be provided by each layer so they may receive events from other layers. PPF provides a routine (PostEvent()) which enables one protocol entity to communicate with another protocol entity at any other layer. This standardized interface makes it possible to interchange protocol entities at a particular layer without having to alter the code in the neighbouring layers, provided that the exchanged protocols provide the same interface and expect identical arguments.

PPF loosely defines a format for both entity and connection control blocks. An entity control block (called an *entity variable area* or EVA) contains information particular to a single protocol entity, including the protocol state machine and connection control blocks. A connection control block defines the state of a single connection. This includes protocol state information, option information, and a record of the connection identifiers of the service user(s) and provider(s).

PPF provides a timer service. Timers can be started and stopped. Timer expiry notification is performed using RFPs inter-layer communication primitives.

PPF defines a buffer structure for protocol data units in the process of being encoded,

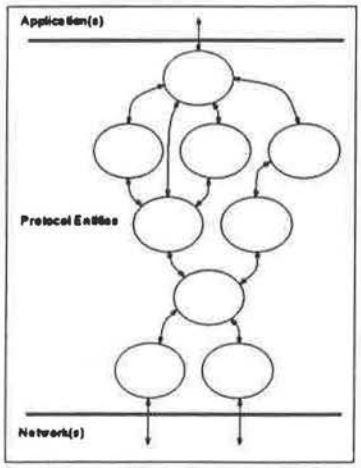

Figure 1: PPF Supports a Hierarchical Graph of Protocols

and for received data in the process of being decoded. These common data structures contribute to the uniformity of the inter-layer interface.

Part of the framework service is provided by the underlying user-level kernel called Pthreads. Pthreads is a user-level kernel for use on multiprocessors. It provides an environment suitable for running a group of parallel cooperating threads. Pthreads allows efficient process creation and destruction, interprocess communication and memory allocation in a preemptive parallel environment. Pthreads provides semaphores (as well as IPC) for application thread synchronization. It also provides a sleep facility and O(1) memory allocation and release routines. There are two levels of memory allocation. One is a general system similar to Malloc() and Free(). The other is designed for efficient allocation, transfer, and release of complex data structures.

1.2.1 What Protocols Can Use the PPF Framework?

PPF can be used to support a wide variety of protocols and protocol organizations. Both connection-oriented and connection less protocols are supported. Protocol organizations are not restricted to stacks. Instead, a hierarchical graph of protocols is supported where multiple protocol entities may exist at any layer. Figure 1 depicts this relationship.

1.2.2 What Is The Form of Communication Between Layers?

Inter-layer communication follows Clark's structuring of protocols using upcalls [Cla85]. Each upcall (or downcall) communicates a protocol event between two protocol entities. Upcalls reduce the total communication overhead between layers to that of a subroutine call, plus a small number of computations.

To receive events, PPF requires that each protocol implementation present a set of six boundary routines. Boundary routines are called when an event is to be delivered to the protocol. Each boundary routine expects a different set of events.

The format of events is also dictated by PPF. Each event is made of two components; a boundary type and an event type. Primitives are provided to compose and examine event components.

1.2.3 What Is The Process Structure?

Protocols using the PPF framework use a process-per-packet approach to process structuring. A single process is capable of ascending or descending through the layers carrying a single packet. This avoids expensive buffering and context switching between layers. Depending on the nature of the desired application interface (synchronous or asynchronous) a packet may never have to cross process boundaries.

1.2.4 What Is The Basis For Parallelism?

Protocols can be implemented to run in parallel using the PPF protocol framework. PPF uses processor-per-packet parallelism. Here, each packet is assigned its own processor to carry it through the protocol graph. Not all protocol processing activities, however, are simple to parallelize.

Protocol processing for connection-oriented protocols (and some connection-less protocols) can be roughly divided into two unequal parts: packet processing and connection control block (CCB) manipulation [GNI92]. Packet processing involves external data representation conversion functions, checksum calculations, encryption and decryption, and

encoding and decoding of packet headers. CCB manipulation includes state machine transitions, enqueuing received segments, etc. In some implementations these parts are intertwined, though our experience shows they are not difficult to isolate. The relative processing overhead of these two parts varies from protocol to protocol. For most protocols, CCB manipulation for a single packet involves only a few lines of code to enqueue a segment, update sequence numbers or initiate an acknowledgement (often not even these are required). Packet processing is normally more complex. Even for protocols with very simple protocol data unit (PDU) formats, such as the network and transport layers, packet manipulation is a significant proportion of the protocol code and execution overhead. At higher layers, such as the presentation and application, packet processing overhead completely dominates all other processing [CT90].

Fortunately, it is the packet processing stage of protocol processing that is easiest to parallelize. Any number of packets may proceed through the packet processing stage of a protocol (even within the same connection) in parallel without synchronization. Synchronization and ordering are only required for the comparably small CCB manipulation stage. Depending on the number of processors available, this arrangement has been shown to provide significant improvements in overall protocol performance [GNI92]. The PPF protocol framework provides mechanisms for incorporating such parallelism. Any protocol whose CCB manipulation and packet processing stages can be identified and isolated in the implementation are likely to benefit from this form of parallelism. Figure 2 shows a parallel protocol graph organization. This figure depicts a simple, linear (ISO-like) protocol stack organization. Note that CCB manipulation at each layer is protected by a mechanism (called the event gate) and that multiple processes proceed in parallel through each protocol layer (or entity).

2 PPF Details

2.1 Pthreads

Pthreads is a user-level kernel for multiprocessors. Its current platform is a shared memory, four-processor risc-based computer running UNIX. It should be a relatively simple port to other shared-memory multiprocessors running most any host operating system. Pthreads provides thread (process) management, synchronization, and communication functions. It

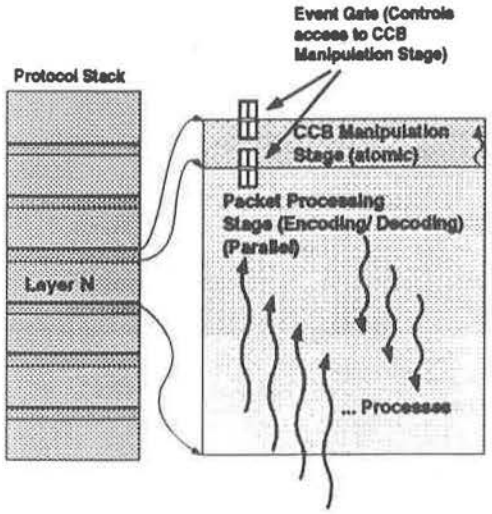

Figure 2: Protocol Organization and Parallel Execution

also provides memory allocation services.

Threads are scheduled using a three priority, round-robin, preemptive scheduling algorithm. No thread is allowed to run if there are any ready threads of higher priority. Pthreads ready queues are shared by all processors. Therefore a ready thread is run by the next available processor and a thread may migrate from processor to processor over its lifetime. Threads run in parallel to the extent of the available processors.

2.1.1 Thread Management

A Pthreads application begins execution at the first statement of the routine *mainp()*[3]. *Mainp()* is passed the environment arguments argc and argv. *Mainp()* is not a thread and therefore most kernel calls which would be appropriate when called by a thread are not appropriate when called from *mainp()*. The normal function of the *mainp()* routine is the creation of some initialization threads.

Threads are created using the following call:

```
PID Create(void(*addr)(), int stksize, char *name, void *arg,
```

[3] *mainp()* in Pthreads is analagous to *main()* for regular *C* applications.

```
PPRIO prio, int level)
```

On success, *Create()* returns the identifier of the newly created ready thread. Failure, for any reason, results in a return value of PNUL. The first parameter, *addr*, is a pointer to the routine which acts as the entry point for the created thread. The *stksize* parameter is the size, in bytes, of the new thread's stack. The stack requirements vary according to the number and size of local variables and parameters, and well as the depth of subroutine calls. A minimum requirement is generally about 3K bytes, though some threads require as much as 10K bytes or more. The third parameter, *name*, points to a text string which acts as a user supplied thread identifier. This string must be null terminated and currently has a length restriction of 32 bytes including the null-terminator. Any number of threads may be identified by the same name. The name is copied by the Create() routine and therefore the memory containing this name may be released by the caller on return from Create(). The fourth parameter, *arg*, is an argument (parameter) for the created thread. This argument is passed transparently to the thread and may be of any (4 byte or smaller) type, although it should be cast to a *(void *)* on call. The entry routine for the new thread receives this argument as a parameter, or may instead not declare any parameters if no creation-time arguments are required. The fifth argument (*prio*) indicates the priority of the created thread. The possible priorities are HIGH, NORM and LOW. The final argument indicates whether the created thread is a system thread (SYS), or an application (USR) thread. The only difference between the two is that Pthreads will exit when there are no more USR level threads. Therefore, perpetual server threads should be created with level SYS.

Threads leave the system in one of three ways. The first possibility is for the thread to "fall of the end" of (i.e. return from) the entry subroutine. The second possibility is for the thread to call the Pexit() routine which causes the thread to leave the system at that point. The final possibility is for a thread to be killed by some other thread using a call to Kill(). Pexit() requires no parameters and returns no results. Kill() requires the thread identifier as the argument and returns the same identifier on success or the value PNUL if the thread to be killed does not exist. The headers for Pexit() and Kill() are as follows:

```
PID Kill (PID pid)
void Pexit()
```

Threads can be suspended and resumed using calls to Suspend() and Resume(). A thread

can only suspend itself, and is suspended until resumed by some other thread. Suspend requires no parameters and returns no value. Resume requires the identifier of the thread to be resumed. The headers for *Suspend()* and *Resume()* are as follows:

```
void Suspend()
int Resume( PID pid )
```

Pthreads provides three routines to help in identifying threads. These are *MyPid()*, *NameToPid()*, and *PExists()*. MyPid() requires no parameters and returns the identifier of the calling thread. NameToPid() requires a null-terminated string as its single argument, and searches the system for any thread whose name (see Create() arguments) matches the given string. If such a thread is found, its identifier is returned, otherwise PNUL is returned. If more than one thread exists with the same name, the identifier returned is arbitrarily selected from the set of appropriate choices. PExists() takes a thread identifier as its only argument and returns true or false (1 or 0) depending on whether there exists a thread with the given identifier. The headers for these three routines are as follows:

```
PID MyPid()
PID NameToPid (char *name)
int PExists( PID pid )
```

2.1.2 Thread Synchronization

Pthreads provides four semaphore operations *NewSem()*, *FreeSem()*, *P()*, and *V()*. The headers for these routines are as follows:

```
int NewSem( int value, int duration )
int FreeSem( int semNo )
int P( int semNo )
int V( int semNo )
```

NewSem() requires two arguments, *value* and *duration*, and returns the identifier of the allocated semaphore (or -1 if no more semaphores are available). *Value* is the value to which the new semaphore is to be initialized. *Duration* is optimization information which indicates how long a thread would expect to be blocked on this semaphore. The possible

values are SHORT (e.g. for protection of a very small critical section) or LONG (e.g. to block a thread pending the availability of data). If in doubt, use LONG[4].

FreeSem() returns a previously allocated semaphore to the system. Its single argument *semNo* is the identifier of the semaphore to be returned. FreeSem() returns 0 on success and -1 on failure.

P() and V() implement the conventional semaphore primitives. Each require the single parameter *semNo* to identify the synchronization semaphore. Both routines return 0 on success or -1 on failure.

2.1.3 Interprocess Communication

Pthreads provides blocking Send()/Receive()/Reply() interprocess communication primitives [Che88]. A sending thread is blocked until the message has been received and a reply has been made. A receiving thread is blocked until some other thread sends it a message. Reply() is a non-blocking operation. There is also a non-blocking routine that allows a thread to test whether it has a message waiting for it to receive. The subroutine headers are as follows:

```
void *Send ( PID to, void *msg, int *len )
void *Receive( PID *pid, int *len )
int Reply(PID sndr, void *msg, int len)
int MsgWaits()
```

Send() has three parameters. The first one, *to* is the identifier of the intended recipient. The second parameter, *msg*, is a pointer to the message (any contiguous buffer). The final parameter, *len* is the length, in bytes, of the message. This parameter is a value/result parameter. On call, this parameter contains the length of the sent message. On return, the parameter contains the length of the replied message. Send() returns a pointer to the reply message on success, or the value NOSUCHPROC on failure.

Receive() requires two result parameters. On return, the first parameter will contain the identifier of the originator of the message. The second parameter (again, on return) contains the length of the sent message. Receive() returns a pointer to the sent message.

[4] In the current version of Pthreads, the second parameter, duration, exists but is not used.

Reply() requires two parameters. The first, *sndr* is the identifier of the reply destination (i.e. the original sender). The second parameter, *msg* is a pointer to the reply message. The final parameter is the length of the reply message. Reply() returns 0 on success or -1 on failure.

MsgWaits() requires no parameters and returns 1 if there are messages waiting to be received by the calling thread, or 0 if no such messages currently exist.

2.1.4 Memory Allocation

Pthreads provides two memory allocation systems. These are the *frame-based* and *general* systems.

The general system allocates memory using calls to Malloc(), Realloc(), and Free(). Memory allocated in this way is permanent is the sense that the memory can exist beyond the lifetime of the allocating thread. Such memory remains allocated until it is explicitly released using a call to Free(). Pthreads uses a very fast (O(1)) memory allocation scheme. The headers for Malloc() and Free() are as follows:

```
void *Malloc(int size)
void *Realloc( void *ptr, int size )
void Free( void *mem )
```

Malloc() requires as its single parameter the size, in bytes, of the requested contiguous memory segment. On success, Malloc() returns the address of the allocated segment. On failure, NULL is returned.

Realloc() expands or contracts a previously allocated memory segment by freeing the initial segment and allocating a new one. It requires two parameters. The first is a pointer to the existing memory segment. The second is the size of the desired segment. Realloc() copies the contents of the existing segment to the new segment to the extent of the smaller of the two. Realloc() returns a pointer to the new segment.

Free() returns a previously allocated segment to the system. The single parameter to Free() is the address of the first byte of the segment to be returned. Free() returns no values.

The frame-based memory allocation system is designed for the management of complex data structures. These data structures are often composed of many individual segments. This memory facility allows these individual segments to be managed and freed as a group. This facility also avoids the potential internal fragmentation and overhead problems associated with allocating a large number of very small (1+ byte) segments.

Each thread has its own stack of *memory frames*. Each memory frame groups a set of associated memory segments. A frame can be transferred from one thread to another, or freed as a unit. Typically, all the segments that comprise a single data structure would be allocated on one frame. This entire data structure could then be transferred to another thread or freed with single library calls. When a thread leaves the system, all the memory on its frame stack is returned to the system. The headers of the subroutines which operate on temporary memory are as follows:

```
void NewFrame()
void FreeFrame()
void SwapFrame()
void *PopFrame()
void PushFrame( void *frame )
int TransferFMem( PID topid )
void *FMalloc( int size )
void *MallocFromFrame( void* frame, int size )
void FreeFMem()
```

NewFrame() requires no parameters and returns no result. It creates a new memory frame and pushes it onto the calling thread's memory frame stack.

FreeFrame() also requires no parameters and returns no result. It frees all memory associated with the calling thread's top memory frame, pops the frame and discards it.

SwapFrame() swaps the top two memory frames of the calling thread. If zero or one frames exist for this thread then SwapFrame() has no effect.

PopFrame() pops and returns a pointer to the top memory frame of the calling thread. This routine should be used with caution, generally in conjunction with PushFrame(). The reason for caution is that a memory frame which is not currently on any thread's memory stack is essentially an orphan. This memory will not be returned to the system should its creator or owner exit.

PushFrame() takes a pointer to a frame and pushes it on the calling thread's memory stack. The PopFrame() / PushFrame() pair can be used to transfer per-thread (frame-based) memory from one thread to another, or to perform stack rearrangement functions. Note that a memory frame cannot exist on more than one memory stack at a time. If an application wishes to move the top memory frame from one thread to another, this may be done using PopFrame(), PushFrame() and inter-process communication, but it is preferable to use TransferTempMem() instead.

TransferFMem() requires the single argument *topid*. This operation transfers the top memory frame of the calling thread to the top of the memory stack of the thread identified by *topid*. This routine avoids the time interval between a PopFrame() and a PushFrame() when a memory frame does not belong to any thread. TransferTempMem() returns 0 on success, or -1 on failure(eg. if an invalid argument is encountered).

FMalloc() takes an integer parameter *size*. This routine allocates memory from the top memory frame of the calling thread. This memory cannot be released using Free(). Frame-based memory is returned to the system using FreeFrame() (discussed previously) or FreeFMem(). An important feature of FMalloc() is that if no memory frame currently exists on the calling thread's memory stack, a new one is created and pushed automatically. In this case, the allocated memory is taken from the new frame.

MallocFromFrame() perfroms the same task as FMalloc(), except in this case the frame from which to allocate ther memory is specified. MallocFromFrame requires two parameters. The first is a pointer to the frame from which to allocate memory. This pointer is obtained by poping the top memory frame using a call to PopFrame(). The second parameter is the size of the requred buffer. This routine can be useful when multiple data structures are being allocated all at once. A small example follows where the application is constructing three data structures, DS1, DS2, and DS3:

```
void *frame1, *frame2, *frame3;    /* frame pointers              */
struct whatever *DS1;
struct whoknows *DS2;
struct whocares *DS3;

/* allocate and pop three new frames, one for each data structure */
NewFrame();
frame1 = PopFrame();
```

```
NewFrame();
frame2 = PopFrame();
NewFrame();
frame3 = PopFrame();

/* build the data structures */
DS1 = (struct whatever *) MallocFromFrame( frame1, sizeof(*DS1) );
DS1->field1 = MallocFromFrame( frame1, 20 );
...

DS2 = (struct whoknows *) MallocFromFrame( frame2, sizeof(*DS2) );
DS2->field1 = MallocFromFrame( frame2, 10 );
...

DS3 = (struct whocares *) MallocFromFrame( frame3, sizeof(*DS3) );
DS3->field1 = MallocFromFrame( frame3, 15 );
...
```

This code segments produces three composite data structures. The memory for each is allocated on its own frame and is therefore disjoint. Each data structure can be passed to another thread as a group, or freed using calls to PushFrame() and FreeFrame();

FreeFMem() returns no values and requires no parameters. It releases the calling thread's frame-based memory (from all frames) to the system. This routine also pops all memory frames from the calling thread leaving it with none.

2.1.5 Sleep

Pthreads allows threads to put themselves to sleep. It uses a hierarchical data structure for sleeping threads which provides O(1) timer manipulation [VL87]. Two routines are provided for this service: *Sleep()* and *ReSleep()*. The headers for these routines are as follows:

```
int Sleep( int ticks )
int ReSleep( PID sleeper, int ticks )
```

Sleep() suspends the execution of the calling thread for a duration of *ticks* (the only parameter) clock ticks[5]. The upper limit of *ticks* is 0xffff. Any thread wanting to sleep

[5] A clock tick defines a time interval. The interval duration is a compile time configuration constant.

longer than this duration will have to make multiple calls to Sleep(). Sleep() returns 0 on success and -1 on failure (eg. if ticks is out of range).

ReSleep() adjusts the wakeup time of a currently sleeping thread. This is useful for timer implementations where the sleeping thread is to be kept sleeping longer, or is to be awoken at once. ReSleep() requires two parameters: *sleeper* and *ticks*. Sleeper is the identifier of the sleeping thread whose waketime is to be altered. Ticks is the new sleep duration relative to the time of the ReSleep() call. A ticks value of 0 has the effect of waking the thread immediately. ReSleep() returns 0 on success and -1 on failure (eg. if the identified thread is not found to be sleeping, or if ticks is out of range).

2.1.6 Elapsed Timers

Pthreads provides a timer service to time the interval between events. These timers (called elapsed timers) are normally used for performance evaluation and are manipulated using four routines whose headers are as follows:

```
int ResetETimer(int timer)
int StartETimer(int timer)
int GetETimer(int timer)
int StopETimer(int timer)
```

Each of these routines requires the timer identifier as the single argument. Timers are numbered starting at zero. The number of timers is a configuration constant (currently set at 32).

ResetETimer() stops the timer (if it was running), and resets the elapsed time to zero. This is to be called before the timer is used. ResetETimer() returns 0 on success, or -1 if the timer number is out of range.

StartETimer() starts the timer running. If the timer is already running when StartETimer() is called, the elapsed time is reset to begin counting again from zero. StartETimer() returns 0 on success, or -1 if the timer number is out of range.

GetETimer() returns the amount of time that has elapsed (in terms of system clock ticks - see Sleep()) over the most recently timed interval. If the timer is running, the value

returned is the number of clock ticks that have occurred during the interval between the most recent call to StartETimer() and this call to GetETimer(). If the timer is not currently running, the value returned is the number of clock ticks that have occurred during the interval between the most recent call to StartETimer() and the most recent (though later) call to StopETimer(). The call to GetETimer() does not change the state of the timer (i.e. noes not alter the time value, start the timer, or stop the timer). If the timer argument is out of range, a -1 is returned.

StopETimer() stops the timer and returns the number of clock ticks during the interval between this call and the most recent call to StartETimer(). This value may be retrieved again using subsequent calls to GetETimer(). If the timer argument is out of range, or the timer is not currently running, a -1 is returned.

A thread may also obtain the current value of the system clock using *Time()*. Time returns the number of clock ticks since system startup. The header is as follows:

```
long Time()
```

2.2 Internal Protocol Details

PPF dictates implementation detail conventions that simplify protocol implementation and readability. This section describes those conventions that apply within a single protocol layer.

2.2.1 Entity Identifier and Entity Variable Area

Each protocol entity has an entity identifier and an entity variable area (EVA) associated with it. The entity identifier is an integer which distinguishes this entity from all others in the system. The identifiers are used by PPF as array indices and are therefore chosen sequentially starting from zero. Examples of identifiers appropriate for an ISO stack are as follows:

```
/* list of entities */
#define PHYSICAL     (ENTITY) 1    /* physical entity          */
#define DATALINK     (ENTITY) 2    /* data link entity         */
```

```
#define NETWORK      (ENTITY) 3    /* network entity              */
#define TRANSPORT    (ENTITY) 4    /* transport entity            */
#define SESSION      (ENTITY) 5    /* session   entity            */
#define PRESENTATION (ENTITY) 6    /* presentation entity         */
#define ACSE         (ENTITY) 7    /* association control entity  */
#define ROSE         (ENTITY) 8    /* remote operations entity    */
#define USER         (ENTITY) 9    /* protocol user interface     */
```

The *EVA* is an implementor-defined data structure which maintains the protocol entities state. For most protocols, this state includes references to protocol state tables and connection control blocks. Other, protocol-specific, information can also be held in the *EVA*. An example of an entity variable area for a session entity is as follows:

```
typedef struct
    {
    int newcidSem;/*semaphore for allocating connection identifiers*/
    int numbccbs; /*total number of connection control blocks      */
    SessCCB *ccbs;/*pointer to connection control blocks           */
    int (*(statetab[NUMSTATES]))(); /* state table addresses      */
    } SessEVA;
```

This (and most) *EVAs* are quite simple, containing only information pertaining to the protocol entity as a whole. Figure 3 shows the organization of an entity variable area.

There are several advantages to grouping protocol information into this data structure. First, all "global information" regarding a protocol entity can be found in a single location rather than being dispersed throughout the code. Secondly, and more importantly, it is possible that more than one protocol entity can make use of the same protocol implementation, so long as each has its own *EVA*. For example, it would be possible to have a choice of two session entities (perhaps with different state tables) within a protocol implementation. Both entities would share a common *EVA* format (with different contents) and could make use of the same supporting code.

2.2.2 Connection Control Blocks

Connection Control Blocks (*CCBs*) tend to be much more protocol-specific than *EVAs*. Each *CCB* maintains state information for a single connection. Examples of information

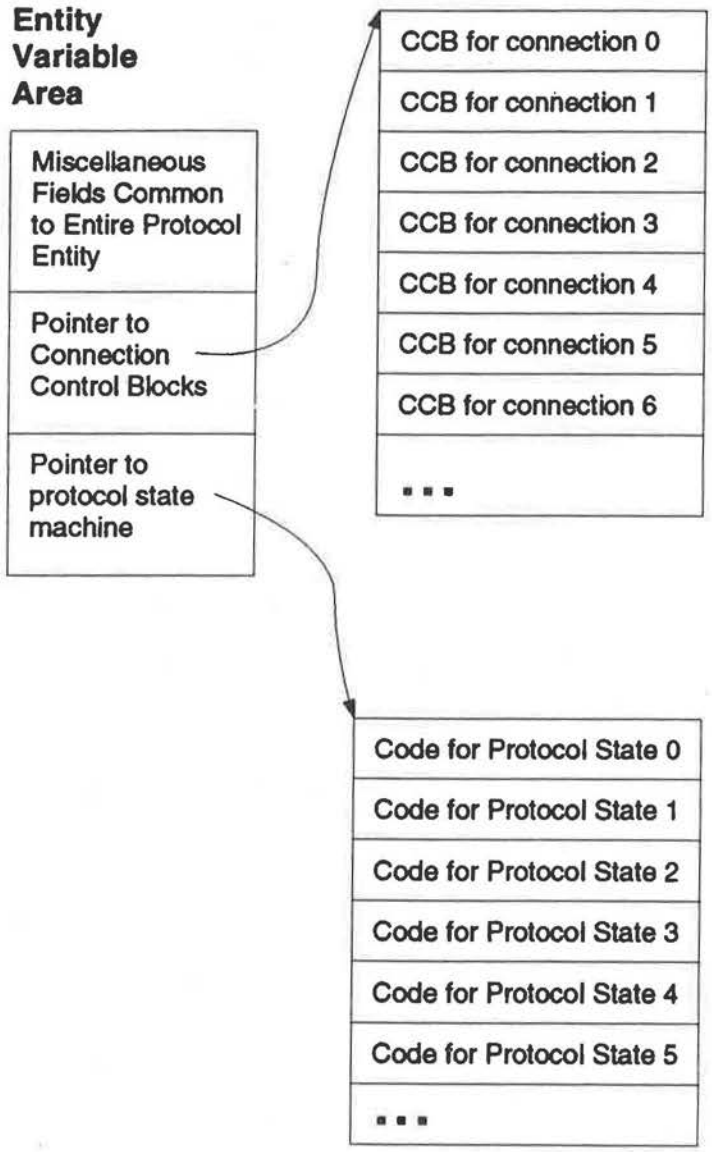

Figure 3: The Organization of an Entity Variable Area

contained in a CCB includes connection state, maximum data transfer size, and connection identifiers (and perhaps entities) of the associated service user and provider. Connection control blocks may be allocated statically (using an array) or dynamically as needed. PPF provides a skeleton subroutine which may be used to allocate CCBs dynamically, but any reasonable allocation algorithm will suffice. An example of a TRANSPORT CCB [CCI89b] follows (some fields have been removed for brevity):

```
typedef struct      /* transport connection control block          */
    {
    BYTE  state;    /* current state for this connection           */
    int   tpdusize; /* agreed upon maximum tpdu size for this conn */
    int   destref;  /* peer connection number                      */
    int   upcid;    /* conn id of service user                     */
    int   upentity; /* entity id of service user                   */
    int   cid;      /* connection identifier for this connection   */
    int   dncid;    /* connection id of service provider           */
    int   dnentity; /* entity id of service provider               */
    DATAHEADER *rdata;  /* received data list (pending reassembly) */
    } TransCCB;
```

2.2.3 Protocol State Tables

Protocol state tables dictate protocol actions on the basis of received events. PPF state tables are normally implemented as subroutines, one subroutine per protocol state. All state subroutines have a common set of parameters. The header for each state is as follows:

```
stateName(CCBType *ccb, int cid, EVENT event, void *evpa, SEQNO seqNo)
```

The state subroutine is passed a reference to the appropriate connection control block, the connection identifier for this connection, the event to act on, event-dependant parameters, and an internal event sequence number (discussed in 2.5). Each state subroutine normally consists of a switch statement with each case corresponding to a possible event type. The following is an example of the data transfer state of a TRANSPORT entity (again, edited for brevity):

```
topen(TransCCB *tempccb,int cid,EVENT event,void *evpa,SEQNO seqNo)
{
    switch ( GETEVENTP( event ) )
        {
        case TDataReq:  /* service user has data to send          */
            {
            create a data PDU and call PostEvent() to pass to the
            NETWORK entity ...
            }
            break;

        case DT:    /* data received from service provider        */
            {
            if data is part of a segmented PDU, add to other
            segments. If SDU is complete, use PostEvent to pass it
            to SESSION entity ...
            }
            break;

        case TDiscReq:  /* SESSION would like connection closed   */
            {
            close this transport connection and ask NETWORK entity
            to close corresponding network connection (by passing
            it a NDiscReq event using PostEvent()) ...
            }
            break;

        case NDiscInd: /* NETWORK would like connection closed    */
            {
            close this transport connection and ask the SESSION
            entity to close its corresponding connection (by passing
            it a TDiscInd event using PostEvent()) ...
            }
            break;

        case DR:    /* peer TRANSPORT would like connection closed*/
            {
            close this connection and ask the SESSION and NETWORK
            entities to close their associated connection by issuing
            NDiscReq and TDiscInd events using PostEvent() ...
            }
            break;
```

```
        case ER:        /* peer TRANSPORT is reporting an error   */
            {
            close this connection and ask the SESSION and NETWORK
            entities to close their associated connection by issuing
            NDiscReq and TDiscInd events using PostEvent() ...
            }
            break;

        default: /* unexpected event - abort connection           */
            {
            close this connection and ask the SESSION and NETWORK
            entities to close their associated connection by issuing
            NDiscReq and TDiscInd events using PostEvent() ...
            }
            break;
        }
}
```

State subroutine entry points are, by convention, stored in a array in the entity variable area. Each state has an integer associated with it which serves as an index into this array. This allows simple invocation of state subroutines and allows simple state transitions. As an example, consider a TRANSPORT entity with five states. Each state would be assigned an integer as follows:

```
/* transport protocol states */
#define CLOSED      0
#define OPEN        1
#define WFCC        2
#define WFNC        3
#define WFTRESP     4
```

Each of these correspond to state subroutines with the following headers:

```
tclosed(TransCCB *tempccb, int cid, EVENT event, void *evpa, SEQNO seqNo)
topen(TransCCB *tempccb, int cid, EVENT event, void *evpa, SEQNO seqNo)
twfcc(TransCCB *tempccb, int cid, EVENT event, void *evpa, SEQNO seqNo)
```

```
twfnc(TransCCB *tempccb, int cid, EVENT event, void *evpa, SEQNO seqNo)
twftresp(TransCCB *tempccb, int cid, EVENT event, void *evpa, SEQNO seqNo)
```

The entry points to these subroutines are maintained in an array (which forms part of the entity variable area) as shown by the following code segment.

```
/* load the state table with addresses of the state routines */
teva->statetab[CLOSED]      = (int (*)()) tclosed;
teva->statetab[OPEN]        = (int (*)()) topen;
teva->statetab[WFCC]        = (int (*)()) twfcc;
teva->statetab[WFNC]        = (int (*)()) twfnc;
teva->statetab[WFTRESP]     = (int (*)()) twftresp;
```

Now, the simplicity of state transitions is illustrated by the following example:

```
tempccb->state = CLOSED;
```

States can be executed (by the boundary routines) as follows:

```
(*(EntVarArea->statetab[tempccb->state]))(tempccb, *cid,
                          event, evpa, seqNo);
```

This has proven to be an efficient, elegant, and easy to read arrangement for the maintenance and execution of states and state tables.

2.2.4 Connectionless Protocols

Most of the discussion up to this point assumes connection-oriented protocols. PPF, however, also supports connection-less protocols. Connection-less protocols come in several varieties. Truely connection-less protocols maintain no per-connection state and require little data structure (i.e. CCB) support. Other protocols implicitly establish connections, without the exchange of specific connection management PDUs. The duration of these implicit connections range from a single PDU transfer in each direction, to arbitrarily long periods.

Most protocols, regardless of the nature of their connections (or lack thereof), can make use of PPF protocol support. Support in the form of entity variable areas, state tables, buffer management, inter-layer communication, user-level kernel facilities, protocol event management, timer facilities, and parallelism constructs are all useful.

2.2.5 Summary

Each protocol entity is defined by its entity variable area (EVA). Each entity is identified by a unique integer called the entity identifier. The EVA contains entity-specific information such as configuration information, connection control blocks (one per open connection), and a reference to the protocol state machine.

2.3 Inter-Layer Communication

PPF protocol implementations structure inter-layer communication using upcalls. The effect is a non-buffered, procedure call interface between layers. Data in upcalls (and downcalls) is in the form of protocol *events*. This section describes events, and explores how events are created, examined and passed between layers.

2.3.1 Constructing and Examining Events

An event is a unit of information passed to a protocol entity, usually from another protocol entity. Events normally correspond to protocol events as described in protocol specifications. Examples of events generated and received by an ISO Class 0 TRANSPORT entity include *TDTreq*, *TCONresp*, *NDISreq*, and *TDISind*. PPF provides mechanisms for generating, sending, and receiving events.

PPF events are composed of two parts, the event boundary and the event identifier. Event boundaries consist of the following: *CNTLUP*, *CNTLDN*, *DATAUP*, *DATADN*, and *TIMEOUT*. Any other event boundary is said to be of boundary *otherwise*. These bondaries are discussed in section 2.3.2.

Events are composed and examined using the following PPF routines:

- MAKEEVENT(boundary, id)
- GETBOUNDRY(event) /* get event boundary */
- GETEVENTP(event) /* get event identifier */

MAKEEVENT() takes an event boundary (one of those listed above) and an event identifier (protocol defined) and creates a PPF event. The event id must be in the range 0 to 255 decimal. An PPF event of type EVENT is composed and returned.

The boundary of a PPF event can be examined using GETBOUNDRY(event). The argument is of type EVENT. This routine returns the event boundary of the PPF event.

The event identifier of a PPF event can be examined using GETEVENTP(event). The argument is of type EVENT. This routine returns the event identifier of the PPF event.

2.3.2 Protocol Boundaries

Protocol layers written for the PPF environment are structured to isolate and divide the protocol interface into six parts (or some subset of these parts). Each of these parts is called a protocol boundary. Each boundary is responsible for receiving and acting on a certain subset of events. The six boundaries are as follows:

- *Data Up*
- *Data Down*
- *Ctrl Up*
- *Ctrl Down*
- *Timeout*
- *Otherwise*

The *Data Up* boundary expects data events from the service provider. The only appropriate event for this interface is received data destined for this protocol entity. As an example, a PRESENTATION [CCI87] entity would expect to receive either a *SDTind*, (data) a *SEXind* (expedited data), or a *STDind* (typed data) at this boundary.

The *Data Down* boundary expects data events from the service user. The only appropriate event at this boundary is a PDU for encapsulation and/or transmission. For example, a SESSION [CCI89a] entity would expect to receive a *SDTreq*, a *SEXreq*, or a *STDreq* at this boundary.

The *Ctrl Up* boundary expects non-data, non-timer events from the service provider. Examples of such events received by a SESSION entity might be *TDiscInd*, *TConnConf*, or *TConnInd*.

The *Ctrl Down* boundary expects non-data, non-timer events from the service user. For example, a TRANSPORT entity would expect to receive events such as *TConnReq*, *TConnResp*, or *TDiscReq* through this boundary.

The *Timeout* boundary receives notification of timer expiries. Events arriving at this boundary consist of a number identifying the expired timer.

Finally, the *Otherwise* boundary routine exists to receive events which do not conform to any of the above categories. The most obvious example is an event to signal an error condition.

2.3.3 Boundary Routine Definition

Each boundary routine for each protocol consists of a subroutine conforming to the following header:

```
Boundary_Name(void *EntVarArea, int *cid, EVENT event,
              void *evpa, SEQNO seqNo)
```

When a boundary routine is invoked it is passed a reference to the entity variable area, a reference to the connection identifier, the event, the event parameter area, and the internal sequence number for the event. Each of these are discussed elsewhere in this paper. The boundary routine performs some initial processing on the event and then passes it to the protocol state machine. This initial processing might involve decoding a received data PDU, or allocating a new connection control block as a result of a connection request. The following is an example of a TRANSPORT entity data-up boundary routine.

```
/* note: parts have been left out for brevity */
TdataUp(TransEVA *vararea,int *cid,EVENT event,void *evpa,SEQNO seqNo)
{
    TPDUINFO   tpdu;
    TransCCB *tempccb;

    /* decode the PDU received into a structure of type TPDUINFO  */
    decTPDU(  &tpdu, (DATAHEADER *) evpa);

    /* find the connection control block for this connection      */
    tempccb = &(vararea->ccbs[*cid]);

    /* now that the received PDU is decoded, alter the event type */
    /* to reflect the type of the decoded PDU.                    */
    event = MAKEEVENT( GETBOUNDRY( event ), tpdu.pdutype );

    /* call the appropriate state machine                         */
    (*(vararea->statetab[tempccb->state]))(tempccb,*cid,event,&tpdu,seqNo);
}
```

Notice that the state machine is passed the actual connection identifier rather than a reference to it. Also, because the state machine operation concerns only this connection (and not the entity as a whole) it is not passed a reference to the entity variable area.

2.3.4 Passing Events

Events are passed from one entity to another using the *PostEvent()* PPF library routine. Events passed using this routine are delivered directly to the appropriate boundary routine of the destination protocol entity (boundary routines are discussed in section 2.3.2). *PostEvent()* provides a subroutine call-based form of communication (upcalls) between protocol entities. There is no context switching or queueing performed as a result of posting an event. The overhead imposed by one call to PostEvent() consists of two array indirections (the call to PostEvent() itself is inline). The header for PostEvent() is as follows:

```
int PostEvent(ENTITY entity, int *conid, EVENT evID, void *epa, int sn)
```

The first two parameters (*entity* and *conid*) identify the event's destination entity and connection (section 2.4 discusses entity identifiers). By convention, the connection identifier (cid) is that of the destination entity rather then the source entity. This implies that at some point each connection must exchange cids with associated provider and user connections. This is normally accomplished at connection establishment time. Here, for the call to PostEvent(), the connection initiator will place the value of its own cid in the *conid* field. The recipient of the event will record this value for future use, and before returning replace the value with its own cid. This value is recorded by the connection initiator on return. It is for this reason that *conid* is a reference parameter rather than a value parameter. This exchange is illustrated by the following two code segments.

```
/* SESSION entity asking the TRANSPORT entity for a new connection */
int    dncid;

dncid = cid;              /* cid is SESSIONs connection identifier */
PostEvent(TRANSPORT, &dncid, MAKEEVENT(CNTLDN,RESERVECID), 0, 0);
tempccb->dncid = dncid;     /* TRANSPORT responded with new cid   */
```

```
/* this is the TRANSPORT control-down boundary routine */
TctrlDn(TransEVA *vararea,int *cid,EVENT event,void *evpa,SEQNO seqNo)
{
    TransCCB *tempccb;

    /* special case as there is no assigned connection yet */
    /* Service user would like a connection established... */
    if( GETEVENTP( event ) == RESERVECID ) /* reserve a connection id   */
        {
        int ncid;
        ncid = newtcid( vararea );          /* allocate a new connection */
        vararea->ccbs[ncid].cid   = ncid;  /* record allocated cid      */
        vararea->ccbs[ncid].upcid = *cid;  /* record svc. users cid     */
        *cid = ncid;                       /* pass svc. user TPORT. cid */
        }

    ...

}
```

In the first code segment the SESSION entity posts the event using its own connection identifier. When the call to PostEvent() returns the cid given by the TRANSPORT entity is saved in the *dncid* field of the connection control block. In the second code segment the TRANSPORT entity has received the event asking for a new connection. A new connection is created (with a call to *newtcid()*), the SESSION's connection identifier is saved in the connection control block, and the newly created identifier is returned to the SESSION entity. All subsequent downward events posted by this SESSION connection will use the connection identifier provided by the TRANSPORT entity..

The third parameter to PostEvent() is the event being transferred. This event is composed of a boundary and event type as discussed in 2.3.1.

The fourth parameter is the event parameter area. This is a pointer to any parameters associated with this event. The contents of the event parameter area is dependant on the event type. There are, however, conventions regarding the parameter area for certain event types. These are discussed in 2.3.6.

The final parameter is the event's internal sequence number. This value is passed transparently to the destination entity and is described in 2.5.

2.3.5 State Consistency

Protocol implementors must keep state consistency issues in mind when posting events. When an event is posted, the thread (process) carrying the event passes from the source entity to the destination entity. Because PostEvent() is essentially a subroutine call, the thread will eventually return from the destination entity (i.e. from the call to PostEvent()). Protocol implementors should be cautious, however, in regard to protocol processing done after return from PostEvent(). In fact, it is usually advisable that all protocol processing be complete before the call to PostEvent(). The state of the entity must be consistent (i.e. state transitions must be complete, lists must be consistent, etc.) before the call to PostEvent().

The reason for ensuring state consistency is that it is possible that an upcall from some layer N to some other layer $N+1$ will produce an immediate downcall back to layer N. The most obvious example of this is when layer $N+1$ refuses a connection presented by layer

N. Any code occuring after the call to PostEvent() in layer N will not have been executed at the time that the downcall occurs. Layer N must be in a consistent state at the time of PostEvent() in this case as it receives an event before its call to PostEvent() returns. Thus, state consistency must be kept in mind when placing calls to PostEvent().

2.3.6 Buffer Management

PPF provides simple buffer management for protocol data. This includes data structures for protocol data units (PDUs) being encoded for transmission, and for PDUs being decoded after reception. Buffers are allocated and freed using threads memory allocations routines as discussed in section 2.1.4.

Received data is managed using the *DATAHEADER* structure. Some of the fields of this structure are shown below:

```
typedef struct dh  /* this is the header that points to upgoing pdu's */
    {
    int       datalen;
    BYTE      *data; /* pointer to data of concern for next layer*/
    BYTE      *mem;  /* points to beginning of mem allocated     */
                     /* for the data (for freeing)               */
    struct dh *next; /* pointer for linked received frames       */
    } DATAHEADER;
```

This structure may be augmented with other fields as required by the protocol set. As data is passed from lower to upper layers, each layer "consumes" its PDU header by advancing the *data* pointer and reducing the *datalen* field. The structure is then passed to the service user. The *mem* pointer remains unchanged and points to the beginning of the received data buffer. This field is used in the release of the buffer. These structures may be linked for reassembly of segmented PDUs.

Data (PDUs) for transmission are passed downward using the *PDUNODE* structure. Some of the fields of this structure are shown below:

```
typedef struct pdunode   /* the structure of an outgoing pdu -     */
    {                    /* each PDU is a list of these nodes      */
```

```
int     datalen;        /* length of pdu pointed to by data        */
BYTE    *data;          /* portion of pdu for this layer           */
struct pdunode *next;/* pointer to next node inline                */
} PDUNODE;
```

Normally, each protocol entity creates its header (or data) and prepends it to a list of PDUNODE structures. The list is passed to the service provider where the process is repeated until the constructed PDU is transmitted by the lowest layer. Being able to prepend variable size buffures greatly simplifies the task of PDU construction. Normally, the linked PDUNODEs are copied into a contiguous buffer at the lowest layer for transmission. This is not necessary, however, if the O/S or device interface supports gather-write.

2.3.7 Multiplexing

PPF's support for a protocol graph (rather than a simple stack) allows multiplexing. More than one entity may exist at each layer. In this case, one protocol entity can make use of more than one service provider, or may serve more than one service user. This raises several issues.

First, each connection must maintain the entity and connection identifiers of all associated service users and service providers. These values are required when posting an event to an associated entity.

Second, there must be a mechanism for determining the destination entity for connection management events. For example, if a connection request arrives at an entity which has three service users, the choice of which service user to pass the event to must be made. A similar situation occurs for downward connection establishment requests. The details of how this choice is made are internal to the protocol. For example, the choice may be made on the basis of quality-of-service requirements or (for ISO protocols), according to Service Access Points.

Finally, for parallel implementations, each source of events destined for the same protocol entity must use a seperate internal sequence number stream. This is discussed in section 2.5.3.

2.3.8 Timer Events

PPF provides a set of interval timers for protocol execution. A separate process (*timerProc*) is dedicated to timer management and the delivery of time-out events. Timers are manipulated by applications using the following four subroutines:

```
int  NewTimer()
void FreeTimer (int tnumb)
int  StartTimer (ENTITY entity, int tnumb, int seconds, int conn)
int  StopTimer (int tnumb)
```

NewTimer() allocates an interval timer to the caller. An integer timer identifier is returned on success. If no more timers are available, a negative value is returned. The number of timers is a PPF configuration constant.

FreeTimer() returns a previously allocated interval timer to PPF. *Tnumb* (the only parameter) is the timer identifier originally returned by *NewTimer()*.

StartTimer() begins the timing of an interval. The first parameter is the identifier of the entity to which the time-out event is to be directed. The second parameter is the timer identifier as returned by *NewTimer()*. The third parameter is the interval time in seconds. The fourth parameter is the identifier of the connection to which the time-out event is to be directed. Time-out events are delivered using PostEvent to the time-out boundary. In this case the event boundary is TIMEOUT, and the event type consists of the expired timer's identifier.

StopTimer() cancels a timing in progress. It requires a single parameter identifying the timer to be stopped. If the parameter identifies a valid timer, a zero is returned, otherwise a negative value is returned.

An example of a time-out boundary routine is as follows:

```
Stimer(vararea, cid, event, evpa, seqNo)
SessEVA *vararea;
int     *cid;
EVENT   event;
void    *evpa;
```

```
SEQNO   seqNo; /* will be NOSEQNO for timer indications */
{
    SessCCB *tempccb;

    /* find the connection control block                */
    tempccb = &( vararea->ccbs[*cid] );

    /* enter CCB manipulation space (described later)   */
    EnterEventGate( tempccb->eventGate, seqNo );

    /* massage event to be useful for state machine     */
    event = MAKEEVENT( 0, STIMER );

    /* call the state machine                           */
    (*(vararea->statetab[tempccb->state]))(tempccb, *cid, event,
          evpa, seqNo);
}
```

2.3.9 Summary

Communication between protocol entities takes the form of upcalls (essentially a subroutine call without buffering). The unit of communication is the protocol event passed from one entity to another. An event is composed of a boundary and a type. An entity receives events using its boundary routines. Each entity has one boundary routine for each event boundary type. The mechanism used to deliver an event to an entity is that of a procedure call to the appropriate boundary. The boundary routine normally performs some initial event processing and then calls the appropriate state machine subroutine. The boundary routine is not called directly by the initiator of the event. Instead, the initiator calls PostEvent() which, in turn, calls the appropriate boundary routine of the appropriate entity.

2.4 Protocol Initialization

Every protocol entity executes some initialization code on startup, before connections may be established or data transferred. This section outlines the initialization steps required by PPF, and discusses how the initialization code can be executed.

Initialization tasks fall into two categories. The first is protocol-dependant initialization

and the second is PPF-based initialization. Protocol-dependant initialization varies according to the protocol, though some activities are common to many protocols. For example, all connection-oriented protocols must create and initialize a data structure to maintain connection control blocks (CCBs). Another example of protocol-dependant initialization is the creation of typical PDU headers to reduce PDU construction overhead during data transfer.

PPF-based initialization consists of two tasks. The first task is the creation and initialization of the entity variable area (EVA). The purpose and format of an EVA is described in section 2.2.1. Initialization consists of allocating memory for the EVA and initializing its fields (eg. state table pointer array and connection control blocks).

The second task is the registration of the entities boundary routines. Boundary routines are registered with PPF by each protocol entity using the following subroutine:

```
SetECBNode(entity, cu, cd, du, dd, to, ot, eva)
ENTITY entity;
int  (*cu)(), (*cd)(), (*du)(), (*dd)(), (*to)(), (*ot)();
void *eva;
```

The first parameter, *entity* is the entity identifier of the calling entity. The next six parameters are pointers to boundary routine entry points. They are (in order) *control-up*, *control-down*, *data-up*, *data-down*, *time-out*, and *otherwise*. Each of these boundaries are discussed in 2.3.2. The final parameter (*eva*) is a pointer to the entity variable area of the calling entity. Once the boundary routines and entity variable areas of each entity are registered, the framework is able to deliver events to the appropriate boundaries.

Both types of protocol initialization (protocol-dependant and PPF-based) should be invoked as a result of a single subroutine. A call to this subroutine is then placed in the framework routine *InitProtos()*. InitProtos() is called once on startup by the application using the protocol service. An example of a TRANSPORT entity initialization routine is as follows:

```
inittransport()
{
    /* create the TRANSPORT entity variable area                    */
```

```
    (*teva) = (TransEVA *) Malloc( sizeof(TransEVA) );

    /* create the initial set of connection control blocks       */
    (*teva)->numbccbs = NUMBINITCCBS;
    (*teva)->ccbs     = (TransCCB *)
            Malloc( (*teva)->numbccbs * sizeof( TransCCB ) );

    /* initialize all connections to closed                      */
    initTccbs( 0, (*teva)->numbccbs - 1 );

    /* load the state table with addresses of the state routines */
    (*teva)->statetab[OPEN]        = (int (*)()) topen;
    (*teva)->statetab[CLOSED]      = (int (*)()) tclosed;
    (*teva)->statetab[RESERVEDCID] = (int (*)()) tclosed;
    (*teva)->statetab[WFCC]        = (int (*)()) twfcc;
    (*teva)->statetab[WFNC]        = (int (*)()) twfnc;
    (*teva)->statetab[WFTRESP]     = (int (*)()) twftresp;
    (*teva)->statetab[RESERVED]    = (int (*)()) treserved;

    /* register boundary routine addresses                       */
    /* note: that this entity has no time-out events and therefore */
    /* registers the "otherwise" routine as the time-out boundary. */
    SetECBNode(TRANSPORT, TctrlUp, TctrlDn, TdataUp, TdataDn,
               Totherwise, Totherwise, (*teva));

    /* new semaphore to ensure mutual exclusion of ccb creation  */
    (*teva)->newcidSem = NewSem( 1, SHORT );
}

initTccbs( first, last )
int     first; /* start initializing at this ccb */
int     last;  /* last ccb to initialize         */
{
    int count;

    for( count = first; count <= last; count++ )
        {
        (*teva)->ccbs[count].cid       = count; /* connection id */
        (*teva)->ccbs[count].state     = CLOSED;/* proto state   */
        (*teva)->ccbs[count].rdata     = NULL;  /* rec'd data    */
        (*teva)->ccbs[count].memFrames = NULL;  /* recv buffers  */
        }
```

```
}
```

2.5 Tools For Parallelism

2.5.1 Protocol Parallelism

As indicated in the introduction, protocol processing can be divided into two parts: packet processing and connection control block (CCB) manipulation. Packet processing includes external data representation conversion functions, checksum calculations, encryption and decryption, and encoding and decoding of packet headers. CCB manipulation includes state machine transitions, enqueuing received segments, etc. The majority of protocol processing overhead is due to packet processing.

Significant parallelism at the packet level (one thread per packet) is possible with reasonable levels of synchronization and communication complexity. Packet processing does not require access to shared data structures. Packets are generally context free in the sense that state information and synchronization are not required for encoding and decoding; only the packet is required for decoding and a reference to the packet contents for encoding. If only one thread is responsible for the encoding or decoding of a single packet, then there is no competition for these data structures. This makes packet processing an excellent candidate for parallelization. CCB manipulation is more difficult using per-packet parallelism. In this case more than one thread may be contending for access to shared data in the connection control block. A simple example is that of enqueuing a segment of a multi-segment PDU. If multiple segments are being enqueued in parallel, access to the list structure must be controlled to maintain list integrity. Likewise, state examination and transitions must be made in a consistent manner. It is therefore difficult to parallelize CCB manipulation.

In order to parallelize packet processing multiple threads can be created to "shepherd" packets through the protocol graph. Each of these threads executes in a loop as illustrated below:

```
PROCESS packetShepherd()
{
    while(1)
        {
        block at O/S interface awaiting a packet ...
```

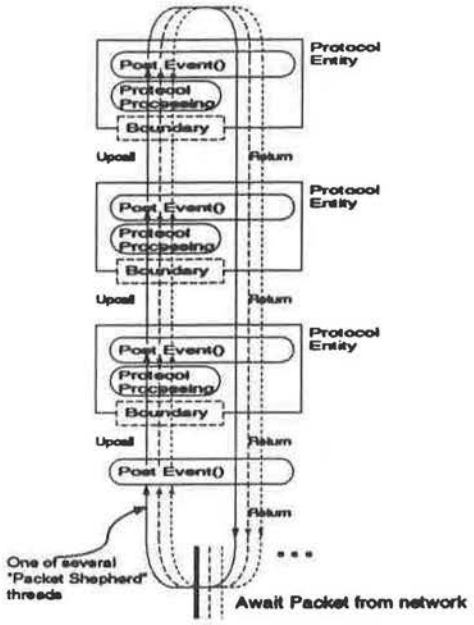

Figure 4: The Packet Shepherd Loop: Await Packet, Process Packet

```
        call PostEvent() to deliver packet to lowest protocol
        entity ... (and continue processing packet up through
        the graph until return)
        }
}
```

Application processes operate in a similar fashion. An application downcall is made to initiate data transfer. This thread proceeds through the layers using PostEvent() to execute downcalls. Once the thread reaches down as far as necessary (usually after interacting with the network device one or more times) it returns to the application. Figure 5 depicts application threads and downcalls. Figure 4 depicts the actions of a packet shepherd process.

Two types of synchronization between these threads are required: mutual exclusion and event ordering for the CCB manipulation stage.

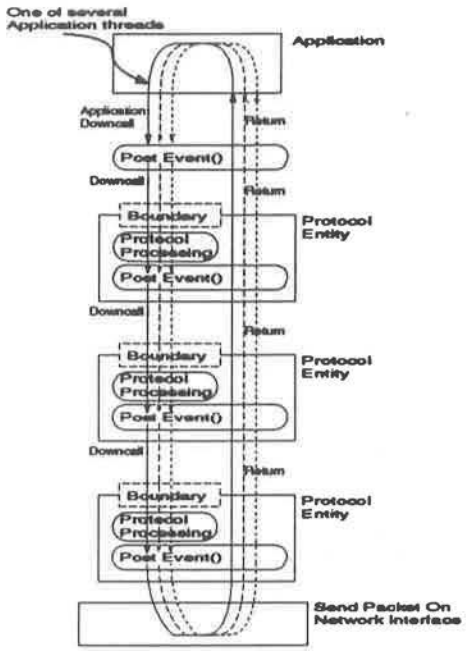

Figure 5: Application Threads initiating Data Transfer Through Downcalls

2.5.2 Mutual Exclusion

PPF provides constructs for ensuring mutual exclusion of CCB manipulation. This mutual exclusion is at the level of a single CCB. Since a CCB pertains to only one connection at a single layer, competition for access to a CCB is restricted to CCB manipulation occurring at the same layer for the same connection. No competition (other than that for shared hardware resources) occurs across layer or connection boundaries.

2.5.3 Event Ordering

Parallel processing of events (eg. received packets) allows the possibility that events may *pass* each other in the graph as they are being processed. This can cause a logically earlier event to arrive at some layer after a logically later event. In order for correct event processing the protocol machine must have knowledge of the logical order of received events. This is accomplished using internal sequence numbers. These sequence numbers are *internal* in the sense that they are private within a protocol implementation. They are an implementation

feature, invisible from a functional point of view.

Internal sequence numbers are generated by all entities which generate events and are associated one-to-one with these events. A sequence number stream is a monotonically increasing-by-one series. There is a separate sequence number stream for each event destination of each connection. For example, a session connection generates two independent sequence number streams. One for events destined for the associated presentation connection, and one for events destined for the associated transport connection. If one connection multiplexes events onto several service providers, separate sequence number streams are generated for each. The sequence number associated with each event determines its logical position in the event sequence regardless of arrival order.

There are also events which have no place (or perhaps more accurately, have *any* place) in the event stream. These are called *unordered* events. Unordered events are generally asynchronous events such as timer expiry. These events are assigned a special null sequence number indicating their unordered status.

2.5.4 The Event Gate

The *event gate* is a PPF module that controls the entry of threads into a connection's CCB manipulation stage. It performs both mutual exclusion and event ordering. An event arriving while the CCB manipulation stage is occupied is suspended and enqueued until the stage is clear. Events are allowed access sequentially in order of internal sequence number. An event arriving out of order is suspended until all intervening in-order events have passed through the CCB manipulation stage. These events may arrive from more than one source. For example, a connection receives events from at least its service user and service provider. Therefore each event gate must keep track of the separate sequence number streams. If more than one event is eligible to enter the CCB manipulation stage of a connection (these would be events from separate streams) the choice of which is allowed in first is arbitrary. The event gate also receives unordered events. These events are allowed in ahead of all other waiting events as soon as the CCB manipulation stage is free.

There is one event gate for each connection active within each layer. PPF parallelism structure requires no synchronization between separate connections, even at the same layer. Synchronization is only required between threads entering the CCB manipulation stage for

the same connection at the same layer. In order to make use of this synchronization tool, it is necessary for the protocol implementor to separate the CCB manipulation stage from the packet processing stage for each protocol entity. Experience shows that this is not a difficult task. Figure 6 shows the division of tasks in a connection within a protocol entity and the role of the event gate. Note that threads enter through the boundary routines, proceed in parallel through the packet processing stage, and proceed one at a time through the CCB manipulation stage.

The event gate module is composed of the following routines:

```
EVENTGATE *InitEventGate()
void FreeEventGate( EVENTGATE *gate )

void EnterEventGate(EVENTGATE *gate, int sequenceNumber)
void ExitEventGate(EVENTGATE  *gate)

#define NOSEQNO 0x0ffffff
#define FIRSTSTREAM(X) ((X) << 28)
```

InitEventGate() creates, initializes, and returns a pointer to a new event gate. Each connection of each protocol entity requires one event gate (assuming a parallel implementation - as is assumed for the remainder of this section). Event gates can be created at startup (when the CCBs are created), or dynamically (as connections are established). Each event gate expects to receive events from multiple sources (called *event streams*). The maximum number of streams is a configuration constant. Normally, an event gate will receive events from two streams, the associated connections of the service provider and the service user. Streams are numbered sequentially starting from zero. The protocol implementor must ensure that sources of events destined for a particular event gate each have separate stream identifiers. The convention for the simple case is that downward events are carried on stream 1, upward events on stream 0.

FreeEventGate() returns a previously allocated event gate to PPF. The single parameter is the pointer obtained from InitEventGate().

EnterEventGate() and *ExitEventGate()* calls surround an entities CCB manipulation code. EnterEventGate() ensures mutual exclusion of the CCB manipulation code and also sequences events according to its second parameter, *sequenceNumber*. ExitEventGate() is

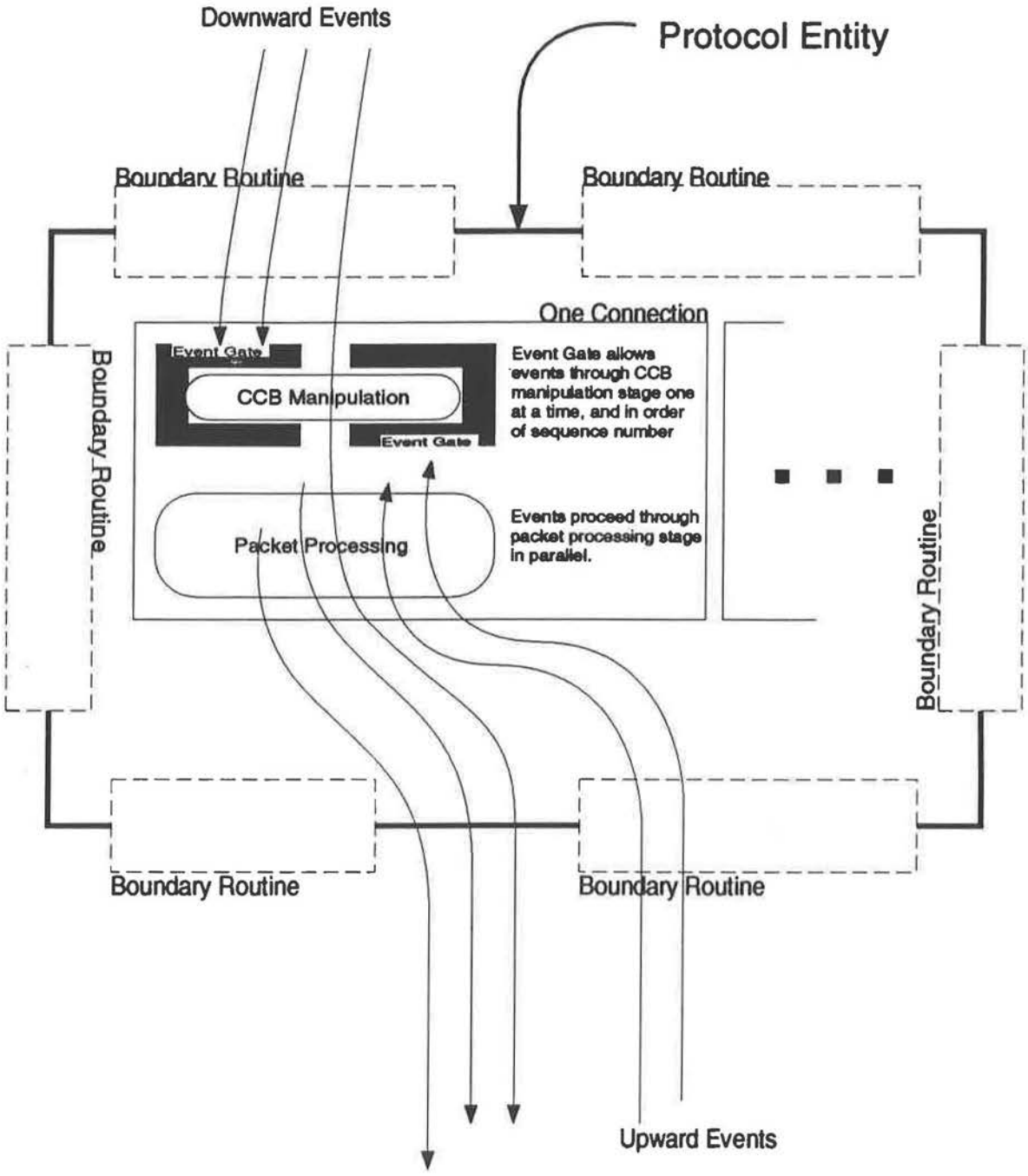

Figure 6: The Division of a Protocol Entity Into CCB Manipulation and Packet Processing Tasks

called after the last line of CCB manipulation code. Both routines require the event gate pointer as their first (and ExitEventGate()'s only) parameter. An example of the use of these routines follows:

```
/* this is a transport boundary routine */

TdataUp( vararea, cid, event, evpa, seqNo )
TransEVA *vararea;
int *cid;
EVENT event;
void *evpa;
SEQNO seqNo;
{
    TPDUINFO   tpdu;
    TransCCB *tempccb;

    /* first - the packet processing stage (PDU decoding in this case) */
    decTPDU(  &tpdu, (DATAHEADER *) evpa);

    tempccb = &(vararea->ccbs[*cid]);

    /* we are about to perform the CCB manipulation stage (i.e. state  */
    /* table transitions and list manipulation).                       */
    EnterEventGate( tempccb->eventGate, seqNo );

    /* make the event the type of received frame                       */
    event = MAKEEVENT( GETBOUNDRY( event ), tpdu.pdutype );

    /* call the appropriate state machine                              */
    (*(vararea->statetab[tempccb->state]))(tempccb,*cid,event,&tpdu,seqNo);
}
```

The thread processing this *dataup* event may continue on up the protocol graph. ExitEventGate() must be called immediately following the CCB manipulation performed on behalf of this event. Care must be taken to be sure that all possible "exits" from this entity are covered. One of the possible states called by the last statement of TdataUp() (shown above) is presented below. Note how the final CCB manipulation statement in each case is followed by a call to ExitEventGate().

```
topen( TransCCB *tempccb, int cid, EVENT event, void *evpa, SEQNO seqNo )
{
    switch ( GETEVENTP( event ) )
        {
        case TDataReq:
            {
            int maxlen, dncid;
            int numbPDUs;
            SEQNO firstSeqNo;

            maxlen     = tempccb->tpdusize;
            dncid      = tempccb->dncid;
            numbPDUs   = countsdata( (PDUNODE *) evpa, maxlen );

            firstSeqNo = tempccb->nextDnSeqNo;
            tempccb->nextDnSeqNo += numbPDUs;

            ExitEventGate( tempccb->eventGate );

            /* construct the PDU and post it to the layer below */
            sendTDT( evpa, maxlen, dncid, numbPDUs, firstSeqNo );
            }
            break;

        case DT:
            {
            TPDUINFO   *tpdu;
            DATAHEADER *dh, *newdh;

            tpdu = (TPDUINFO *) evpa;

            dh = tpdu->frameptr;

            /* enqueue PDU (OK as we are protected by event gate) */
            addTdh( tempccb, dh );

            if( tpdu->eot )  /* if this packet is complete */
                {
                /* the data and header are on the top mem frame */
                int  upcid;
```

```
            int  upSNo;

            upcid    = tempccb->upcid;
            newdh    = streamlist( tempccb );
            upSNo    = tempccb->nextUpSeqNo++;

            ExitEventGate( tempccb->eventGate );

            PostEvent(SESSION, &upcid, MAKEEVENT(DATAUP,TDataInd),
                      newdh, upSNo);
            }
        else
            ExitEventGate( tempccb->eventGate );
        }
        break;

        ... code deleted for brevity ...

    }
}
```

Notice, in the above example, that ExitEventGate() is called before the upcall or downcall (PostEvent() and sendTDT() [which calls PostEvent()], respectively) to the adjacent protocol layer. If, instead, the call to ExitEventGate() was not made until sometime after the upcall or downcall, the CCB manipulation stage would be locked until the upcall or downcall returned. This could be a considerable duration as any amount of CCB manipulation and packet processing could be carried out in an arbitrary number of protocol entities before control returns. Thus, ExitEventGate() is called after the last CCB manipulation statement, but before control is passed to another protocol entity.

Generating sequence numbers is done using the NOSEQNO and FIRSTSTREAM(X) macros. NOSEQNO is the *non*-sequence number. It is used for events which can take any place in the sequence number stream. An example of such an event is a time-out event. A normal sequence number stream is initiated using FIRSTSTREAM(). FIRSTSTREAM takes a single parameter identifying the stream number (typically some small value such as 0, 1 or 2). It returns the first sequence number in that stream. Subsequent sequence numbers are generated by incrementing the previous value. These must be generated in

"protected" mode (i.e. within the area guarded by the event gate) in order to ensure that the numbered sequence corresponds to the logical sequence. This usually involves code similar to the following:

```
/* generate next sequence number in protected space */
newseqno = tempccb->nextUpSeqNo++;

/* leave the protected area (CCB manipulation area) */
ExitEventGate( tempccb->eventGate );

/* post the event on to the next layer                */
PostEvent( NETWORK, &dncid, MAKEEVENT(CNTLDN, NConnReq),
            cinfo->calledAddr->nAddress, newseqno );
```

FIRSTSTREAM is normally called during CCB initialization as follows:

```
/* find a new connection identifier and initialize the connection */
int newtcid( )
    {
    /* make CCB creation mutually exclusive */
    P((*teva)->newcidSem);

    ... find a new CCB in the table ''ccbvec'' ...

    /* now that one is found - initialize the fields */
    ccbvec[tempcid].eventGate   = InitEventGate();

    /* generate first sequence number of stream 0 for events     */
    /* generated by this connection going upward.                */
    ccbvec[tempcid].nextUpSeqNo = FIRSTSTREAM(0);

    /* generate first sequence number of stream 1 for events     */
    /* generated by this connection going downward.              */
    ccbvec[tempcid].nextDnSeqNo = FIRSTSTREAM(1);

    /* set the connection state to some initial state */
    ccbvec[tempcid].state       = RESERVEDCID;

    V((*teva)->newcidSem);
```

```
return(tempcid);
}
```

2.5.5 Summary

There are two parts to protocol processing: CCB manipulation and packet processing. Packet processing overhead dominates protocol processing. Packet processing can be executed in parallel (even within a single connection). CCB manipulation must be atomic and synchronized. PPF provides the Event Gate mechanism to synchronize CCB manipulation activities. It ensures that (in the presence of multiple processors) CCB manipulation is atomic and that events are processed in their correct order.

References

[CCI87] CCITT. Presentation Protocol Specification for Open Systems Interconnection for CCITT Applications, December 1987.

[CCI89a] CCITT. Session Protocol Specification for Open Systems Interconnection for CCITT Applications. In *Blue Book, Volume VIII - Fascicle VIII.5*, 1989.

[CCI89b] CCITT. Transport Protocol Specification for Open Systems Interconnection for CCITT Applications. In *Blue Book, Volume VIII - Fascicle VIII.5*, 1989.

[Che88] David R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):315–333, March 1988.

[Cla85] David D. Clark. The structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, volume 19, pages 171–180, Cambridge, MA 02139, USA, December 1985. Laboratory for Computer Science, Massachusetts Institute of Technology.

[CT90] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM SIGCOMM 90, Communication Architectures and Protocols*, pages 200–209. Laboratory for Computer Science, M.I.T., 1990.

[GNI92] Murray W. Goldberg, Gerald W. Neufeld, and Mabo R. Ito. A parallel approach to osi connection-oriented protocols. In *IFIP WG6.1/WG6.4 Third International Workshop on Protocols for High-Speed Networks*, Vancouver, British Columbia, Canada, May 1992. University Of British Columbia.

[HP88] Norman C. Hutchinson and Larry L. Peterson. Design of the x-kernel. In *ACM Sigcomm 1988 Symposium*, pages 65–75. Computer Systems Laboratory, Stanford University, August 1988.

[NG90] Gerald W. Neufeld and Murray W. Goldberg. A request/response protocol for iso remote operations. In *IEEE Region 10 Conference on Computer and Communication Systems*, Vancouver, British Columbia, Canada, September 1990. University Of British Columbia.

[VL87] George Varghese and Tony Lauck. Hashed and hierarchical wheels: Data structures for the efficient implementation of a timer facility. *ACM Operating Systems Review*, 21(5):25–38, November 1987.