x-Kernel Programmer's Manual (Version 3.2)

Norman C. Hutchinson*
Larry L. Peterson
Sean W. O'Malley
Edwin F. Menze III
Hilarie K. Orman †

September 23, 1992

Abstract

This report describes how to implement protocols in the x-kernel. It gives the x-kernel's programming interface, describes how to configure an x-kernel that contains a certain collection of protocols, and demonstrates how to run and debug an x-kernel. The x-kernel can be run in one of two environments: as a user program on a Sun SparcStation running SunOS4.1, or as a task on a Mach3.0 system (Decstation 5000). Protocols can be moved between the two environments without modification. This document assumes that the reader is already familiar with the x-kernel's object-oriented infrastructure for implementing protocols.

---

*University of British Columbia

†This work supported in part by National Science Foundation Grants CCR-8811423 and IRI-9015407, National Aeronautics and Space Administration Grant NCC-2-561, Sun Microsystems Inc., and DARPA Contract DABT63-91-C-0030

Contents

1 Introduction

This draft document describes the *x*-kernel 3.2 interface, noting important differences from 3.1. It is an evolving document that is intended to track the 3.2 interface closely during development. The last modification was made on September 23, 1992.

The *x*-kernel is an object-oriented multi-threaded architecture for protocol implementation. Each protocol is represented as an object instance; sessions are specialized instances of protocol objects; each thread is a calling sequence of object invocations. Generally speaking, each message in an *x*-kernel is processed by a thread of protocol/session layers. The *x*-kernel provides the interface functions that allow one object to invoke another and subsystems for manipulating messages, associative memory tables, events, addresses, and debugging.

Version 3.2 of the *x*-kernel has changed significantly from version 3.1. Unlike 3.1, which was a true *kernel* in the sense that it ran stand-alone on Sun3 workstations, 3.2 is really a *protocol implementation environment* that can be embedded in any platform, including both hardware and software platforms. By defining all the interfaces between the protocol and the host environment, the *x*-kernel completely isolates the protocol from the underlying operating system. As a result, protocol source code can be moved from one platform to another without modification.

This manual covers the *x*-kernel functionality, the protocol library, and the tools for building and running *x*-kernels on various platforms. Sections 2–9 define the uniform protocol interface and the various libraries that make up the *x*-kernel. Sections 10 and 11 give additional guidelines about using these routines. Finally, Sections 12–16 describe procedures for maintaining, configuring, running, and installing the *x*-kernel.

Please let us know of any problems you encounter so that we can continue to improve the distribution. Our mail address is:

The *x*-Kernel Project
Department of Computer Science
University of Arizona
Tucson, AZ 85721

We can be reached by electronic mail at:

xkernel-bugs@cs.arizona.edu
uunet!arizona!xkernel-bugs

Because of limited resources we can't promise to fix every problem, but we appreciate all comments and will acknowledge all mail.

2 Uniform Protocol Interface

Each *x*-kernel protocol is encapsulated in a uniform protocol interface. The suite of protocols configured into the system form a *protocol graph* and the collection of currently opened sessions (connections) form a *session graph.*

2.1 Type Definitions

2.1.1 XObj

An *XObj* is the fundamental identifier in the system. It can represent either a protocol or a session. Fields which can be read or written by the programmer are so indicated.

```
typedef struct xobj {
    XObjType     type;
    char         *fullName;     /* the name given in graph.comp, e.g., "ethdrv/SE0" */
    char         *name;         /* the protocol name, e.g., "ethdrv" */
    char         *instName;     /* the instance name, e.g., "SE0" */
    char         *state;        /* Readable/writable – Owned by the session/protocol */
    int          rcnt;          /* readable */
    int          id;            /* readable */
    int          instance;      /* readable */
    /* Interface functions */
    Pfo          open;          /* protocols only */
    Pfk          closedone;     /* protocols only */
    Pfk          openenable;    /* protocols only */
    Pfk          opendisable;   /* protocols only */
    Pfk          opendone;      /* protocols only */
    Pfk          demux;         /* protocols only */
    Pfk          calldemux;     /* protocols only */
    Pfk          pop;           /* sessions only */
    Pfk          callpop;       /* sessions only */
    Pfh          push;          /* sessions only */
    Pfk          call;          /* sessions only */
    Pfk          close;         /* sessions or protocols */
    Pfi          control;       /* sessions or protocols */
    int          numdown;       /* readable */
                 /* down info is owned by the protocol for its sessions */
    int          downlistsz;
    struct xobj  *down[8];
    struct xobj  **downlist;
    struct xobj  *myprotl;      /* readable – a session is an instance of this protocol */
    struct xobj  *up;           /* for sessions only */
} *XObj;
```

If you think of the *x*-kernel as implementing protocol and session graphs, then each *XObj* represents a node in one of the graphs. Furthermore, an object's *down* vector represents graph edges;

they contain pointers to the *XObjs* that are below the object in the protocol or session graphs. The fields *myprotl* and *up* link a session to the protocols that own and created the session, respectively.

Note that the one aspect of Version 3.2 that differs most from descriptions of the *x*-kernel found in the literature is that *protocol* and *session* objects have been unified, and replaced with a single type *XObj*. As a consequence, routines that really require a session or a protocol object check the type field of the *XObj* object, where session objects have the type *Session* and protocol objects have the type *Protocol*.

2.1.2 Enable Objects

The *Enable* object is provided for use by the programmer in maintaining maps that associate identifiers and protocols or sessions. A pointer to an *Enable* object is often used as the internal identifier in *mapBind* calls. An *Enable* object has a field for reference counting. Protocol writers will find this useful in handling *xOpenEnable* calls with identical participants—the calls are redundant with respect to session creation—but they must be reference counted in order to properly handle *xOpenDisable* calls.

```
typedef struct xenable {
        XObj        hlpRcv;      /* upper protocol */
        XObj        hlpType;     /* upper protocol */
        Bind        binding;     /* from mapBind */
        int         rcnt;        /* A use count */
} Enable;
```

2.1.3 Return Values

Most routines have a return value type of *xkern_return_t*, which is either *XK_SUCCESS* or *XK_FAILURE*. Routines that return type *XObj* have a failure value of *ERR_XOBJ*. Some message handling routines use type *xmsg_handle*; see section 2.2.5. Severe error conditions will result in console error messages and the termination of the *x*-kernel.

2.1.4 Function Types

The following typedefs are used in the *XObj* structure and throughout this document:

```
typedef xkern_return_t  ( * Pfk )();
typedef void  ( * Pfv )();
typedef struct xobj  ( * Pfo )();
typedef xmsg_handle_t  ( * Pfh )();
typedef int  ( * Pfi )();
```

2.2 Protocol-Protocol Operations

This section defines the operations that protocols (and sessions) invoke on each other. In general, each of the operations invokes a corresponding operation in the target protocol or session. For example, an *xOpen* call will result in the invocation of a protocol specific open routine, e.g. *udp_open*. For each operation, we give the interface to both the generic *x*-kernel operation and an example protocol-specific procedure that implements the generic operation. Although nearly the same, the specification for the generic operation and the specification for the protocol-specific routine typically differ in that a *self* pointer is passed to the protocol-specific routine.

2.2.1 xOpen

The *xOpen* function is used by high-level protocol *hlpRcv* to actively open a session associated with low-level protocol *llp* on behalf of high-level protocol *hlpType*. Typically, *hlpRcv* and *hlpType* refer to the same protocol (see section 2.4.1.) The *participants* argument is a list of addresses for each participant in the communication. For this, and all calls returning type *XObj*, a return value of *ERR_XOBJ* indicates failure, and this must be checked by all callers before using the return value.

Caveat: the high-level protocol will use its *self* object as the first argument in *xOpen*, and the lower-level protocol object as the third argument. The lower-level's open routine will see its own *self* object as the first argument, and the high-level protocols as the second and third arguments. This reversal of argument order preserves the convention that the current protocol's self object is the first argument in the list.

Generic: *XObj* **xOpen**(*XObj* hlpRcv, *XObj* hlpType, *XObj* llp, *Part* *participants)

Specific: *XObj* **udp_open**(*XObj* self, *XObj* hlpRcv, *XObj* hlpType, *Part* *participants)

2.2.2 xOpenEnable

Used by high-level protocol *hlpRcv* to passively open a session associated with low-level protocol *llp* on behalf of high-level protocol *hlpType*. As with *xOpen*, *hlpRcv* and *hlpType* usually refer to the same protocol. A passive open indicates a willingness to accept connections initiated by remote participants. A session is not actually returned, but the low-level protocol, by convention, "remembers" this enabling, and later calls the high-level protocol's *xOpenDone* operation to complete the passive open. *participants* is an ordered list of addresses for each participant for which the communication has been enabled. In most cases, it contains only a single element: the address of the local participant. A return value of *XK_FAILURE* indicates failure.

The lower-level protocol generally "remembers" an invocation of its *xOpenEnable* operation by binding an *Enable* object to the participant information using *mapBind*.

Generic: *xkern_return_t* **xOpenEnable** (*XObj* hlpRcv, *XObj* hlpType, *XObj* llp, *Part* *p)

Specific: *xkern_return_t* **udp_openenable** (*XObj* self, *XObj* hlpRcv, *XObj* hlpType, *Part* *p)

2.2.3 xOpenDisable

Used by high-level protocol *hlpRcv* to undo the effects of an earlier invocation of *xOpenEnable*. The contents of the *participant* argument, *hlpRcv* and *hlpType* must be the same as the ones given to *xOpenEnable*.

Generic: *xkern_return_t* **xOpenDisable**(*XObj* hlpRcv, *XObj* hlpType, *XObj* llp, *Part**p)

Specific: *xkern_return_t* **udp_opendisable**(*XObj* self, *XObj* hlpRcv, *XObj* hlpType, *Part**p)

2.2.4 xOpenDone

Used by low-level protocol to inform high-level protocol that session (s) has now been created corresponding to an earlier *xOpenEnable* on behalf of *hlpType*.

> *3.1 compatibility note:* This is like 3.1's x_callopendone without participants or hlp; the original x_opendone no longer exists.

Generic: *xkern_return_t* **xOpenDone** (*XObj* hlpRcv, *XObj* s, *XObj* llp)

Specific: *xkern_return_t* **udp_opendone** (*XObj* self, *XObj* lls, *XObj* llp, *XObj* hlpType)

2.2.5 xPush

Used by a high-level protocol that opened session s to pass the message *msg* down through that session. The return value is an opaque handle on the message that was sent. This handle may be used to identify this message in subsequent *xControl*. A return value of *XMSG_ERR_HANDLE* indicates failure.

> *3.1 compatibility note:* The reply message arg from 3.1 has been eliminated. Use xCall if a reply message is expected.

Generic: *xmsg_handle_t* **xPush** (*XObj* s, *Msg* *msg)

Specific: *xmsg_handle_t* **udp_push** (*XObj* self, *Msg* *msg)

2.2.6 xCall

Similar to *xPush* except that a "reply" message may be returned through the argument *reply_msg*. Used with synchronous (RPC-like) protocols. Because the lower protocol typically retains no state for the request message after *xCall* returns, a message handle is not returned. The message structure for the reply must be initialized; see msgConstructEmpty.

Generic: *xkern_return_t* **xCall** (*XObj* s, *Msg* *msg, *Msg* *reply_msg)

Specific: *xkern_return_t* **udp_call** (*XObj* s, *Msg* *msg, *Msg* *reply_msg)

2.2.7 xPop

Used by a protocol to pass an incoming message *msg* up to session s for processing, and to indicate the lower-level session from which the message was received (*lls*). This calls the pop routine of the session s and increments the session reference count (see section 11). This call is invoked by a protocol on one of its own sessions.

To facilitate "stream" protocols, where there is a one-to-one correspondence between high-level and low-level sessions, the arguments to the specific instances of protocol functions for *xPop* and *xDemux* are compatible. This allows a high-level protocol to create a session object has a "demux" function pointer that points to a "pop" routine, and to modify the associated lower-level session to point to this session as its "up" object. When the lower session calls *xDemux* to deliver the message upward, it is vectored directly to the correct upper level session and its pop routine, thus optimizing message delivery. Protocols that take this approach and bypass *xPop* must take special reference count precautions (see section 11.)

Generic: *xkern_return_t* **xPop** (*XObj* s, *XObj* lls, *Msg* *msg)

Specific: *xkern_return_t* **udp_pop** (*XObj* self, *XObj* lls, *Msg* *msg)

2.2.8 xCallPop

Similar to *xPop* except it allows the upper protocol to return a message. Used with synchronous (RPC-like) protocols.

Generic: *xkern_return_t* **xCallPop** (*XObj* s, *XObj* lls, *Msg* *msg, *Msg* *rmsg)

Specific: *xkern_return_t* **udp_callpop** (*XObj* self, *XObj* lls, *Msg* *msg, *Msg* *rmsg)

2.2.9 xCloseDone

Used by a low-level protocol to inform the high-level protocol that originally opened session *s* that the session has been closed by a peer participant.

Generic:*xkern_return_t* **xCloseDone** (*XObj* s)

Specific: *xkern_return_t* **udp_closedone** (*XObj* s)

2.2.10 xClose

Decrements the reference count of an *XObj* (see section 11), calling the object's close function *only* if the reference count is zero. This call applies to *XObj*s that represent both protocols and sessions, but the implications of invoking *xClose* on a protocol is not fully determined at this time.

> *3.1 compatibility note:* Replaces xCloseProtl and xCloseSession. Note that in 3.1, the *XObj*'s close function was called every time xCloseProtl or xCloseSession was called.

Generic: *xkern_return_t* **xClose** (*XObj* object)

Specific: *xkern_return_t* **udp_close** (*XObj* object)

2.2.11 xControl

Used by one *XObj* to act upon another *XObj* for retrieving information or for setting processing parameters. The operation code *opcode* identifies the action; *buf* is a character buffer from which an argument is retrieved and/or into which a result is placed; and *len* is the length of the buffer. There are two "classes" of operations: standard ones that may be implemented by more than one protocol, and protocol-specific ones. A full discussion of control operation codes is in section 10.

> *3.1 compatibility note:* The routines controlprotl and controlsessn no longer exist.

Generic: *int* **xControl** (*XObj* object, *int* opcode, *char* *buf, *int* len)

Specific: *int* **udp_control** (*XObj* self, *int* opcode, *char* *buf, *int* len)

2.2.12 xDemux

Used by low-level session *s* to pass message *msg* to the high-level protocol that created it. The higher-level protocol is determined from the session object. The higher-level protocol demux routine should find the appropriate session, creating it if necessary, and *xPop* the message to the session. See 2.3.1 for guidelines on when session creation is appropriate.

The called routine is presented with its protocol object, the session object from the lower-level protocol, and the message.

Generic: *xkern_return_t* **xDemux** (*XObj* s, *Msg* *msg)

Specific: *xkern_return_t* **udp_demux** (*XObj* self, *XObj* lls, *Msg* *msg)

2.2.13 xCallDemux

This call is like *xDemux* but allows an argument to contain a return message. Used with synchronous (RPC-like) protocols.

Generic: *xkern_return_t* **xCallDemux** (*XObj* s, *Msg* *msg, *Msg* *rmsg)

Specific: *xkern_return_t* **upd_calldemux** (*XObj* self, *XObj* lls, *Msg* *msg, *Msg* *rmsg)

2.2.14 xDuplicate

Increments the reference count of session *s*. This can be used to create a permanent handle on the *XObj* from a temporary handle, or to create a new equivalent handle from an existing handle. For a full discussion of session reference counts, see section 11.

Generic: *xkern_return_t* **xDuplicate** (*XObj* s)

Specific: *xkern_return_t* **udp_duplicate** (*XObj* self)

2.3 Graph Manipulation Operations

Unlike the previous set of operations—which protocols invoke on each other to open/close connections and to send/receive messages—the operations defined in this section actually manipulate the protocol and session graphs; i.e, create nodes and edges. These operations are either called by the *x*-kernel at start-up time to create and link together protocol objects, or by protocols at runtime to create and link together session objects.

2.3.1 xCreateSessn

Called by a protocol to create a session that will handle data associated with a common source/destination pair. Usually called in response to an *xOpen* call, or because data has arrived with participants that match a previous *xOpenEnable* call. By convention, a protocol will only create one session at a time for a source/destination pair, even if there have been multiple *xOpenEnable*'s that would match incoming data.

The session is initialized using information found in high-level protocol *hlp* and low-level protocol *llp*. The count *downc* indicates how many lower level sessions this session requires. It need be non-zero only for sessions that will store sessions in the "down" vector *downv* (it will be zero for device drivers and one for normal sessions.) The initialization function pointer *f* may be null; otherwise this function should fill in the interface function pointers in the *XObj* structure. These pointers are initialized to default (usually null) functions by the system initialization code.

XObj **xCreateSessn**(*Pfv* f, *XObj* hlp, *XObj* llp, *int* downc, *XObj* *downv)

Example Init Function: *void* **udp_sessn_init** (*XObj* self)

2.3.2 xCreateProtl

Called during system start-up for each protocol in the graph. The function *f* is called to initialize a protocol object. It must have a well-known name derived from the concatenation of the protocol name and the string "*_init*" (e.g., *tcp_init*). The initialization generally allocates and initializes protocol state and fills in the interface function pointers. Because function pointers are initialized to null functions before *f* is called, only those functions actually used by the protocol need be defined.

Use of *xCreateProtl* outside of initialization, e.g., to dynamically load new protocols, is not supported at this time.

> *3.1 compatibility note:* Replaces x_instantiateprotl and x_createprotl (the user routine) and eliminates the need for x_getproc.

XObj **xCreateProtl** (*Pfv* f, *char* *name, *int* downc, *XObj* *downv)

Example Init Function: *void* **udp_init** (*XObj* self)

2.3.3 xDestroy

Destroys protocol and session objects. It is the inverse of *xCreateSessn* and *xCreateProtl*. Storage for the *XObj* is freed, and if the state pointer of the *XObj* is non-null, it is also freed.

xkern_return_t **xDestroy**(*XObj* object)

2.3.4 xGetProtlByName

Returns a capability for a protocol object given its mnemonic name. (See the discussion of graph.comp in section 13.)

XObj **xGetProtlByName** (*char* *name)

2.3.5 xSetDown

Sets the *index*th member of *self*'s down vector to be *object*. It sets or increments the *XObj* field *numdown* as a side effect.

xkern_return_t **xSetDown** (*XObj* self, *int* index, *XObj* object)

2.3.6 xGetDown

Returns the *index*th member of *self*'s down vector. Returns *ERR_XOBJ* if the index is larger than the down vector.

XObj **xGetDown** (*XObj* self, *int* index)

2.4 Usage Rules

This section has some protocol design rules that protocol writers should follow in order to develop "well-behaved" protocols that interact properly with other protocols with which they might be composed.

2.4.1 hlpRcv and hlpType

The operations *xOpen*, *xOpenEnable*, and *xOpenDisable* take two high-level protocols, *hlpRcv* and *hlpType*. *hlpRcv* is the protocol to which the new lower session should route incoming messages. The lower protocol uses *hlpRcv* as the hlp argument to *xCreateSessn*. The lower protocol uses *hlpType* to determine which messages the new session should handle. For example, when *ethOpen* is called with IP as the *hlpType*, ETH knows that the new session will deal with packets that have the IP ethernet type. The lower protocol typically determines the number that corresponds to *hlpType* by using it in a call to *relProtNum* (see section 12.)

Most protocols use their *self* pointer as both *hlpRcv* and *hlpType* when making these calls, though virtual protocols (see below) are the exception.

2.4.2 Protocol Realms

Although the *x*-kernel defines a single interface for all protocols, not all protocols are created equal. Our experience is that protocols can be classified into different categories, which we call realms. Chances are, any protocol you write falls into one of the following realms. In some cases, the realm a protocol falls into both defines a restricted subset of the interface that the protocol implements, and the set of protocols with which it may be composed.

Asynchronous Protocols

Most protocols—e.g., protocols like IP, TCP, and UDP—fall in this category. The *x*-kernel supports asynchronous protocols through the use of *xPush*, *xPop* and *xDemux* operations. Asynchronous protocols are typically symmetric in the sense that the protocol's sessions process both incoming and outgoing messages. While it seems possible for asynchronous protocols to have asymmetric sessions (a given session can handle only incoming or outgoing messages, but not both), we have thus far been able to make all our asynchronous protocols be symmetric, and we strongly encourage such designs. Knowing that any low-level protocols you may use are symmetric enhances our ability to compose protocols and makes implementing a given protocol much easier.

Synchronous Protocols

These are RPC protocols. They are typically asymmetric in the sense that client-side sessions and the server-side sessions are quite different. The *x*-kernel explicitly supports synchronous/asymmetric sessions through the use of *xCall*, *xCallPop* and *xCallDemux*. Since synchronous protocols are asymmetric, *xCall* is used on the client side and *xCallPop* and *xCallDeumx* are used on the server side.

Note that some protocols lie on the boundary between the synchronous and asynchronous realms. For example, a protocol that implements RPC (as opposed to one that uses it) probably looks asynchronous from the bottom (i.e., lower level protocols call its *xPop* routine), but synchronous from above (i.e., higher level protocols call its *xCall* routine).

Control Protocols

These protocols support neither a *xPush/xPop* nor a *xCall/xCallPop* interface. Typically, only control operations may be performed on these protocols. ARP and ICMP fall into this category.

2.4.3 Virtual Protocols

Virtual protocols occupy places in the protocol (and sometimes the session) graphs, but they neither produce nor interpret network headers. They typically make decisions about how messages should be routed through the session graph based on participants in *xOpen* or on properties of messages, such as size.

The *xOpen*, *xOpenEnable*, and *xOpenDisable* routines of virtual protocols differ from those of conventional protocols. A virtual protocol's implementation of *xOpen*, for example, will usually make an *xOpen* call to its lower protocols using the *hlpType* that was passed into the virtual protocol, but using its *self* pointer as *hlpRcv*. This allows arbitrary chains of virtual protocols to insert their sessions between the upper and lower conventional sessions while still passing "type information" from the upper protocol to the lower protocol.

Note that virtual protocols can be either synchronous (support the *xCall/xCallPop/xCallDemux* interface) or asynchronous (support the *xPush/xPop/xDemux* interface).

3 Participant Library

Participant lists identify members of a session and are used for opening connections. Upper protocols interested in establishing a connection will construct the participant list and pass it to the lower protocol as a parameter of an open routine. The lower protocol will then extract information from the participant list, possibly passing the participant list on to its own lower protocol.

Each participant in the list contains a participant address stack, designed to facilitate a general method of communicating encapsulated address information between protocol layers. By using pointers to address information, one layer can pass address information through a lower layer without having the lower layer manipulate the address information at all, not even by copying. The address information for each participant is kept as a list of of *void* * pointers to address components. The component pointers are pushed or popped onto the stack by utility functions.

3.1 Type Definitions

The participant data structure is used to collect addressing information for opening connections. A participant list is defined to be an array of type *Part*, and a *PartStack* is the main field in a single *Part*. The fields of these structures should not be directly accessed by the protocol developer.

```
#define         PART_MAX_STACK    20
typedef struct {
        void *      arr[PART_MAX_STACK];
        int         top;
} PartStack;

typedef struct {
        int         len;
        PartStack   stack; /* A stack of void* pointers */
} Part;
```

3.2 Participant List Operations

The following operations provide a convenient interface that hides the *PartStack* data structure. However, the fact that a participant list is really an array of type *Part* is visible to the programmer.

3.2.1 partInit

Initialize participant list *p* to have *N* entries.

void **partInit**(*Part* *p, *int* N);

3.2.2 partPush

Push address *addr* onto participant stack *p*.

void **partPush**(*Part* p, *void* * addr)

3.2.3 partPop

Pop an address off of participant stack *p*. Returns NULL if there are no more elements on the stack.

*void ** **partPop**(*Part* p)

3.2.4 partLen

Report the number of participants in participant list *p*.

int **partLen**(*Part* p)

3.3 Relative Protocol Numbers

Participant lists are used for passing addressing information between protocols. An additional problem is how a high-level protocol identifies *itself* to a low-level protocol. In most conventional protocols, a low-level protocol uses a *relative protocol number* to identify the protocols above it; e.g., IP identifies UDP with protocol number 17 and TCP as protocol number 6. However, protocols that have been especially designed to use the *x*-kernel use an *absolute* addressing scheme.

Version 3.2 of the *x*-kernel reconciles these two approaches by maintaining a table of relative protocol numbers. (See Section 12 for the format of this table.) Rather than embed protocol numbers in the protocol source code, protocols learn the protocol numbers of protocols above them by querying this table using the following operation:

xkern_return_t **relProtNum**(*XObj* hlp, *XObj* llp)

This operation returns the protocol number of the high-level protocol relative to the low-level protocol. This number will have to be cast into the appropriate type; e.g., an unsigned short by the ETH protocol and an unsigned char by IP.

3.4 Usage Rules

By convention, active participant lists have the remote participant(s) first followed by an optional local participant. The local participant can often be omitted, in which case the protocol tries to use a reasonable default. (For example, a UDP participant contains a UDP port and an IP host. If the local participant is missing from an active participant list, UDP selects an available port for the local participant.)

In some cases, it is necessary to specify part of the information in a participant, but it is convenient to allow the lower protocol to "fill in" the rest. To allow this flexibility, one can use the constant pointer ANY_HOST to specify "wildcard" hosts. For example, if one wants to open UDP with a specific local port, but does not care which local host number is used, one could construct a local participant with the specific local port but with the pointer ANY_HOST pushed on the stack. The protocol that interprets the host part of the participant stack could then choose a reasonable default.

A wildcard value ANY_PORT also exists for protocols that use ports on their stacks. Protocols that support this wildcard indicate this in their manual page.

When using *PartStack*s that have been passed from other protocols, the user must keep in mind that the address pointers may be valid only for the duration of the current subroutine. Data that is needed beyond this time should be explicitly copied into static storage. In addition, because the participant structure is passed by reference in the *xOpen* call, the caller should consider the contents invalid after the return.

4 Message Library

The message library provides a set of efficient, high-level operations for manipulating messages. The underlying data structure that implements messages is optimized for fragmentation/reassembly, and for adding/stripping headers. Protocol programmers should manipulate messages only with the operations documented here.

Future releases of the message library will accomodate the use of buffers that are owned and managed by different buffer pools, including device memory, user buffers, simple kernel buffers, and structured kernel buffers such as BSD *mbufs* and System V *mblocks*.

4.1 Type Definitions

Loosely speaking, Version 3.2 messages include two main components: a *stack* that holds the message header, and pointer (*tree*) to a tree structure of buffers that hold the data.

```
typedef struct {
        char          *headPtr;
        char          *tailPtr;
        char          *stackHeadPtr;
        char          *stackTailPtr;
        struct {
                      unsigned short  numNodes;
                      unsigned int    myStack;
        } state;
        MNodeLeaf*    stack;
        MNode*        tree;
        void*         attr;
} Msg ;
```

4.2 Constructor/Destructor Operations

These operations are used to create and destroy messages. Many of them are, for example, used by device drivers and system call code that has to incorporate a data buffer into an *x*-kernel message.

4.2.1 msgConstructEmpty

A message structure is initialized with a data length set to zero.

void **msgConstructEmpty** (*Msg** this);

4.2.2 msgConstructBuffer

Copies data from a user buffer into a message structure. The message data area is allocated and a copy is performed. This constructor is used when the data buffer already exists. Use *msgConstructAllocate* when you have the opportunity to fill the buffer after it has been created.

void **msgConstructBuffer** (*Msg** this, *char* *buf, *long* len);

4.2.3 msgConstructAllocate

A data area of size *len* is allocated and associated with message *this*. A pointer to the data area is returned in *buf*. A device driver might use this constructor, handing *buf* to the device as a place to put down an incoming packet.

void **msgConstructAllocate** (*Msg** this, *long* len, *char* **buf);

4.2.4 msgConstructCopy

The uninitialized message *this* will refer to the same data as the message *another*. No data is copied. See also *msgAssign*.

void **msgConstructCopy** (*Msg** this, *Msg* *another);

4.2.5 msgAssign

The assignment of one message *m* to message *this* means that message *this* will refer to the same data that as message *m* currently does. No data copying is involved. This is equivalent to doing a *msgDestroy* to *this*, followed by a *msgConstructCopy*. Therefore, this function should be used only when both messages are valid. Copying to an uninitialized structure should be done with msgConstructCopy.

void **msgAssign** (*Msg** this, *Msg** m);

4.2.6 msgDestroy

Logically frees message *this*. Data portions of the message deallocated are freed as a result of this function should there be no other outstanding references to them.

void **msgDestroy** (*Msg** this);

4.3 Manipulation Operations

Protocols manipulate messages—e.g., add and strip headers, fragment and reassemble packets—using the following set of operations.

4.3.1 msgLen

Returns the number of bytes of data in message *this*.

long **msgLen** (*Msg* *this);

4.3.2 msgTruncate

Truncate the data in the message *this* to the given length. Attempts to to reduce the length to less than zero will result in no change to the message. No storage is freed as a result of truncation. This operation is used to strip trailers from a message.

void **msgTruncate** (*Msg** this, *long* newLength);

4.3.3 msgChopOff

Remove data from the message *this* and assign it to the message *head*. No copying is done. This operation is used to fragment a message into smaller pieces. Both messages must be valid at the time of the call.

3.1 Compatibility notes: The 3.1 version of this was *msg_break*, but the arguments were reversed.

void **msgChopOff** (*Msg** this, *Msg** head, *long* len);

4.3.4 msgJoin

Assign to *this* message the concatenation of messages *m1* to the front of *m2*. This operation is used to reassemble fragments into a larger message. The first argument must be a valid message. The arguments need not refer to distinct messages.

void **msgJoin** (*Msg** this, *Msg** m1, *Msg** m2);

Example of using non-unique arguments: **msgJoin** (this, this, MessageTail);

4.3.5 msgPush

Prepend data pointed to by *hdr* to the front of message *this*. The supplied function *store* will be called with the *hdr* pointer, a pointer to the message data area where the header data can be copied, the *hdrLen* value, and the *arg* pointer.

void **msgPush** (*Msg** this, *MStoreFun* store, *void* *hdr, *long* hdrLen, *void* *arg);

Function *store* typically converts the header to network byte order and stores it in a potentially unaligned buffer; *len* is the number of bytes to be read and *arg* is an arbitrary argument passed through from *msgPush*.

typedef void (***MStoreFun**)(*void* *hdr, *char* *des, *long* len, *void* *arg);

Example Declaration: *void* **copypush**(*void* *src, *char* *dst, *long* count, *void* *type))

4.3.6 msgPop

Removes data from the front of message *this*. After making the requested data contiguous, *msgPop* calls the function *load* with the *hdr* argument, a pointer to the start of the contiguous data, *hdrLen*, and the *arg* pointer. Function *load* should return an integer representing the amount data that should be popped off the message. In optimized code, no validity check is done on this number. This function returns TRUE if successful, FALSE if the message is shorter than *hdrLen* bytes.

bool **msgPop** (*Msg** this, *MLoadFun* load, *void* *hdr, *long* hdrLen, *void* *arg);

Function *load* loads the header from a potentially unaligned buffer and converts it to machine byte order. The *len* is the number of bytes guaranteed contiguous and *arg* is an arbitrary argument passed through from *msgPop*. This function should return the number of bytes to be actually popped from the message. This information is used by *msgPop* to update the internal message structure.

typedef long (***MLoadFun**)(*void* *hdr, *char* *src, *long* len, *void* *arg);

Example declaration: *long* **copypop**(*void* *dst, *char* *src, *long* count, *void* *type))

4.3.7 msgPopDiscard

Discards *len* bytes from the front of message *this*. Returns *TRUE* if successful.

bool **msgPopDiscard** (*Msg* *this, *long* len)

4.3.8 msgSetAttr

Associates an *attribute* with message *this*. A common use occurs in *xDemux* after a message header has been copied to a local buffer. The structure holding the header is then associated with the message using *msgSetAttr*.

void **msgSetAttr** (*Msg* *this, *void* *attribute);

4.3.9 msgGetAttr

Retrieve an attribute previously associated with a message *this*.

void **msgGetAttr** (*Msg* *this, *void* *attribute);

4.3.10 msgForEach

For each contiguous data area in the message, invoke the function *f* with three arguments: the address of the data, the length of the data, and a user supplied argument *arg*. The iteration stops when *f* does not return *TRUE*.

void **msgForEach** (*Msg* *this, *MCharFun* f, *void* *arg);

typedef bool (***MCharFun**)(*char* *ptr, *long* len, *void* *arg);

4.4 Usage Rules

The *x*-kernel coding conventions dictate that messages should be destroyed by the same entity that originally constructed them. Thus, the ethernet driver is responsible for destroying messages after successfully delivering them upward, and the top-level protocols that interface to user functions should destroy messages that have been successfully delivered to their destination.

With the exception of a protocol that creates a message, a protocol that passes a message to an adjacent protocol loses its reference to the message. The contents of the message at that point depend on which lower headers were pushed onto it. Should a protocol want to keep a reference to the message—e.g., so it can later retransmit it—it must explicitly save a copy using the *msgAssign* operation before passing the message on to another protocol.

Note that there is an internal data limit of slightly less than 16K bytes for any single message operation. However, multiple operations on a message may result in an aggregate message that is longer than 16K bytes.

5 Event Library

The event library provides a mechanism for scheduling a procedure to be called after a certain amount of time. By registering a procedure with the event library, protocols are able to "timeout" and act on messages that have not been acknowledged or to perform periodic maintenance functions.

5.1 Type Definitions

The only event-related type that protocol programmers need be aware of is the type *Event*. However, this type is defined by the underlying platform and is opaque to the protocol programmer.

5.2 Operations

5.2.1 evSchedule

Schedule an event that executes function *f* with argument *a* after delay *t* usec; *t* may equal 0. A handle to the event is returned, and this can be used to cancel the event at some later time. When an event fires, a new thread is created to run function *f*. Note that even after an event fires and a thread had been scheduled to handle handle it, the thread does not run until sometime after the currently executing thread gives up the processor. See Section 7 for a description of how threads are scheduled.

Event **evSchedule** (*Pfvv* f, *void** a, *int* t)

Function *f* must be of type *void* and take two arguments: the first, of type *Event*, is a handle to the event itself and the second, of type *void*, is the argument passed to evSchedule. In order to satisfy the C compiler type checking rules when accessing the arguments, function *f* must begin by casting its second argument to be a non-void type.

5.2.2 evDetach

Releases a handle to an event. As soon as *f* completes, the internal resources associated with the event are freed. All events should eventually be either detached or canceled to assure that system resources are released.

void **evDetach** (*Event* e)

5.2.3 evCancel

Cancels event *e* and returns EVENT_FINISHED if the event has already happened, EVENT_RUNNING if the event is currently running, and EVENT_CANCELLED if the event has not run and can be guaranteed to not run. In the case where **evCancel** returns EVENT_RUNNING, the caller must be careful to not delete resources required by the event.

EvCancelReturn **evCancel** (*Event* e)

5.2.4 evIsCancelled

Returns true if an **evCancel** has been performed on the event. Because event handlers receive their event as the first calling argument, it is possible for a handler to check for cancellation of itself from other threads.

int **evIsCancelled** (*Event* **e**)

5.3 Usage Rules

5.3.1 Repeating Events

Each event that is scheduled executes at most one time. Repeating events are programmed as follows:

```
foo_init()
{
    ...
    evDetach( evSchedule( f, argument, INTERVAL ) );
}


f( Event self, void *arg )
{
    actual work
    ...
    evDetach( evSchedule( f, arg, INTERVAL ) );
}
```

5.3.2 Cancellable Events

The evIsCancelled routine is designed to make it easy to write events which might be cancelled before (or while) they run. It is common practice, for example, for a session to pass session state to a timeout event. The evIsCancelled notification can be used to synchronize the timeout event and the possible destruction of the session state. Here is an example:

```
foo_destroy()
{
    ...
    evCancel( state->timeoutEvent );
    ...
}


foo_timeout( Event self, void *state )
```

```
{
   ...
   xPush(lls, retransmitMsg);
   /*
    * xPush may have blocked -- check to see if state is
    * still valid
    */
    if ( evIsCancelled(self) ) {
        return;
    }
    state->timeoutEvent = evSchedule( foo_timeout, arg, INTERVAL );
}
```

5.3.3 Event Granularity

Although the event library uses an efficient representation (timing wheels) protocol programmers should be careful to not schedule events that are too fine grained. For example, in TCP, it is better to schedule one event for every session rather than for every message that is sent.

6 Map Library

The map library provides a facility for maintaining a set of bindings between identifiers. The map library supports operations for adding new bindings to the set, removing bindings from the set, and mapping one identifier into another relative to a set of bindings (lookup). Protocol implementations use these operations to translate identifiers extracted from messages headers (e.g., addresses, port numbers) into capabilities for kernel objects (e.g., *XObj*, *Enable*).

6.1 Type Definitions

The map library defines two data structures: *MapElement* and *Map*. *Bind* is a pointer to a *MapElement*.

```
typedef struct mapelement {
        struct mapelement       *next;
        void *          internalid;
        void *          externalid;
} MapElement, *Bind;

typedef struct {
        int             nEntries, keySize;
        MapElement      *cache;
        MapElement      *freelist;
        MapElement      **table;
        void *          (*resolve)();
        Bind            (*bind)();
        xkern_return_t  (*unbind)();
        xkern_return_t  (*remove)();
        int             (*next)();
} *Map;
```

6.2 Operations

6.2.1 mapCreate

Creates a map with *nEntries* elements in it. External ids bound in this map are *keySize* bytes long. The maximum value for the key size is MAX_MAP_KEY_SIZE, currently 48 bytes. Programmers will normally use sizeof(structuretype) as the keysize to facilitate platform independence. Note that maps never overflow, but they perform best if *nEntries* is chosen so that the map is at most 50-80% full.

Map **mapCreate** (*int* nEntries, *int* keySize);

6.2.2 mapResolve

Looks for the internal identifier bound to the external identifier *ext* in the map *Map*. Note that the resolution will be done with *keysize* bytes of what the *ext* identifier points to, where *keysize* is the parameter that was used in the *mapCreate* call. If a binding is found, **ptr* is assigned the value of the internal identifier and XK_SUCCESS is returned. If no appropriate binding is found, mapResolve returns XK_FAILURE. If *ptr* is NULL, only the error code is returned.

xkern_return_t **mapResolve** (*Map* Map, *void* *ext, *void* **ptr,);

6.2.3 mapBind

Adds a binding of external id *ext* to internal id *intern* into map *Map*. Note that the binding will be done with *keysize* bytes of what the *ext* identifier points to, where *keysize* is the parameter that was used in the *mapCreate* call. The return value uniquely identifies this binding; it can later be given as an argument to *mapRemoveBinding*. The internal identifier should not have the value *XK_FAILURE*; this would conflict with the error return value for *mapResolve*. A return value of ERR_BIND indicates that the external identifier is already bound in the map.

Bind **mapBind** (*Map* Map, *void* *ext, *int* intern);

6.2.4 mapRemoveBinding

Removes binding *Bind* from map *Map*. Returns *XK_FAILURE* if the item is not in the map.

3.1 Compatibility note: Same as 3.1 *mapunbindbinding*.

xkern_return_t **mapRemoveBinding** (*Map* Map, *Bind* Bind);

6.2.5 mapUnbind

Removes binding of the association key *ext* from the map *Map*. This is the inverse of *mapBind*. Returns XK_FAILURE if the item is not in the map.

3.1 Compatibility note: Same as 3.1 mapremove.

xkern_return_t **mapUnbind** (*Map* Map, *void* *ext);

6.2.6 mapClose

Destroys the map and frees its space. Any elements left in the map will be unbound before the map is destroyed.

void **mapClose** (*Map* Map);

6.2.7 mapForEach

Allows iterative access to the entries of a map by the provided function *f*. Each call to *mapForEach* puts the key and its value into arguments passed the function *f*. The third argument passed to *f* is the supplied value *arg*. As long as *f* returns *TRUE* and there are unprocessed keys, *mapForEach* will continue to call *f*.

The iteration is guaranteed not to return the same key/value twice in one iteration, but it might not return all keys/values in the map if the map changes between calls to *mapForEach*. The programmer can avoid this situation by not having the function *f* block or give up control. This is sufficient because *x*-kernel threads are not preempted. The order in which keys are returned depends on the internal structure of the map.

void **mapForEach**(*Map* m, *MapForEachFun* f, *void* *arg)

typedef int **MapForEachFun**(*void* *key, *int* value, *void* *arg)

Example declaration: *int* **copyMapData**(*void* *key, *int* value, *void* *arg)

6.3 Usage Rules

6.3.1 External Keys

Maps are used to bind a variable length external id to an internal id of type ***int***. The size of the external id is given as an argument when a particular map is created. All external ids bound using this map are expected to be of this size. The user is warned to use a zero-izing routine like bzero before assigning values to a structure that will be used with the map routines. The C language can have uninitialized data in the interstices of structures (i.e., padding areas), and these can cause structures that are "equal" (i.e., all fields have the same values) fail to map to the same value in the *x*-kernel.

6.3.2 Active and Passive Maps

Protocols generally maintain two maps: an active map and a passive map. Active maps are used to map keys found in incoming messages into the session that will process the message. Thus, the active map holds information about the set of currently active connections. Passive maps are used to bind identifiers to ***Enable*** objects, thereby allowing a protocol to create a session when a message that is part of a new connection arrives.

A protocol's implementation of ***xDemux*** will generally check the active map for an existing session, going to the passive map only if no active session exists.

7 Thread Library

The *x*-kernel uses a "thread-per-message" model of computation, and provides primitives for synchronizing threads. The operations affect thread scheduling. Of these, only *semWait* and *xk_master_unlock* can cause the *x*-kernel to run a different thread than the current one.

Note that this section does not define any operations for creating or destroying threads. This is because *x*-kernel threads are created and destroyed implicitly: threads are created by the device driver (in the case of incoming messages), by the system call interface (in the case of outgoing messages), and by the event library (in the case of an event firing). Threads are destroyed when they return from the outer-most procedure.

7.1 Type Definitions

The only thread-related type that protocol programmers need be aware of is the type *Semaphore*. However, this type is defined by the underlying platform and is opaque to the protocol programmer.

7.2 Synchronization Operations

7.2.1 semInit

Initializes semaphore *s* with a count of *cnt*. Semaphores in the kernel are normally allocated statically (i.e., *Semaphore x*) and must be initialized (*semInit(&x, 1)*) before they are used.

void **semInit** (*Semaphore* *s, *int* cnt)

7.2.2 semWait

Increments the use count for the semaphore. The current thread will either acquire the semaphore *s* or give up control until a *semSignal* is done by another thread and the scheduler runs.

void **semWait** (*Semaphore* *s)

7.2.3 semSignal

The current thread decrements the use count for semaphore *s*. The current thread continues executing. Note that if multiple threads are blocked on the semaphore, there is no policy about which thread will be awakened by the *semSignal*.

void **semSignal** (*Semaphore* *s)

7.2.4 xk_master_lock

The current thread acquires the master *x*-kernel lock. This prevents other xkernel threads from running unless they have explicitly released the *x*-kernel lock with *xk_master_unlock*. Threads started by *evInit* acquire the master lock automatically when they start running.

void **xk_master_lock** (*void*)

7.2.5 xk_master_unlock

The current thread (which must hold the master lock) releases the master xkernel lock. Unless a call is explicitly documented otherwise, threads may not make *x*-kernel system or library calls without holding the master lock.

void **xk_master_unlock** (*void*)

7.3 Usage Rules

7.3.1 Scheduling and Preemption

The currently executing thread gives up control by either terminating, executing a *semWait* operation, or explicitly executing an *xk_master_unlock* operation. In other words, the *x*-kernel does not preempt threads; threads voluntarily give up control of the processor. However, because each protocol is assumed to be an independent component, protocols are written to assume that control may be given up when a higher or lower level protocol is invoked. Therefore, all protocol-protocol operations are considered to have the potential to cause a thread switch, and all data structures must be "secured" before calling such operations.

7.3.2 Blocking

Although the *x*-kernel advocates a "thread-per-message" model, and it provides primitives for blocking theads, as a general rule, threads should not block except when waiting for a reply in an RPC-like protocol. In most other cases, should a thread (message) not be able to proceed, it should put the message in a protocol-dependent queue and return. Later, another thread can pick the message up out of the queue and continue processing it.

For example, when an incoming thread/message arrives in IP and discovers that it is just one fragment of a larger datagram, rather than blocking the thread and wait for the other fragments to arrive, the thread should insert the fragment into a reassembly buffer and return. The thread that delivers the last fragment will then reassemble the fragments into a single datagram and continue.

7.3.3 External Threads

Where the *x*-kernel is embedded in another operating system, there may be asynchronous threads representing device drivers or user requests. In general, the semaphore interface is used to coordinate resource sharing between these threads and *x*-kernel threads. The resources, semaphores, and their uses are completely platform dependent and will be documented in separate appendices at a later time.

7.3.4 Thread Turnaround

Protocols should refrain from taking threads which are shepherding outgoing messages down the protocol stack and turning them around to accompany messages traveling up the protocol stack. Since protocols *are* allowed to reverse thread direction from incoming to outgoing, allowing turnaround from outgoing to incoming could lead to a thread caught in a recursive loop. If an outgoing thread needs to send a message back up, it should start a new thread to do this. The push routine of the ethernet protocol (/usr/xkernel/protocols/eth) has an example of how this is done.

8 Trace Library

The trace library provides a facility for tracing system events. Most importantly, the library allows conditional printing in *printf* format of statements taking from zero to six variables.

8.1 Type Definitions

The current value of the tracing variable *tracevar* is used to control whether or not a particular trace operation takes place. The trace variable values can be set at system build time; see section 13. The following defined constants are suggestive of how to use trace levels:

TR_NEVER	for debugging statements that are unused (noop)
TR_FULL_TRACE	every subroutine entry and exit
TR_DETAILED	all functions plus dumps of data structures at strategic points
TR_FUNCTIONAL_TRACE	all the functions of the module and their parameters
TR_MORE_EVENTS	even more detail on events
TR_EVENTS	more detail than major events
TR_SOFT_ERROR	mild warnings
TR_MAJOR_EVENTS	open, close, etc.
TR_GROSS_EVENTS	the coarsest tracing level
TR_ERRORS	serious non-fatal errors; some residual event traces
TR_ALWAYS	normally only used during protocol development

8.2 Operations

8.2.1 xTrace

The *xTrace*n macro takes n arguments in addition to the variables *tracevar*, *tracelevel*, and *formatstring*. *tracevar* is a name associated with the variable being traced. *tracelevel* is compared to the value of the trace variable to determine at runtime if the trace statement should be printed. *formatstring* is a *printf*-style formatting statement.

Each protocol has a trace variable based on the protocol name with "p" appended; e.g. udp has *traceudpp*. In addition to protocol tracing, there are *x*-kernel trace variables for subsystems: init, processswitch, protocol, processcreation, cksum, event, system, msg, ptbl, and sessn. Most of these are defined in file /usr/xkernel/include/debug.h.

Note that the trace facility automatically supplies a newline at the end of the trace message; the supplied format string need not. Also, note that there should be no whitespace preceding a trace variable name in any tracing statement. Whitespace will cause errors in the macro expansion and result in compilation errors.

xTrace*n*(*int* tracevar, *int* tracelevel, *char* *formatstring, args, ...)

For example:

```
int traceudpp;

   xTrace2(udpp, TR_ERRORS, "input port %d output port %d", inp, outp);
```

8.2.2 xIfTrace

If the *tracelevel* is less than or equal to the value of the *tracevar*, then execute the statement directly following.

xIfTrace (*int* tracevar, *int* tracelevel)

For example:

```
int traceudpp;

xIfTrace(udpp, TR_ERRORS)
        dump_header();
```

8.2.3 xAssert

If the expression *exp* evaluates to FALSE, the *x*-kernel will print a message and halt.

xAssert (*bool* exp)

8.2.4 xError

Non-fatal error conditions can print warnings even in nondebugging mode by using the *xError* call.

xError (*char* * ErrorString)

8.3 Usage Rules

If you are writing a new protocol, you should insert trace statements. Even though there will never be bugs left after you release your protocol, it may help other in debugging their protocol which sits on top of, or below, yours. Don't delete these very helpful debugging libraries when you are done.

9 Utility Routines

9.1 Storage: xMalloc and xFree

These are just like Unix malloc and free functions. The *xMalloc* function will cause an *x*-kernel abort if no storage is available; therefore, it has no error return value.

char *__xMalloc__(*int* size)

int __xFree__(*char* *buf)

9.2 Time: XTime and xGetTime

Uses the same time structure as Unix:

```
typedef struct {
        long sec;
        long usec;
} XTime ;
```

Set *t to the current time of day:

void __xGetTime__ (*XTime* *t)

9.3 Byte Order: ntohs, ntohl, ntons, and htonl

The byte order functions are the same as the Unix functions.

u_short __ntohs__(*u_short* n)
u_long __ntohl__(*u_long* n)
u_short __htons__(*u_short* n)
u_long __htonl__(*u_long* n)

9.4 Checksum

9.4.1 inCkSum

Calculate a 16-bit 1's complement checksum over the buffer **buf* (of length *len*) and the msg **m*, returning the bit complement of the sum. *len* should be even and the buffer must be aligned on a 16-bit boundary. *len* may be zero.

u_short __inCkSum__ (*Msg* *m, *u_short* *buf, *int* len);

9.4.2 ocsum

Return the 1's complement sum of the *count* 16-bit words pointed to by *hdr*, which must be aligned on a 16-bit boundary.

u_short __ocsum__ (*u_short* *hdr, *int* count);

10 Control Operations

Control operations are used to perform arbitrary operations on protocols and sessions. Control operations return an integer that indicates the length in bytes of the information which was written into the buffer, or -1 to indicate an error.

All implementations of control operations should check the length field before reading or writing the buffer, returning -1 if the buffer is too small. The checkLen(actualLength, expectedLength) macro can be used for this.

There are two "classes" of operations: standard ones that may be implemented by more than one protocol, and protocol-specific ones.

10.1 Standard Control Operations

10.1.1 Protocol and Session Operations

These operations can be performed on both sessions and protocols.

GETMYHOST, GETMYHOSTCOUNT

> When used on a protocol, GETMYHOST asks for all possible host addresses for the local host. When used on a session, GETMYHOST asks for the local host addresses actually being used on the connection. If the buffer is too small for all of the hosts, GETMYHOST will write as many hosts as the buffer allows (GETMYHOST with a buffer large enough to hold one host will return the most common or default host.) GETMYHOSTCOUNT asks for the number of hosts which will be returned by GETMYHOST.

GETMAXPACKET, GETOPTPACKET

> Treats the buffer as a pointer to an integer and sets it to the length of the longest message that the *XObj* can deliver (GETMAXPACKET) or the length of the longest message that can be delivered without fragmentation (GETOPTPACKET.) A protocol typically implements this operation by querying the lower *XObj*s and then subtracting its header length.
>
> Although GETMAXPACKET and GETOPTPACKET can be performed on protocols, it is preferable to use them on sessions, since different sessions of the same protocol may return different values.

RESOLVE, RRESOLVE

> These operations map high-level addresses into low-level addresses (RESOLVE) and vice versa (RRESOLVE.)

10.1.2 Session Operations

GETPEERHOST, GETPEERHOSTCOUNT

GETPEERHOST returns the host addresses of all peers of a session. It is an error to submit a buffer that is too small for all of the peer hosts, and -1 will be returned. GETPEERHOSTCOUNT asks for the number of hosts which will be returned by GETPEERHOST.

GETMYPROTO, GETPEERPROTO

Treats the buffer as a pointer to a long and sets it to the local or remote "protocol number" of the session. For example, UDP returns the local UDP port from a GETMYPROTO operation.

GETPARTICIPANTS

Treats the buffer as an array of (potentially uninitialized) participants. The session fills in the participants with data that was used to open the session. Sessions typically implement this control op by executing it on their lower session and then pushing their protocol-specific information on the participant stacks. By convention, the pointers placed on the participant stacks will be dynamically allocated. The caller of GETPARTICIPANTS is responsible for ultimately freeing the storage.

FREERESOURCES

Treats the buffer as a pointer to an *xmsg_handle_t*. This value is interpreted as the result of a previous *xPush* and frees the resources associated with that message.

10.2 Protocol-Specific Control Operations

Protocol-specific opcodes are defined relative to an identifier that has been assigned to each protocol. For example, the protocol ARP has been assigned the id ARP_CTL. Individual opcodes are then defined (in arp.h) as

```
#define ARP_INSTALL                 (ARP_CTL*MAXOPS + 0)
#define ARP_IPINTERFACES            (ARP_CTL*MAXOPS + 1)
#define ARP_IPADDRS                 (ARP_CTL*MAXOPS + 2)
```

This scheme is used to ensure that all control opcodes are unique. By convention, protocol-specific opcodes defined by protocol XYZ are prefixed with XYZ_. Also, until an identifier has been assigned to a protocol you are writing (i.e., upi.h has been edited), you can use a set of temporary ids: TMP0_CTL, TMP1_CTL, etc.

Protocol-specific control operations are described in the manual page for each protocol in appendix A.

10.3 Forwarding Control Operations

There are several situations where an *XObj* may not be prepared to handle a control operation. For example, a protocol-specific control operation may be sent through several intermediate *XObj*s in a graph before it reaches an *XObj* that understands the operation. Because of this, *XObj*s should be prepared to forward control operations which they don't understand or can't satisfy to their lower *XObj*s.

11 Reference Counting XObjs

Version 3.2 of the *x*-kernel maintains reference counts for sessions in order to facilitate their destruction. The following implementation notes should guide the protocol writer towards programming techniques that are consistent with the reference count mechanism.

In v3.1, the *XObj* structure had a reference count field which was supposed to keep track of the number of external references to that object. The intent was that the object could safely be destroyed when the reference count fell to zero.

Unfortunately, the proper use of reference counts was never documented. With each protocol writer deciding to handle session reference counts in a different manner (if at all), reference counts were seldom accurate. This resulted in idle sessions that were never freed and in dangling references to sessions which had been destroyed.

Constructing a protocol which directly manipulates reference counts and does so correctly can be awkward and tedious. Version 3.2 of the *x*-kernel has addressed this difficulty by:

- moving all direct reference count manipulation into the *x*-kernel infrastructure.
- changing the semantics of some of the UPI functions
- explicitly documenting how references to other *XObj*s may be used.
- adding system support for session caching and garbage collection.

The rest of this section discusses the system support for reference counts and explains how to modify a v3.1 protocol to properly use reference counts in v3.2.

We discuss *XObj* reference counts in the context of sessions (rather than protocols) because protocols are relatively static objects and the issues surrounding their reference counts are not very interesting.

11.1 References

An *XObj* reference is a pointer to an *XObj*. References come in two flavors: *permanent* and *temporary*.

A permanent reference can be used indefinitely. As long as the holder of a permanent reference does not call *xClose* on the reference, the holder knows that the pointer will remain valid. Permanent references are returned by *xOpen*.

An *XObj* pointer received as a parameter in a function call is a temporary reference. It can only be safely used for the duration of that function call.

A permanent reference may be created from a temporary reference by using the UPI function *xDuplicate*. For example, *xDemux* passes a reference to the lower session as a parameter to the upper protocol. If the upper protocol wishes to save this reference beyond the extent of the demux call, the protocol should call *xDuplicate* on the session. The protocol can then safely use the reference until it calls *xClose*.

11.2 Reference counts

Session reference counts in v3.2 are a sum of:

- the number of permanent external references to the session
- the number of outstanding *xPop*'s on the session

11.2.1 Counting External References

Several of the UPI functions are involved in maintaining the count of permanent external references. They are listed here along with their semantics with respect to reference counts (and how these semantics have changed from *x*-kernel version 3.1):

xCreateSessn(...)

> The newly created session has an initial reference count of 0 (it was 1 in v3.1.)

xOpen(...)

> The session returned by invoking the lower protocol's open routine has its reference count incremented before it is returned to the caller of *xOpen*, indicating that the caller now has a permanent reference to the session. (*xOpen* did not manipulate reference counts in v3.1.)

xClose(XObj s)

> Decrements the reference count of the session, indicating that a permanent reference to the session has been released. The lower protocol's close operation is called *only* if the session's new reference count is zero. (*xClose* always invoked the lower protocol's close operation in v3.1.)

xDuplicate(XObj s)

> By default, increments the session's reference count, indicating that a new permanent reference to the session has been created. If the session has its own duplicate function, that is called instead. (*xDuplicate* did not exist in v3.1.)

11.2.2 Counting xPops

To understand the second component of the reference count, the number of outstanding *xPop*s on a session, it is important to realize that a protocol does not have explicit references to its own sessions. That is, while a protocol usually maintains pointers to its sessions (in a map) and invokes operations on them, the sessions' reference counts do not take these pointers into account. (This is why *xCreateSessn* returns an object with a reference count of 0 and not 1.)

For a protocol to safely send this session pointer outside of the protocol (e.g., as a parameter in *xDemux*), it needs to turn the pointer into a reference by incrementing the session's reference count. It could do this by calling *xDuplicate* on the session as soon as it is extracted from the session map, making the external call, and then closing the session. But since all protocols must do this for incoming messages, this functionality has been absorbed into *xPop*. *xPop* increments the reference count of the session before calling the pop function and decrements it (possibly calling the session's close operation) afterwards.

It could be argued that all UPI functions, not just *xPop*, should indicate their use of a session by incrementing the reference count at their start and decrementing it at their completion. To perform any other UPI operation on a session, however, one must already have a reference to that session. As long as an *xClose* is not performed on that reference, the session is not going to go away; maintaining reference counts for these operations is not necessary. *xPop*s are different in that a session's reference count does not reflect that its protocol may send messages up through it.

Note that reference counts will not help a protocol which performs an *xClose* on a session reference while another thread has an outstanding operation on that same reference. To perform such a sequence is a protocol error. If two threads share a session reference, they should either synchronize to avoid such sequences or they should duplicate the reference with *xDuplicate* and each thread should *xClose* its reference when it is through.

11.3 Internal vs. External Reference Counts

Session reference counts are meant to count the number of *external* references to the session (i.e., references held outside of the protocol.) If a protocol must keep track of session references internal to the protocol itself, a separate mechanism must be used.

This requirement is driven by upper protocols which require that when they release all of their references to a lower session that the session's reference count goes to zero. Virtual protocols which manipulate lower session's upper protocol pointers are specific examples of these upper protocols.

11.4 Session Caching Strategies

In the presence of correct management of reference counts, protocols may want to implement some sort of session caching. For example, many of the sessions created on the receiving end of a remote procedure call will have their reference counts go to zero after the reply has been sent, and without caching, all of those sessions will be destroyed. While caching may not be important for all protocols, it is vital to the performance of protocols that experience a high frequency of traffic from the same sources.

If a protocol does cache idle sessions, it is important to make the caching transparent to upper protocols (i.e., an upper protocol should not be able to distinguish a newly created lower session from a cached lower session that is being reused.) One aspect of this requirement is that protocols which cache sessions must be sure to check for the presence of openEnables when reusing sessions which would otherwise be passively created and to call xOpenDone for when reusing such sessions from the cache.

Here are some examples of caching strategies and how a protocol might implement them. These are just examples – other strategies are certainly possible:

- Destroy the session if inactive.

  In this "null caching" strategy, the session's close routine destroys the session:

  ```
  foo_close(s)
          xAssert(s->rcnt == 0);
          xDestroy(s);
  ```

- Garbage collector on a per-session basis

  A garbage collector event is started for each idle session. When the event expires, the session is collected.

```
foo_demux()
        if ( active session does not exist ) {
                if ( openEnable exists ) {
                        if ( cached session exists ) {
                                evCancel(sstate->gc);
                                sstate->gc = 0;
                        } else {
                                s = foo_open();
                        }
                        xOpenDone(s, ...);
                } else {
                        drop packet; return;
                }
        }
        xPop()

foo_open()
        ...
        if ( active session does not exist ) {
                if (cached session exists ) {
                        evCancel(sstate->gc);
                        sstate->gc = 0;
                }
        }
        ...

foo_close(s)
        sstate->gc = evRegister(foo_gc, s);
        mark session as cached

foo_gc(s)
        xDestroy(s);
```

- Garbage collector on a per-protocol basis

  There is a single garbage collector for the protocol—it runs every so often and looks for idle sessions in the protocol's session map, calling a protocol-specific destroy function if it finds one.

  The *x*-kernel has a default session garbage collector which a protocol can use for this purpose. See xkernel/include/gc.h for the interface.

```
foo_init()
        ...
        initSessionCollector(cacheMap, interval, foo_destroy);

foo_demux()
        if ( active session does not exist ) {
                if ( openEnable exists ) {
                        if ( cached session exists ) {
                                move sessn from cached map to
                                active map
```

```
                        } else {
                                s = foo_open();
                        }
                        xOpenDone(s, ...);
                } else {
                        drop packet; return;
                }
        }
        xPop()

foo_open()
        ...
        if ( active session does not exist ) {
                if (cached session exists ) {
                        move sessn from cached map to active map
                }
        }
        ...

foo_close(s)
        xAssert(s->rcnt == 0);
        move sessn from active map to cached map

foo_destroy(s)
        xDestroy(s);
```

11.5 Modifying v3.1 protocols

In addition to the other modifications one needs to make to translate an existing protocol from v3.1 to v3.2, one should make the following modifications with respect to reference counts:

- foo_open should not increment the reference count

  v3.1 protocols often followed the format:

```
foo_open()
        if (session s exists) {
                s->rcnt++;
        } else {
                s = xCreateSessn();
                fill in state;
        }
        return s;
```

xOpen now automatically increments the reference count so the format should be more like:

```
foo_open()
        if (session s does not exist) {
                s = xCreateSessn();
                fill in state;
        }
        return s;
```

- `foo_close` should not decrement the reference count

  In v3.1, `foo_close` was called whenever *xClose* was called, so `foo_close` often looked like:

```
foo_close()
        if (--s->rcnt == 0) {
                xDestroy(s);
        }
```

  In v3.2, the protocol's close function is called only when the reference count reaches zero (see the examples in the caching section.)

- *xPop* should not be bypassed

  Since *xPop* in v3.1 did nothing more than call the session's pop operation, v3.1 protocols often either called their pop routine directly or, if their pop routine was small, they absorbed this functionality into their demux routine and bypassed *xPop* altogether. Since *xPop* manipulates reference counts in v3.2, it can not be bypassed.

- *xDemux* should not use *xOpen* to create new sessions

  It is common practice for a protocol to create passive sessions (where session creation is triggered by an incoming packet rather than an xOpen) by having `foo_demux` call `foo_open` and then possibly customizing the session. There is nothing wrong with this practice, but such a protocol should invoke `foo_open` directly rather than using *xOpen*. The passively created session should have an initial reference count of 0, but *xOpen* would return a session with an initial reference count of 1.

12 Protocol Tables

The *x*-kernel determines relative protocol numbers by reading protocol table files at boot time. Here is an example of a file defining a protocol table (see section 3.3 for an operation that queries this table):

```
#
# prottbl
#
# This file describes absolute protocol id's
# and gives relative protocol numbers for those
# protocols which use them
#
# x-kernel v3.2

eth         1
{
        ip          x0800
        arp         x0806
        rarp        x8035
        #
        # ethernet types x3*** are not reserved
        #
        blast       x3001
}
ip          2
{
        icmp        1
        tcp         6
        udp         17
        #
        # IP protocol numbers n, 91 < n < 255, are unassigned
        #
        blast       101
}
arp         3
rarp        4
udp         5
tcp         6
icmp        7
blast       8
sunrpc      9
```

Each protocol has an entry of the form:

```
name idNumber [ { hlp1 relNum1 hlp2 relNum2 ...  } ]
```

where the idNumber uniquely identifies each protocol.

There are two ways for a protocol to define its relative protocol number space. Using the first technique (e.g., ETH and IP), the protocol explicitly indicates which protocols may be configured above it and what their relative numbers are. Use of this numbering scheme is indicated by the presence of the optional hlp list after the protocol's idNumber.

With the second technique, the protocol does not explicitly name its allowed upper protocols, but will implicitly use each protocol's unique idNumber as its relative protocol number. BLAST, for example, employs this technique and would use SUNRPC's idNumber 9 as its relative protocol number above BLAST. Protocols using this second scheme use a 4-byte field in their headers for the relative protocol number.

The latter technique clearly allows more flexibility in how protocols may be composed. As flexible composability is one of the goals of the *x*-kernel, all new protocols written in the *x*-kernel should use this absolute numbering scheme.

All protocols must have an entry in the protocol table. If you are writing a new protocol, you will probably want to define an auxiliary protocol table which assigns a temporary idNumber to your new protocol and then configure your kernel to read in both the system table and your auxiliary table (see section 13.1 for configuration examples.) If you need to configure your new protocol above an existing protocol which uses explicit numbering, you can augment the table for the existing protocol as in this example:

```
#
# Auxiliary protocol table
#

yap 1000

ip 2  {
        yap 200
}
```

Here we define our new protocol YAP and indicate that it has protocol number 200 relative to IP. Note that when augmenting the tables of other protocols, the idNumber of the other protocol must match its number in the system file.

The *x*-kernel runs consistency checks on the protocol tables and will give error messages and abort in the presence of inconsistencies.

13 Configuring a Kernel

This section describes how to configure and build an *x*-kernel.

13.1 Specifying a Protocol Graph

A new instance of the *x*-kernel is configured by specifying the collection of protocols that are to be included in the kernel in a file called graph.comp. Example graph.comp files can be found in xkernel/build/Template.

The graph.comp file is divided into three sections: device drivers, protocols, and miscellaneous configuration parameters. The sections are separated by lines beginning with @; and each section may be empty.

The first two sections, device drivers and protocols, describe the protocol graph to be configured into your kernel. Device drivers and protocols are described by the same types of entries, as in the example:

```
name=yap files=yap,yap_clnt,yap_srvr dir=yap, protocols=ip,eth trace=TR_MAJOR_EVENTS;
```

The first field gives the protocol's name. The rest of the fields are optional and may occur in any order. The dir and files fields describe the names and locations of the source files used to build the protocol. Files are specified without extensions. An empty dir field will cause a search for the source files in the current build directory, a subdirectory in the current build directory with the protocol's name, and finally in the system protocol library. An empty files entry defaults to a single .c file with the protocols name. An entry for a protocol from the library should contain neither a dir nor a files entry.

The protocols field indicates the protocols below the current protocol in the graph. When this field contains multiple protocols, order is significant (the lower protocols will be loaded into the upper protocol's down vector in the order in which they are listed.) A protocol which expects multiple protocols below it will describe the expected semantics of the lower protocols in its manual page in appendix A.

The trace field defines the debugging level used in trace statements depending on the protocol variable traceyapp. See section 8 for the symbolic names associated with trace levels.

Multiple instantiations of protocols are supported by using a "/" character after the protocol name to add an extension. In the following example, two instantiations of "yap" are indicated, one over "ip" and one over "eth," and both are used by the "prt" protocol:

```
name=yaap/ip files=yap,yap_clnt,yap_srvr dir=yap,protocols=ip trace=TR_MAJOR_EVENTS;
name=yaap/eth protocols=eth;
name=prt files=prt dir=prt, protocols=yaap/ip,yaap/eth trace=TR_ERRORS;
```

Only the first of multiple instantiations should have dir, files, or trace fields.

The third section contains the names of files with protocol number tables that are to be loaded during initialization. It also contains the names of subsystems and their configuration parameters. Currently trace variables are the only parameters that can be set here. The following illustrates a typical use of the third section:

```
@;
#
# You can also specify auxiliary protocol tables to be read in at boot
# time.  DEFAULT loads the system protocol table file as listed in etc/site.h
#
prottbl=DEFAULT;
prottbl=./prottbl;

#subsystem tracing for messages and protocol operations
name=msg trace=TR_GROSS_EVENTS;
name=protocol trace=TR_MAJOR_EVENTS;
```

The graph.comp file is read by an *x*-kernel utility program called *compose* which generates startup code to build the protocol graph and set up the described configuration. The protocol graph is built bottom-up; when a protocol's initialization function is called, the lower level-protocols have already been initialized.

13.2 Build Procedure

A given instance of the kernel is built in a working directory in /usr/xkernel/build. It is a good idea to adopt a convention for naming the working directories. For example, each user that builds a kernel could be given a directory named for his or her login-id; e.g., /usr/xkernel/build/smith. Another possibility is to name each working directory after the key protocol configured into the kernel. For example, if a user is building a kernel to test a new protocol called YAP (yet another protocol), then the kernel might be built in /usr/xkernel/build/yap.

For the purpose of the following discussion, we assume the user is configuring a kernel so as to implement and evaluate protocol YAP. A user configuring a kernel that contains only existing protocols should ignore all references to YAP. The machine architecture in this example is MIPS; another supported architecture is the SunOS simulator ("sunos").

1. Put /usr/xkernel/bin/mips and /usr/xkernel/bin/ in your search path. These must be before /bin and /usr/bin. This will allow use of the version of make distributed with the *x*-kernel (GNU make v. 3.57) rather than the standard Unix make.

2. Create a working directory to hold your kernel: cd /usr/xkernel/build; mkdir yap

3. cd yap

4. Copy the Makefile from /usr/xkernel/build/Template/Makefile.mips into your configuration directory under the name Makefile:

   cp ../Template/Makefile.mips Makefile

5. Make the Makefile writable:

   chmod u+w Makefile

6. Edit the Makefile. Select an appropriate HOWTOCOMPILE. Generally, you want DEBUG until you are ready to run serious performance tests, then change it to OPTIMIZE. (See Section 5.)
7. make setup (generates errors if run more than once.) This creates object directories (if needed) for your platform and creates an example graph.comp in your build directory.
8. If you are including a new protocol in your kernel, then create (at least) a shell for that protocol; e.g., create a shell version of yap.c. The protocol shell should define an initialization routine; e.g., yap_open. If you are configuring a kernel that contains only existing protocols, then this step is not necessary.
9. Edit graph.comp to specify the desired protocols into your kernel.

   Finally, the following two steps must be redone if graph.comp is later modified:
10. make compose
11. make depend
12. make

13.3 Debug versus Optimized Mode

The *x*-kernel uses a trace package to generate debugging information. To enable the tracing facility, edit the Makefile to set HOWTOCOMPILE to DEBUG; tracing can be disabled by setting HOWTOCOMPILE to OPTIMIZE. Then type make. See also section 8.

If you are interested in accurate performance timings, you should set HOWTOCOMPILE to OPTIMIZE in the Makefile. This causes all trace of tracing code to be eliminated form the kernel.

14 Running a Kernel

This section describes the procedure for running a kernel in each platform.

14.1 SunOS4

How the *x*-kernel runs on top of SunOS depends on what "device drivers" are configured into the kernel. In the standard release, the ethernet driver is really a simulator that uses UDP to send packets between hosts. In this case, if you implement IP and UDP <u>within</u> the *x*-kernel, then the simulated UDP packets you send are, in turn, encapsulated in real UDP packets. This means that protocols and programs built on top of simulated UDP can only talk to their peers in other simulators; they cannot communicate with "real" versions of those protocols running stand-alone, for example, on a Unix machine.

An alternative is to write an ETH protocol that sends and receives real ethernet packets, perhaps by using the STREAM interface to the driver available in SunOS 4.1. Similarly, one could imagine writing a UNIXIP device driver that sends and receives real IP packets using a raw datagram socket. The rest of this section assumes the simulator version of ETH that runs on top of the UDP protocol provided by the SunOS kernel.

As the result of configuring a kernel (section 13), a file named xkernel should exist in your current directory (/usr/xkernel/build/yap). While in this directory, create a sub-directory for each host you want to simulate; e.g.,

```
mkdir client
mkdir server
```

In each of these directories, create a file named rom. This file contains information that the simulated host "reads in from ROM". The following are example rom files that might be contained in client and server, respectively.

```
eth 3051
arp 128.10.5.23 192.12.69.88 3051
arp 128.10.5.54 192.12.69.89 3050
```

```
eth 3050
arp 128.10.5.54 192.12.69.89 3050
arp 128.10.5.23 192.12.69.88 3051
```

The first field of each line indicates the protocol for which the information is relevant.

For the eth entry, the second field indicates the UDP port number on which this simulated host will receive network packets. A unique port number must be used for each simulated host running on any given real processor. Simulated hosts running on different processors can use the same port number. (In this example, the two simulated hosts run on two different processors <u>and</u> use different UDP port numbers: 3051 and 3050.)

For the arp entries, each line corresponds to a simulated IP host. The second field is the simulated IP host number, the third field is the actual IP host number where the simulator runs, and the third field is the simulator's UDP port number. The first arp entry must correspond to the local host. Note that the simulated IP host numbers do not necessarily correspond to the real IP address of the machine on which the simulated host is running.

Each simulated host runs as a separate Unix process. It is convenient if this process is started from a different window; i.e., one window per simulated host. For each simulated host, cd to the sub-directory that contains that host's rom file—e.g., cd client—and then type ../xkernel. To stop a simulated host, type DELETE or CTRL-C.

14.2 Mach 3

As the result of configuring a kernel (section 13), a file named xkernel should exist in your current directory (/usr/xkernel/build/yap). You may optionally create a file named rom in this directory. This file contains information that the host "reads in from ROM". This is an example rom file:

```
eth priority 200
arp 192.12.69.88 8:0:20:1:51:E6
arp 192.12.69.89 8:0:20:1:50:40
```

The first field of each line indicates the protocol for which the information is relevant.

Each ARP entry corresponds to an initial binding between the IP address and the ethernet address. These entries are not really necessary if you have hosts on your network which will respond to your RARP and ARP requests.

The ETH entry indicates the priority of the Mach packet filter used by the *x*-kernel. The default priority, 200, causes the *x*-kernel to grab ethernet packets and not allow the native Mach networking code to see them. This allows the *x*-kernel to fully communicate with other hosts, but makes it difficult to run *x*-kernel tests from a remote host.

By lowering the filter priority below 100, both Mach and the *x*-kernel see all ethernet packets. This is useful for allowing normal network activity while running *x*-kernel tests, but can result in troublesome interaction between the *x*-kernel protocols and the Mach protocols running on the same host. If you want to lower the filter priority, we recommend either

- Configuring your *x*-kernel without TCP, SUNRPC and any other protocols which might cause interaction problems

  or

- modifying your protocol table file to create an alternate protocol number for IP relative to ETH (the default is x0800.) In this case, of course, the *x*-kernel protocols will only be able to communicate with other hosts running the *x*-kernel with the althernate protocol table.

Since the *x*-kernel reads from the ethernet device, it needs to run with root uid. You will need to either su to root before running it or run it setuid root.

Since the *x*-kernel on the Mach platform does not respond well to CTRL-C's, we recommend running it in the background so you can easily send it a SIGKILL with the 'kill' command.

14.3 Running Test Suites

If you have configured protocols from the test directory into the kernel, you can run any subset of them by putting the test names onto the command line, e.g.:

```
xkernel -testip -testudp
```

With no command line selections, all of the configured test protocols will run.

Some of the tests run with a "client side" and a "server side". The server side needs no additional parameters, but the client side must be told the IP address of the server host test peer. This can be done with the "-c" command line option, e.g.:

```
xkernel -testip -testudp -c192.12.69.231
```

15 Releasing a Protocol

Once you have debugged and tuned a new protocol, you can make it available for others to use by creating a new directory in /usr/xkernel/protocols, copying your source files into that directory, creating a makefile in that directory, and modifying the makefile in /usr/xkernel/protocols. If your protocol is a device driver, put the source files in /usr/xkernel/PLATFORM/drivers instead of /usr/xkernel/protocols.

If you want to make your new protocol available to other sites, then make a tar file of your protocol available and drop a note to xkernel-bugs@cs.arizona.edu. We will include your protocol in the next release. You should also create a "man page" for you protocol similar to those found in Appendix A. The source files for these man pages are in /usr/xkernel/doc/manual/protocols.

16 System Installation

This section describes how to install Version 3.2 of the *x*-kernel. By installing the *x*-kernel we mean setting up the *x*-kernel environment so that programmers can begin writing protocols and configuring new kernels.

The *x*-kernel is commonly used for both research and instruction. A typical scenario is for a researcher (instructor) to install the *x*-kernel for use by various network projects (students). Individual projects (students) then build and test kernels in private working directories under the main *x*-kernel directory. This document describes the work the researcher (instructor) needs to do to set up the *x*-kernel directory. Most of the installation can be handled by the researcher (instructor). Only a few things need to be done by a system administrator that has root permission, and these are clearly documented below.

The *x*-kernel runs in either of two environments: as a task under Mach3 on a DecStation, or as a user process on top of SunOS, release 4.1 or latter.

16.1 Unpacking the *tar* file

If you haven't done so already, create a new directory for installing the *x*-kernel. For the purpose of this report, we refer to this directory as `/usr/xkernel`. You may chose another name, however. `cd` to this directory and type

`tar xf` *file*

where *file* is the disk or tape file of the *x*-kernel distribution. This will create several subdirectories in `/usr/xkernel`. We have found it useful to have all users working with the *x*-kernel access the *x*-kernel hierarchy via a shared group. For example, you may want to create an `xkernel` group and make the files and directories in `/usr/xkernel` owned by this group.

16.2 Directory Structure

The distribution contains the following directories:

`pi`
: Sources and binaries for platform independent code: the *x*-kernel's object-oriented infrastructure, protocol table code, and the message, participant, and map libraries.

`sunos`
: Sources and binaries for Unix-based *x*-kernel.

`mach3`
: Sources and binaries for the Mach3-based *x*-kernel.

`protocols`
: Sources and binaries for all protocols supported in this release.

`bin`
: Binaries for utility programs used to develop new kernels.

`etc`

Configuration-related data files and protocol tables.

`doc`

Documentation.

Note that those directories that contain *x*-kernel source and binaries actually place the binaries in several sub-directories, one for each "kind" of binary: `DEBUGsunos` (Sun binaries, compiled with debugging turned on), `OPTIMIZEsunos` (Sun binaries, compiled with optimization turned on), `DEBUGmach3` (Mach3 MIPS binaries, compiled with debugging turned on), `OPTIMIZEmach3` (Mach3 MIPS binaries, compiled with optimization turned on),

16.3 About Compilers

Version 3.2 of the *x*-kernel now uses either the Gnu C compiler (`gcc`) or the standard Unix compiler (`cc`). If you do not currently have Gnu C at your site, you can get a copy by doing anonymous FTP to `prep.ai.mit.edu`, and retrieving `pub/gnu/gcc.xtar`.

16.4 Building and Testing the System

Installing the *x*-kernel consists of building a few utility programs used to develop new kernels. These are the procedures for building these programs, though binary versions are distributed for both platforms and rebuilding these programs is probably unnecessary. For both of these procedures, you need to define the environment variable `ARCH` to be either `sparc` or `mips`. This will tell the installation process where to place your binaries.

The *x*-kernel depends on gnumake, so you first have to make gnumake:

`cd` to `/usr/xkernel/util/make` and type `make`.

Next, put `/usr/xkernel/bin/PLATFORM` and `/usr/xkernel/bin` in your search path, where PLATFORM is either `mips` or `sparc`. They should appear before `/bin` and `/usr/bin` in order to pick up gnumake before the standard Unix `make`. Finally, `cd /usr/xkernel/util` and type `make setup` followed by `make install`. This will install a handful of programs used to develop new kernels.

To test a system, build a kernel as described in section 13 which includes one or more of the test protocols (e.g., `udptest`). You should then be able to run your test kernel as described in section 14.

A Protocol Specifications

This appendix describes each of the protocols currently implemented in the *x*-kernel. The description for each protocol provides the following information.

NAME

Name of the protocol. This name, given in all lower-case letters, can be given as an argument to xGetProtlByName to get a capability for the protocol. Note that there are multiple implementations of various protocols; i.e., a given name might map to multiple implementations. The implementation bound to a name in a given kernel is set in `graph.comp`.

SPECIFICATION

Reference to a document that gives the specification for the protocol. In cases where no formal specification exists, this section gives a high-level description of the protocol.

SYNOPSIS

A brief description of what the protocol does. Outlines any unusual features and bugs, including any features of the protocol specification not implemented.

REALM

Indicates whether the protocol is in the ASYNC realm (supporting push, demux and pop), the RPC realm (supporting call, calldemux and callpop) or the CONTROL realm (existing only to allow control operations.)

PARTICIPANTS

A discussion of the number of participants the protocol expects to see and what it expects to see on the participants' stacks.

CONTROL OPERATIONS

Non-standard control operations supported by the protocol. For each control operation, the type of the input and output argument is given (i.e., the type used to interpret the buffer argument) and whether the control operation is understood by `x_controlsessn` or `controlprotl` (or both) is specified. In the case of control operations that take multiple arguments, we give a set of types. Non-primitive types are generally defined in the protocol's `.h` file.

EXTERNAL INTERFACE

Interfaces not encapsulated within *x*-kernel operations

CONFIGURATION

A description of configuration options for the protocol, including descriptions of of what this protocol expects of the protocols below it. If the protocol can only be configured above a certain protocol, the appropriate graph.comp line is given explicitly.

AUTHORS

Who to complain to when the protocol fails to work as advertized.

NAME

ARP (Address Resolution Protocol)

SPECIFICATION

D. Plummer. *An Ethernet Address Resolution Protocol*. Request for Comments 826, USC Information Sciences Institute, Marina del Ray, Calif., Nov. 1982.

SYNOPSIS

ARP translates IP addresses into ethernet addresses, and vice versa (i.e., it also implements RARP). This implementation of ARP supports a single interface, but may be multiply instantiated to support several network interfaces.

REALM

ARP is in the CONTROL realm. There are no ARP sessions – control operations may be performed on the protocol object only.

CONTROL OPERATIONS

`RESOLVE`: Maps an IP address into an ethernet address.

> Input: `IPhost`
>
> Output: `ETHhost`

`RRESOLVE`: Maps an ethernet address into an IP address.

> Input: `ETHhost`
>
> Output: `IPhost`

`ARP_INSTALL`: Installs an IP address to ETH address binding.

> Input: ArpBinding == {`ETHhost eth; IPhost ip;`}
>
> Output: none

`ARP_GETMYBINDING`: Return the IP and ETH address of the local host for the interface.

> Input: none
>
> Output: ArpBinding == {`ETHhost eth; IPhost ip;`}

CONFIGURATION

`name=arp protocols=eth;`

AUTHORS

Larry Peterson and Norm Hutchinson

NAME

BID (Boot ID Protocol)

SPECIFICATION

BID is the filtering module of the BootId protocol. The BootId protocol is designed to advise workstations that a peer has rebooted, to protect protocols from receiving messages generated during previous boot incarnations, and to inform higher protocols of a peer's reboot in a timely fashion.

If an upper protocol registers with BIDCTL protocol and messages from its session pass through BID sessions, the BootId protocol guarantees that a message from a rebooted peer will not be sent to an upper protocol until the upper protocol has been informed of the reboot.

SYNOPSIS

BID sessions stamp all outgoing messages with a local and remote BootId and filter out all incoming messages which do not have the correct BootIds. Determination of the "correct" BootId is made by the BIDCTL protocol. BID requires BIDCTL.

BID is not reliable. If there is confusion between two peers as to what their mutual BootIds are, messages between them will be silently dropped until the confusion is resolved.

REALM

BID is in the ASYNC realm.

PARTICIPANTS

BID expects an IPhost on the top of each participant. It examines this value but does not remove it form the participant stack before opening its transport protocol.

CONFIGURATION

BID expects to be configured above two protocols. The first is the transport protocol and the second is the BIDCTL protocol.

AUTHORS

Ed Menze

NAME

BIDCTL (Bootid Control Protocol)

SPECIFICATION

BIDCTL is the control module of the BootId protocol. The BootId protocol is designed to advise workstations that a peer has rebooted, to protect protocols from receiving messages generated during previous boot incarnations, and to inform higher protocols of a peer's reboot in a timely fashion.

If an upper protocol registers with BIDCTL protocol and messages from its session pass through BID sessions, the BootId protocol guarantees that a message from a rebooted peer will not be sent to an upper protocol until the upper protocol has been informed of the reboot.

SYNOPSIS

Upper protocols register their desire to be informed of a peer's reboot by openEnabling BIDCTL with that remote peer's IPhost. When BIDCTL determines that the remote peer has rebooted, it informs all interested upper protocols via a control operation (see below.) If an upper protocol is no longer interested in learning about a peer's reboot, it may openDisable BIDCTL.

REALM

BIDCTL is in the CONTROL realm. There are no BIDCTL sessions.

PARTICIPANTS

BIDCTL openEnable and openDisable expect a single participant containing the IPhost of the remote peer.

CONTROL OPERATIONS

`BIDCTL_PEER_REBOOTED`: Invoked *by* BIDCTL on registered upper protocols to inform them that a peer has rebooted. The *id* field of the BidctlBootMsg contains the new remote BootId. The upper protocol's control function should not block while handling this notification.

Input: `BidctlBootMsg == { IPhost h; BootId id; }`

Output: none

The remaining control operations are not necessary for most users of BIDCTL. They are provided mostly for the use of filtering protocols (e.g., BID) which work in conjunction with BIDCTL.

`BIDCTL_FIRST_CONTACT`: Invoked *by* BIDCTL on registered upper protocols to inform them that an initial bootid for the given peer has been discovered. The *id* field of the BidctlBootMsg contains the new remote BootId.

Input: `BidctlBootMsg`

Output: none

`BIDCTL_GET_LOCAL_BID`: Returns the current BootId of the local host.

> **Input:** none
>
> **Output:** `BootId`

`BIDCTL_GET_PEER_BID`: Returns the last confirmed BootId of the given remote host. If the *id* field of the input structure is non-zero, it indicates a possible new value. If this value differs from BIDCTL's confirmed value for that peer, BIDCTL will start a handshake with the remote peer to determine a new confirmed value. The input id can be used by a filtering protocol to indicate that it has seen a new remote BootId value.

> The BootId of the output structure will be 0 (an invalid BootId) if BIDCTL doesn't yet know the peer's BootId.
>
> **Input:** `BidctlBootMsg`
>
> **Output:** `BidctlBootMsg`

`BIDCTL_GET_PEER_BID_BLOCKING`: Differs from `BIDCTL_GET_PEER_BID` in that the calling thread will block if BIDCTL has not yet learned the peer's BootId or if the suggested *id* field is non-zero and differs from the protocol's current value for the peer BootId. If the operation blocks, it will not release the calling thread until the peer BootId has been confirmed. There is no timeout.

> **Input:** `BidctlBootMsg`
>
> **Output:** `BidctlBootMsg`

CONFIGURATION

BIDCTL expects only its transport protocol below it. It will open the transport protocol with a single participant consisting of the remote IP host.

BIDCTL uses an internal checksum and works correctly in the presence of dropped messages, so a reliable transport protocol is not necessary.

As an optimization, BIDCTL can perform an IP local-net broadcast to inform interested peers that it has rebooted. A rom file entry of the form:

```
bidctl bcast
```

will cause the broadcast and an entry of the form:

```
bidctl nobcast
```

will suppress it. Without a rom file entry, BIDCTL will perform the broadcast unless BIDCTL_NO_BOOT_BCAST is defined during compilation.

AUTHORS

Ed Menze

NAME

BLAST (RPC Blast Micro-Protocol)

SPECIFICATION

N. Hutchinson, L. Peterson, M. Abbott and S. O'Malley, RPC in the *x*-kernel: Evaluating New Design Techniques. In *Proceedings of the 12th Symposium on Operating System Principles*, December 1989, 91-101.

S. O'Malley and L. Peterson. *A New Methodology for Implementing Network Software*, University of Arizona Tech Report TR90-29, August 1990.

B. Welch. *The Sprite remote procedure call.* University of California at Berkeley, Tech Report UCB/CSD 86/302, June 1986.

SYNOPSIS

BLAST is a micro-protocol version of Sprite RPC's fragmentation algorithm. The algorithm was extracted from Sprite and made into a stand-alone protocol. BLAST takes a message of up to 16 Kbytes and fragments it into smaller packets and sends them. The receiver gathers all of the packets and sends a NACK if the last message arrives and there are missing fragments or if all fragments have not arrived in a given period of time. BLAST can handle any number on outstanding messages between two hosts (buffer space permitting, of course). The protocol is bi-directional; i.e., it supports blasts in both directions over the same session. Small messages take a short cut through the protocol and do not require the allocation of any resources. The sender keeps a copy of the message around until a timeout occurs or the higher level protocol that sent the message notifies BLAST that it can free the message. BLAST is not reliable but it is very persistent.

REALM

BLAST is in the ASYNC realm.

PARTICIPANTS

BLAST neither removes nor adds anything to the participant stacks.

CONTROL OPERATIONS

`BLAST_SETOUTSTANDINGMSGS`: Set the number of outstanding messages allowed (protocol only).

> Input: `int`
>
> Output: none

`BLAST_GETOUTSTANDINGMSGS`: Get the number of outstanding messages allowed (protocol only).

> Input: none
>
> Output: `int`

CONFIGURATION

BLAST requires only its lower transport protocol. Since BLAST doesn't use host addresses, it can sit on top of protocols using different address types without modification.

AUTHOR

Sean O'Malley and Ed Menze

NAME

CHAN (RPC Channel Micro-Protocol)

SPECIFICATION

N. Hutchinson, L. Peterson, M. Abbott and S. O'Malley, RPC in the *x*-kernel: Evaluating New Design Techniques. In *Proceedings of the 12th Symposium on Operating System Principles*, December 1989, 91-101.

S. O'Malley and L. Peterson. *A New Methodology for Implementing Network Software*, University of Arizona Tech Report TR90-29, August 1990.

B. Welch. *The Sprite remote procedure call.* University of California at Berkeley, Tech Report UCB/CSD 86/302, June 1986.

SYNOPSIS

CHAN is a single protocol version of Sprite RPC's reliable request-reply channel. The algorithm was extracted from Sprite and made into a stand-alone protocol. Each CHAN session supports the Birrel-Nelson implicit acking RPC algorithm between two hosts.

CHAN provides "at most once" RPC semantics. When a CHAN call returns successfully, the protocol guarantees that the request has been processed exactly once by the server. If CHAN returns unsuccessfully (XK_FAILURE), the server may have processed the request once, or it may not have seen the request at all.

Channel numbers are entirely internal to the CHAN protocol. When a new client channel session is created, a new host-host channel number is selected internally by CHAN. When protocols openEnable CHAN, they will receive connections from any channel number on any remote host. Each open of CHAN by a client session will result in the passive creation of a corresponding session on the server.

Each channel session will accept only a single outstanding request. Sending additional requests on a channel before the first request has returned is not allowed.

CHAN relies on the BIDCTL and BID protocols to determine when a peer has rebooted. When notified of a peer's reboot, CHAN will disable all active channels to that host. Outstanding calls will return XK_FAILURE, as will all subsequent calls on that channel session. Replies sent through disabled server channels will be discarded.

REALM

CHAN lies on the boundary between the ASYNC realm and the RPC realm. That is, it looks like an ASYNC protocol to protocols below it, and an RPC protocol to protocols above it.

CONTROL OPERATIONS

`CHAN_ABORT_CALL`: When invoked on a channel session, it causes the current call (if one is outstanding) to abort and return XK_FAILURE.

Input: `none`

Output: `none`

PARTICIPANTS

CHAN neither removes from nor adds to the participant stacks, passing the participants untouched to the transport protocol on an open and ignoring the participant structure on an openenable.

CONFIGURATION

CHAN requires its lower transport protocol configured as the first lower protocol and BIDCTL configured as the second lower protocol. CHAN requires that it's transport protocol will deliver incoming messages from different hosts through different lower sessions and that all CHAN messages from the same host come from the same lower session.

CHAN is a realm boundary protocol which assumes its transport protocol is symmetric (in the ASYNC realm.)

AUTHOR

Sean O'Malley

NAME

ETH (Ethernet Protocol)

SYNOPSIS

This hardware-independent protocol provides the interface between the rest of the *x*-kernel protocols and the actual ethernet drivers. It has a UPI interface to protocols above it and interacts with the drivers through a specialized UPI interface. There should be a separate instantiation of the ETH protocol for each driver protocol.

REALM

ETH is in the ASYNC realm.

PARTICIPANTS

ETH expects a single remote participant with an ETHhost pointer on the top of the stack. If the local participant is present it is ignored.

CONTROL OPERATIONS

`ETH_SETPROMISCUOUS`: Sets the corresponding device controller in promiscuous mode and deliver copies of all packets to this session. (session only)

Input: none

Output: none

EXTERNAL INTERFACE

Ethernet driver protocols should include the file protocols/eth/eth_i.h which defines the interface between ETH and the drivers.

ETH will openenable its driver protocol once at initialization time, without a participant list. This gives the driver protocol the XObj it should use in xDemux when it delivers messages.

ETH calls xPush with the driver *protocol* object (not a session) to send a message. ETH never opens the lower protocol.

ETH will attach a pointer to an ETHhdr as a message attribute for each outgoing message:

```
typedef struct {
    ETHhost     dst;
    ETHhost     src;
    u_short     type;
} ETHhdr;
```

ETH requires that the driver attach a message attribute pointing to an appropriate ETHhdr structure for every incoming message. For both incoming and outgoing messages, the ETHhdr type field will be in network byte order.

ETH requires the driver protocol to implement the control op GETMYHOST.

CONFIGURATION

Each instantiation of ETH should be configured above its corresponding driver protocol.

NAME

ETHDRV (Out-of-kernel Ethernet Driver Protocol (mach3 platform))

SPECIFICATION

ETHDRV is the Mach out-of-kernel ethernet driver, interacting with an ethernet device through a Mach device control port.

SYNOPSIS

Each instantiation of ETHDRV is associated with a single ethernet device. ETHDRV has the ability to block out certain packet types and to prevent other Mach servers from receiving ethernet packets.

REALM

ETHDRV is in the ASYNC realm, supporting the ethernet driver interface described in the ETH appendix.

PARTICIPANTS

ETHDRV supports the ethernet driver interface rather than a standard xkernel UPI interface and thus makes no use of participant stacks.

EXTERNAL INTERFACE

ETHDRV supports the ethernet driver interface described in the ETH appendix.

CONFIGURATION

ETHDRV requires no lower protocol. It's instantiation specification must be the name of the unmapped device as passed to the mach system call `device_open`. This name is case sensitive. E.g.,

`name=ethdrv/SE0;`

ETHDRV recognizes the following ROM options:

`ethdrv/xxx priority n`: Instantiation xxx of ETHDRV should use packet filter priority n. A priority ¿= 200 will not allow other servers to receive ethernet packets. Adjusting the priority in this way is a privileged Mach operation.

`ethdrv/xxx block n`: Instantiation xxx of ETHDRV should drop packets with ethernet type n. The type may be specified in either hexadecimal (prefixed with 'x') or decimal.

AUTHORS

Mats Bjorkman, Hilarie Orman, Michael Pagels, Ed Menze

NAME

ICMP (Internet Control Message Protocol)

SPECIFICATION

J. Postel. *Internet Protocol*. Request for Comments 792, USC Information Sciences Institute, Marina del Ray, Calif., Sept. 1981. ;

SYNOPSIS

ICMP handles control messages for IP. This implementation is complete in that it handles all possible incoming ICMP requests.

REALM

ICMP is in the CONTROL realm. ICMP sessions may be opened to allow control operations.

PARTICIPANTS

ICMP neither removes nor adds anything to the participant stacks. It passes the participants directly to IP.

CONTROL OPERATIONS

`ICMP_ECHO_REQ`: Send an ICMP Echo Request message to the peer host and wait for a reply. The buffer contains the length of the message. Returns 0 if successful, -1 if a timeout occurred. (session only)

Input: `int`

Output: none

CONFIGURATION

`name=icmp protocols=ip;`

AUTHOR

Clinton Jeffery

NAME

IP (Internet Protocol)

SPECIFICATION

J. Postel. *Internet Protocol.* Request for Comments 768, USC Information Sciences Institute, Marina del Ray, Calif., Aug. 1980.

SYNOPSIS

IP handles fragmentation and routing required in transmitting messages across heterogeneous interconnected networks. This implementation is complete, with the exception of some of the optional header fields.

REALM

IP is in the ASYNC realm.

PARTICIPANTS

IP removes a pointer to an IPhost from the top of the stack of each participant. If the local participant is missing or if the local IPhost pointer is ANY_HOST, IP will select an appropriate local IPhost.

CONTROL OPERATIONS

`IP_MYNET`: Return local host's IP network number. This is an IP address with the host component set to 0. (session only)

Input: none

Output: `IPhost`

`IP_REDIRECT`: Modifies routing table to use a specified gateway when delivering packets to a specified IP address. The first address is for the destination and the second is for the gateway. (session or protocol)

Input: `IPhost[2]`

Output: none

CONFIGURATION

IP must be configured above VNET:

```
name=ip protocols=vnet;
```

If an explicit route for a remote network is not specified, IP will forward packets for that network to a *default gateway.* The default gateway is initialized to IP_SITE_GTW from `xkernel/etc/site.h`, but this can be overridden by a rom file entry of the form:

```
ip    gateway     127.1.22.11
```

If the default gateway is the distinguished address `0.0.0.0` or can not be reached directly, IP will operate without a default gateway and ERR_XOBJ will be returned in cases where a default gateway would otherwise have been used.

AUTHOR

Clinton Jeffery, David Kays and Ed Menze

NAME

SELECT (RPC Select Micro-Protocol)

SPECIFICATION

N. Hutchinson, L. Peterson, M. Abbott and S. O'Malley, RPC in the *x*-kernel: Evaluating New Design Techniques. In *Proceedings of the 12th Symposium on Operating System Principles*, December 1989, 91-101.

S. O'Malley and L. Peterson. *A New Methodology for Implementing Network Software*, University of Arizona Tech Report TR90-29, August 1990.

B. Welch. *The Sprite remote procedure call.* University of California at Berkeley, Tech Report UCB/CSD 86/302, June 1986.

SYNOPSIS

SELECT is a micro-protocol that performs the addressing function of Sprite RPC; i.e., it demultiplexes request messages to the appropriate procedure.

REALM

SELECT is in the RPC realm.

PARTICIPANTS

SELECT removes a pointer to a long (the remote procedure number) from the top of the stack of the first participant.

CONFIGURATION

SELECT expects one RPC realm protocol below it.

AUTHOR

Sean O'Malley

NAME

SIMETH (Simulated Ethernet Driver Protocol (sunos platform))

SPECIFICATION

SIMETH simulates an *x*-kernel ethernet driver by sending and receiving messages using Unix UDP sockets.

SYNOPSIS

Each instantiation of SIMETH is associated with a specific Unix UDP port and simulates an ethernet driver for a single interface. SIMETH transmits outgoing messages by sending to other UDP ports and presents UDP messages received on its port as incoming ethernet packets. Note that since messages sent from one sunos *x*-kernel to another are encapsulated within Unix UDP packets, it is only possible to communicate with another peer running the *x*-kernel with this same driver. Communication with arbitrary peers is not possible with this driver.

The mapping between Unix UDP ports and SIMETH ethernet addresses is very simple. The six bytes of SIMETH ethernet address are formed by the concatenation of the four byte IP host number for the Unix host on which the simulator is running and the two byte UDP port used by the SIMETH instantiation. Note that this is the *real* IP host number, not the *simulated* IP host number. See the CONFIGURATION section below.

Note that an *x*-kernel may be configured with multiple instantiations of SIMETH, each with its own UDP port, to simulate a multihomed host.

SIMETH can awkwardly simulate ethernet broadcast messages. When an outgoing broadcast message is sent to SIMETH, SIMETH asks its corresponding ARP protocol for a dump of all hosts in its table. SIMETH then sends the message to each of these hosts in a point-to-point fashion. Note that for a reasonable simulation of ethernet broadcast, all *x*-kernels in communication should have the same ARP table (see the ARP appendix.)

REALM

SIMETH is in the ASYNC realm, supporting the ethernet driver interface described in the ETH appendix.

PARTICIPANTS

SIMETH supports the ethernet driver interface rather than a standard xkernel UPI interface and thus makes no use of participant stacks.

CONTROL OPERATIONS

`ETH_REGISTER_ARP`: Used by an ARP instantiation to register itself with its corresponding SIMETH driver. This is used to simulate ethernet broadcasts as described above. If no ARP protocol registers with a SIMETH instantiation, broadcasts on that instantiation will not be possible.

Input: `XObj /* ARP protocol object */`

Output: none

EXTERNAL INTERFACE

SIMETH supports the ethernet driver interface described in the ETH appendix.

CONFIGURATION

SIMETH requires no lower protocol. It can be configured in either the driver section or the protocol section of graph.comp.

SIMETH recognizes the following ROM options:

`simeth nnnn`: This instantiation of simeth should use UDP port nnnn. There must be such a line for each instantiation of SIMETH in the *x*-kernel.

AUTHORS

Larry Peterson and Norm Hutchinson

NAME

SUNRPC (Sun Remote Procedure Call Protocol)

SPECIFICATION

Remote Procedure Calls: Protocol Specification. Sun Microsystems, Inc., Mountain View, Calif., May 1988.

SYNOPSIS

Sun RPC is a fairly complete implementation of Sun's remote procedure call protocol. This implementation is compatible with Sun's native implementation. The interface to our implementation of Sun RPC in no way resembles Sun's. The Sun Portmapper is treated as a separate protocol that sits on top of Sun RPC. This implementation only supports UDP as a lower level protocol. The protocol supports unreliable remote procedure calls to possibly heterogeneous hosts. Our implementation of the Sun RPC protocol uses most but not all of the original Sun RPC include files and XDR files. Thus, you must have access to Sun include and XDR files to compile this protocol.

REALM

SUNRPC is in the RPC realm.

PARTICIPANTS

SUNRPC removes three pointers to longs from the remote participant stack, representing the procedure, version and program number (starting from the top of the stack.)

The local participant may be present, in which case it is passed on untouched to the next protocol, or it may be absent.

CONTROL OPERATIONS

`SUNRPC_CLNTGETERROR`: Returns an RPC error status structure. (session only)

Input: none

Output: `struct rpc_err`

`SUNRPC_CLNTSETTOUT`: Sets the client timeout value; the amount of time the client will wait before resending the message. (session only)

Input: `struct timeval`

Output: none

`SUNRPC_CLNTGETTOUT`: Returns the client timeout value. (session only)

Input: none

Output: `struct timeval`

`SUNRPC_CLNTSETWAIT`: Sets the client wait value, the amount of time the client will wait before returning an error. (session only)

Input: `struct timeval`

Output: none

`SUNRPC_CLNTGETWAIT`: Returns the client wait value. (session only)

Input: none

Output: `struct timeval`

`SUNRPC_GETCREDTYPE`: Returns the type of authentication credentials used. Valid types are NULL authentication, Unix authentication and DES authentication. Note that the protocol or user above Sun RPC is completely responsible for all authentication; the Sun RPC protocol merely copies credentials into and out of the header. (session only)

Input: none

Output: `int`

`SUNRPC_GETCRED`: Returns the authentication credentials used in an opaque byte string. (session only)

Input: none

Output: `char *`

`SUNRPC_SETCREDTYPE`: Sets the type of authentication credentials used (session only)

Input: `int`

output: none

`SUNRPC_SETCRED`: Sets the authentication credentials used in an opaque byte string (already xdr'ed). (session only)

Input: `char *`

Output: none

`SUNRPC_GETVERFTYPE`: Returns the type of authentication verification used. Valid types are NULL authentication, Unix authentication, and DES authentication. (session only)

Input: none

Output: `int`

`SUNRPC_GETVERF`: Returns the authentication verification used contained in an opaque byte string. (session only)

Input: none

Output: `char *`

`SUNRPC_SETVERFTYPE`: Sets the type of authentication verification used. (session only)

Input: `int`

Output: none

`SUNRPC_SETVERF`: Sets the authentication verification used contained in an opaque byte string (already xdr'ed) (session only)

Input: `char *`

Output: none

`SUNRPC_SVCAUTHERR`: Causes session to return authentication error message and terminate. (session only)

Input: `char *`

Output: none

`SUNRPC_SVCSYSTEMERR`: Causes session to return a system error message and terminate. (session only)

Input: `char *`

Output: none

CONFIGURATION

SUNRPC is a realm boundary protocol which assumes its lower protocol is symmetric (in the ASYNC realm.)

`name=sunrpc protocols=udp;`

AUTHOR

Sean O'Malley

NAME

Sun Portmapper

SPECIFICATION

? Sun Microsystems, Inc., Mountain View, Calif., May 1988.

SYNOPSIS

The portmapper supports all five of the procedures defined by Sun. This implementation is compatible with Sun's native implementation. The *x*-kernel portmapper is treated as a separate protocol that sits on top of the *x*-kernel implementation of Sun RPC. This implementation only supports UDP as a lower level protocol. Client and server protocols running above the sunrpc protocol make remote procedures calls to the portmapper, using its well-known UDP port number 111.

Our implementation of the Sun Portmapper protocol uses most, but not all of the original SUN_RPC include files and XDR files. You must have access to Sun include and XDR files to compile this protocol.

REALM

The portmapper is in the ASYNC realm.

PARTICIPANTS

???

CONTROL OPERATIONS

TBD

CONFIGURATION

The portmapper is a realm boundary protocol which assumes its single lower protocol is in the RPC realm.
`name=pmap protocols=sunrpc;`

AUTHOR

Sean O'Malley and Richard Schroeppel

NAME

TCP (Transmission Control Protocol)

SPECIFICATION

Transmission Control Protocol. Request for Comments 793, USC Information Sciences Institute, Marina Del Rey, Calif., Sept. 1981

SYNOPSIS

TCP is a reliable stream transport protocol. It maintains a connection between the server and the client, and provides reliable stream delivery to the process. This implementation is an encapsulation of the Unix 4.3 BSD implementation.

REALM

TCP is in the ASYNC realm.

PARTICIPANTS

TCP removes a pointer to a long (the TCP port number) from the participant stack. TCP ports must be less than 0x10000. If the local participant is missing, or if the local protocol number is ANY_PROT, TCP will select an unused local port.

CONTROL OPERATIONS

`TCP_PUSH`: Force a TCP message to be sent. (session only)

> Input: none
>
> Output: none

`TCP_GETSTATEINFO`: Returns state of the connection. (session only)

> Input: none
>
> Output: `int`

`TCP_DUMPSTATEINFO`: Prints out statistics gathered by TCP. (protocol only)

> Input: none
>
> Output: none

CONFIGURATION

```
name=tcp protocols=ip;
```

AUTHORS

Norm Hutchinson and Herman Rao

NAME

UDP (User Datagram Protocol)

SPECIFICATION

J. Postel. *User Datagram Protocol*. Request for Comments 768, USC Information Sciences Institute, Marina del Ray, Calif., Aug. 1980.

SYNOPSIS

UDP is a trivial protocol that dispatches messages that arrive at the host to a process running on the host.

REALM

UDP is in the ASYNC realm.

PARTICIPANTS

UDP removes a pointer to a long (the UDP port number) from the participant stack. UDP ports must be less than 0x10000. If the local participant is missing, or if the local protocol number is ANY_PROT, UDP will select an unused local port.

CONTROL OPERATIONS

`UDP_ENABLE_CHECKSUM`: Cause the session to use checksums on its outgoing packets.

> **Input:** none
>
> **Output:** none

`UDP_ENABLE_CHECKSUM`: Cause the session to not use checksums on its outgoing packets.

> **Input:** none
>
> **Output:** none

CONFIGURATION

```
name=udp protocols=ip;
```

AUTHORS

Larry Peterson and Sean O'Malley

NAME

VCACHE (Virtual Caching Protocol)

SPECIFICATION

VCACHE is a simple session caching protocol which provides a time buffer between when an upper protocol closes a lower session and when that close is propagated to the lower session.

SYNOPSIS

When a VCACHE session is closed, it is placed in a time-expiration cache. When VCACHE receives subsequent opens (active or passive), it first looks in its cache for sessions with the same hlp and the same remote host. After a configurable amount of time, closed VCACHE sessions will be destroyed and the lower session actually closed.

As an example, VCACHE can be inserted between a UDP server and UDP to prevent UDP sessions from being closed when the server releases all references.

Server sessions (those created as a result of a match between an incoming packet and an openenable) are always cached. If an instantiation of VCACHE is running in *symmetric* mode, client sessions (those created by an xOpen) will be cached as well. It is reasonable to configure VCACHE in symmetric mode above symmetric (ASYNC realm) protocols and in asymmetric mode above asymmetric (RPC realm) protocols, though this usage is certainly not required.

REALM

VCACHE can be used with either ASYNC or RPC realm lower protocols and assumes the realm of the protocol below it.

PARTICIPANTS

VCACHE passes participants directly to the lower protocol and does not make direct use of any participant information.

CONFIGURATION

VCACHE uses a single lower protocol.
VCACHE recognizes the following ROM options:

`vcache symmetric` : Run this instantiation in symmetric mode.

`vcache asymmetric` : Run this instantiation in asymmetric mode (default).

`vcache interval n` : Set the timeout interval for `n` seconds. A cached VCACHE session is eligible to be destroyed if it is idle for `n` seconds and is guaranteed to be destroyed if it is idle for more than `2 * n` seconds.

AUTHOR

Ed Menze

NAME

VCHAN (Channel Virtual-Protocol)

SPECIFICATION

N. Hutchinson, L. Peterson, M. Abbott and S. O'Malley, RPC in the *x*-kernel: Evaluating New Design Techniques. In *Proceedings of the 12th Symposium on Operating System Principles*, December 1989, 91-101.

S. O'Malley and L. Peterson. *A New Methodology for Implementing Network Software*, University of Arizona Tech Report TR90-29, August 1990.

B. Welch. *The Sprite remote procedure call.* University of California at Berkeley, Tech Report UCB/CSD 86/302, June 1986.

SYNOPSIS

VCHAN is a virtual protocol that multiplexes multiple client procedure invocations over some number of open channels. The call blocks if there are no idle channels available. VCHAN was originally based on the Sprite RPC protocol.

VCHAN initially opens a default number of channels for a new session, though this number can be increased or decreased via control operations.

REALM

VCHAN is in the RPC realm.

PARTICIPANTS

VCHAN expects an IPhost pointer on the stack of each participant. It will not remove this pointer before passing the address down to the lower protocol.

CONTROL OPERATIONS

`VCHAN_INCCONCURRENCY`: Increase the number of active channels by the number given (`xControlSessn` only).

Input: `int`

Output: none;

`VCHAN_DECCONCURRENCY`: Decrease the number of active channels by the number given (`xControlSessn` only).

Input: `int`

Output: none;

CONFIGURATION

VCHAN expects to be configured above another RPC realm protocol. It expects that each xOpen on the lower protocol with the same participants will return a new lower session.

AUTHOR

Ed Menze

NAME

VMUX (Virtual Muliplexing Protocol)

SPECIFICATION

VMUX sits above several lower protocols, making them appear as a single protocol. VMUX makes very simple decisions about which protocol to use for each connection.

SYNOPSIS

VMUX is only active during connection establishment. When openenabled, VMUX openenables all of its lower protocols on behalf of the upper protocol. When opened, VMUX attempts opens its lower protocols in the order in which they were specified in the down vector until a successful open occurs. The successfully opened lower session is then returned.

REALM

Since VMUX is active only during open, it should be considered in the same realm as the protocols below it (which should probably all be in the same realm.)

PARTICIPANTS

VMUX passes its participants to the lower protocols unmodified.

CONTROL OPERATIONS

VMUX forwards control operations to the last protocol in the down vector list.

CONFIGURATION

The lower protocols will be opened in the order in which they are listed in the configuration entry.

AUTHOR

Ed Menze

NAME

VNET (Virtual Network Protocol)

SPECIFICATION

VNET is a virtual protocol which manages multiple physical network protocols. When opened with an IP address, VNET determines if the host can be reached directly on one of its physical networks. If it can, a session on that network is opened. If it can not be directly reached, an ERR_XOBJ is returned.

SYNOPSIS

VNET sits above pairs of network protocols (one per interface) and ARP protocols. When opened with a remote IP address, VNET compares the net number with that of its lower protocols to determine if the host can be reached directly on a local network, opening the appropriate interface protocol (if possible.)

If opened with an IP broadcast address, VNET will determine which networks are matched by the broadcast address and will open a lower session on each of those networks. A push on a VNET broadcast session will result in a push on all of these lower network sessions.

Use of the IP broadcast address 255.255.255.255 will result in a VNET session which broadcasts on all of the local networks.

REALM

VNET is in the ASYNC realm.

PARTICIPANTS

VNET removes a pointer to an IPhost from the top of the stack of the remote participant. Only the remote participant is processed. New participants are created for opening the lower network protocols.

CONTROL OPERATIONS

`VNET_GETADDRCLASS`: Determines the address class of the given IP host. The address class is one of the following:

- `LOCAL_ADDR_C`: An address for the local host.
- `REMOTE_HOST_ADDR_C`: Remote host directly reachable on a local net.
- `REMOTE_NET_ADDR_C`: Remote host on a remote network.
- `BCAST_LOCAL_ADDR_C`: 255.255.255.255 – broadcast address for all local nets.
- `BCAST_NET_ADDR_C`: Broadcast address for a single network or a single subnet (if subnets are being used.)
- `BCAST_SUBNET_ADDR_C`: Broadcast for a network (more than a single subnet) when subnets are being used.

Input: VnetClassBuf == { `int class; IPhost host;` }

Output: VnetClassBuf

`VNET_GETNUMINTERFACES`: Indicate the number of interfaces used by the VNET protocol (protocol only.)

Input: none

Output: `int`

`VNET_HOSTONLOCALNET`: Indicates (through the xControl return value) whether the given host is on one of VNET's interfaces. When performed on a session, only those interfaces active on that session will be considered (a typical VNET session only uses one interface, though a broadcast session may have more than one.) When performed on a protocol, all interfaces are considered.

Returns sizeof(IPhost) if it is on a local network, 0 if it is not.

Input: `IPhost`

Output: none

`VNET_GETINTERFACEID`: Returns an opaque identifier indicating the interface used by this session. This identifier may be used in subsequent `VNET_DISABLEINTERFACE` and `VNET_ENABLEINTERFACE` calls. This operation will fail (and return 0) if performed on a broadcast session with more than one interface. Broadcast sessions never process incoming packets, however, so this operation will always succeed when performed on a session delivering incoming packets. (Session only.)

Input: none

Output: `VOID *`

`VNET_DISABLEINTERFACE`: The session will no longer send messages out over the interface corresponding to the given interface identifier. (Session only.)

Input: `VOID *`

Output: none

`VNET_ENABLEINTERFACE`: Undoes the effect of a previous `VNET_DISABLEINTERFACE` (Session only.)

Input: `VOID *`

Output: none

`VNET_ISMYADDR`: Indicates (through the xControl return value) whether the given host is an address which might be used to reach this host on VNET's local networks (i.e., if the address is one of this host's IP addresses or is a broadcast address.) Returns sizeof(IPhost) if it is local (or broadcast), 0 if it is not.

Input: `IPhost`

Output: none

CONFIGURATION

VNET expects its lower protocols to be configured in network/ARP pairs:

```
name=vnet protocols=eth/1,arp/1,eth/2,arp/2;
```

AUTHOR

Ed Menze

NAME

VSIZE (Size Virtual-Protocol)

SPECIFICATION

N. Hutchinson, L. Peterson, M. Abbott and S. O'Malley, RPC in the *x*-kernel: Evaluating New Design Techniques. In *Proceedings of the 12th Symposium on Operating System Principles*, December 1989, 91-101.

S. O'Malley and L. Peterson. *A New Methodology for Implementing Network Software*, University of Arizona Tech Report TR90-29, August 1990.

SYNOPSIS

VSIZE is a virtual protocol that selects a lower-level network protocol (e.g., BLAST vs. ETH) based on the size of the message being sent.

REALM

VSIZE is in the ASYNC realm.

PARTICIPANTS

VSIZE passes participants to the lower protocols without manipulating them.

CONTROL OPERATIONS

VSIZE forwards control operations to the "large message" protocol.

CONFIGURATION

VSIZE expects to be configured above two ASYNC protocols with the "short message" protocol first.

AUTHOR

Ed Menze