x-kernel Programmer's Manual
(Version 3.3)

Network Systems Research Group

June 1997

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 two different environments: (1) as a user program on top of Unix, and (2) as a network simulator on top of Unix. In both cases, the Unix platforms currently supported include Solaris, OSF/1 (Digital Unix), and Linux. (The distribution also includes source code for SunOS and Irix from earlier releases, but these platforms are not supported in the current release.) Protocols can be moved among the different environments without modification. This document assumes that the reader is generally familiar with the *x*-kernel's object-based infrastructure for implementing protocols.

Contents

1 Introduction

The *x*-kernel is an object-based protocol implementation framework It defines an interface that protocols use to invoke operations on one another (this is called the Uniform Protocol Interface, or UPI), and a collection of libraries for manipulating messages, participant addresses, events, associative memory tables (maps), and threads.

Version 3.3 represents a re-engineering of most of the *x*-kernel libraries, a consolidation of the platforms on which the *x*-kernel runs, and the addition of a protocol simulation platform. Like the previous version, Version 3.3 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. However, there are several minor differences between the Version 3.2 and 3.3 interfaces.

Sections 2 through 11 of this manual define the Uniform Protocol Interface and the libraries that make up the *x*-kernel. Sections 12–14 then describe the procedures for configuring and running the *x*-kernel, and for releasing protocols.

1.1 Other Sources of Information

This document is intended as a reference manual for a user that is already familiar with the *x*-kernel. There are several other sources of information that you should look at to learn more about the *x*-kernel.

First, the *x*-kernel was originally described in a pair of research papers [2, 5]. These are a good place to start to understand the motivation and design rationale behind the *x*-kernel.

Second, this Programmer's Manual, while thorough, is somewhat cryptic. It does not serve as a tutorial that teaches you how to write *x*-kernel protocols. For help in learning how to write *x*-kernel protocols, including examples from several existing protocols, see [8]. This tutorial borrows liberally from [7], which provides an even more comprehensive discussion of protocol design and implementation.

Third, if you have just picked up the *x*-kernel and want to try it out without having to first learn everything there is to know about it, then [6] is a good place to begin. Once you have a version of the *x*-kernel that builds and runs, it is much easier to start playing with the various features and options discussed in the Programmer's Manual.

Fourth, the *x*-kernel can now be run as a network simulator rather than on top of a real network. This simulator, called x-sim, provides a complete and realistic framework for developing, analyzing, and testing network protocols. Information about how to configure and use x-sim can be found in [1].

Finally, various components of the *x*-kernel are described in detail in a collection of design documents. In particular, [4] describes the implementation of the message library and [3] describes the implementation of the map library. Note that it is not necessary to understand how these components are implemented in order to write protocols; these reports are intended for advanced users that want to know more about how the *x*-kernel is implemented.

1.2 Acknowledgements

Many people at the University of Arizona and elsewhere have contributed to the *x*-kernel. They include Andy Bavier, Mats Bjorkman, Lawrence Brakmo, Peter Druschel, Norm Hutchinson, Hasnain Karampurwala, Ed Menze, Sandra Miller, David Mosberger-Tang, Erich Nahum, Sean O'Malley, Hilarie Orman, Larry Peterson, Rich Schroeppel, David Yates, and Andrey Yeatts. Many others have contributed protocols, as noted in the Appendix.

Our work with the *x*-kernel has been supported over the years by several different organizations and companies, including the National Science Foundation (through grants CCR-8811423, IRI-9015407, CCR-9102040, and NCR-9204393), the Advanced Research Projects Agency (through contracts DABT63-91-C-0030, DABT63-94-C-0002, and DABT63-95-C-0075), the National Computer Security Center (through University research grant MDA904-92-C-515), Sun Microsystems, and Digital Equipment Corporation, Intel, and Hewlett-Packard.

1.3 Our Address

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
PO BOX 210077
Tucson, AZ 85721-0077

We can be reached by electronic mail at:

xkernel-help@cs.arizona.edu

Because of limited resources we can't promise to fix every problem, but we appreciate all comments. Also, we typically post messages about the *x*-kernel (including notices of future releases) to

xkernel-interest@cs.arizona.edu

Send mail to

xkernel-interest-request@cs.arizona.edu

to be added to to this mailing list. Finally, we are on the Web at

http://www.cs.arizona.edu/xkernel/

1.4 Copyright Notice

2 Uniform Protocol Interface (UPI)

Each *x*-kernel protocol is encapsulated in a uniform protocol interface (UPI). 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 Protocol and Session Objects

The Protl and Sessn structures are the fundamental objects in the system. Most fields in the Protl and Sessn structures are not directly read or written by the programmer; those that are available to the programmer are so indicated in the comments.

```
typedef struct protl {
    char            *name;        /* the protocol name, e.g., "ethdrv" */
    char            *instName;    /* the instance name, e.g., "SE0" */
    char            *fullName;    /* the name given in graph.comp, e.g., "ethdrv/SE0" */
    char            *state;       /* readable/writable */
    Binding         binding;      /* readable/writable */
    int             id;
    int             *traceVar;    /* readable */

    /* pointers to protocols configured below this one */
    int             numdown;      /* readable - total number in down list */
    int             downlistsz;   /* size of downlist */
    struct protl    *down[8];     /* first 8 in down list */
    struct protl    **downlist;   /* overflow from down array */

    /* interface functions */
    XOpenFunc               open;
    XOpenEnableFunc         openenable;
    XOpenDisableFunc        opendisable;
    XOpenDisableAllFunc     opendisableall;
    XOpenDoneFunc           opendone;
    XCloseDoneFunc          closedone;
    XDemuxFunc              demux;
    XCallDemuxFunc          calldemux;
    XControlProtlFunc       controlprotl;
} *Protl;

typedef struct sessn {
    char            *state;       /* readable/writable */
    Binding         binding;      /* readable/writable */
    int             rcnt;
    unsigned char   idle;

    /* pointers to open sessions below this one */
    int             numdown;      /* readable - total number in down list */
    int             downlistsz;   /* size of downlist */
    struct sessn    *down[8];     /* first 8 in down list */
    struct sessn    **downlist;   /* overflow from down array */
```

```
    /* interface functions */
    XCloseFunc              close;
    XPopFunc                pop;
    XCallPopFunc            callpop;
    XPushFunc               push;
    XCallFunc               call;
    XControlSessnFunc       controlsessn;
    XGetParticipantsFunc    getparticipants;
    XDuplicateFunc          duplicate;

    /* pointers to protocols associated with this session */
    struct protl    *myprotl;    /* session is an instance of this protocol */
    struct protl    *up;         /* session was created by this protocol */
    struct protl    *hlpType;    /* session was created on behalf of this protocol */
} *Sessn;
```

If you think of the *x*-kernel as implementing protocol and session graphs, then each Protl represents a node in the protocol graph and each Sessn represents a node in the session graph. A protocol's down vector represents protocol graph edges; it contains pointers to the Protls that are below the protocol in the graph. The same is true for a session's down vector. The fields myprotl and up in the Sessn structure link a session to the protocols that own and created it, respectively.

For historical reasons, there are some fields in the actual Protl and Sessn structures that aren't shown in this document. These fields should not be used, as they will eventually be removed.

2.1.2 Enable Objects

Protocol writers use Enable objects to remember xOpenEnable calls. Typically, a protocol saves a pointer to an Enable object in its passive map, using mapBind. An Enable object has a field for reference counting. Calls to xOpenEnable with identical participants (the calls are redundant with respect to session creation) must be reference counted in order to properly handle xOpenDisable calls.

```
typedef struct xenable {
    Protl     hlp;         /* upper protocol */
    Protl     hlpType;     /* upper protocol */
    Binding   binding;     /* from mapBind */
    int       rcnt;        /* use count */
}Enable;
```

2.1.3 Return Values

Most routines have a return value type of XkReturn, which is either XK_SUCCESS or XK_FAILURE. Routines that return type Protl or Sessn have a failure value of ERR_PROTL or ERR_SESSN, respectively. Some message handling routines use type XkHandle (see Section 2.2.13). Severe error conditions will result in console error messages and the termination of the *x*-kernel.

2.1.4 Function Types

The following function typedefs are used in the Protl and Sessn structures.

```
typedef struct sessn *(*XOpenFunc)(Protl, Protl, Protl, Part *);
typedef XkReturn     (*XOpenEnableFunc)(Protl, Protl, Protl, Part *);
typedef XkReturn     (*XOpenDisableFunc)(Protl, Protl, Protl, Part *);
```

```
typedef XkReturn     (*XOpenDisableAllFunc)(Protl, Protl);
typedef XkReturn     (*XOpenDoneFunc)(Protl, Protl, Sessn, Protl);
typedef XkReturn     (*XCloseDoneFunc)(Sessn);
typedef XkReturn     (*XDemuxFunc)(Protl, Sessn, Msg *);
typedef XkReturn     (*XCallDemuxFunc)(Protl, Sessn, Msg *, Msg *);
typedef int          (*XControlProtlFunc)(Protl, int, char *, int);
typedef XkReturn     (*XCloseFunc)(Sessn);
typedef XkReturn     (*XPopFunc)(Sessn, Sessn, Msg *, void *);
typedef XkReturn     (*XCallPopFunc)(Sessn, Sessn, Msg *, void *, Msg *);
typedef XkHandle     (*XPushFunc)(Sessn, Msg *);
typedef XkReturn     (*XCallFunc)(Sessn, Msg *, Msg *);
typedef int          (*XControlSessnFunc)(Sessn, int, char *, int);
typedef Part         *(*XGetParticipantsFunc)(Sessn);
typedef XkReturn     (*XDuplicateFunc)(Sessn);
```

2.2 Protocol and Session Operations

This section defines the operations that protocols and sessions invoke on each other. In general, each of these 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 hlp to actively open a session associated with low-level protocol llp on behalf of high-level protocol hlpType. Typically, hlp and hlpType refer to the same protocol (see Section 2.5.2). The participants argument is a list of addresses for each participant in the communication. For this, and all calls returning type Protl or Sessn, a return value of ERR_PROTL or ERR_SESSN, respectively, indicates failure. This must be checked by all callers before using the return value.

Note that the high-level protocol will use its self object as the first (and usually second) argument in xOpen, and the lower-level protocol object as the third argument. The lower-level protocol'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 of the protocol-specific function.

Generic: Sessn xOpen(Protl hlp, Protl hlpType, Protl llp, Part *participants)

Specific: Sessn udp_open(Protl self, Protl hlp, Protl hlpType, Part *participants)

2.2.2 xOpenEnable

Used by high-level protocol hlp to passively open a session associated with low-level protocol llp on behalf of high-level protocol hlpType. As with xOpen, hlp 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. The participants argument is an ordered list of addresses of 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: XkReturn xOpenEnable(Protl hlp, Protl hlpType, Protl llp, Part *participants)

Specific: XkReturn udp_openenable(Protl self, Protl hlp, Protl hlpType, Part *participants)

2.2.3 xOpenDisable

Used by high-level protocol hlp to undo the effects of an earlier invocation of xOpenEnable. The hlp and hlpType arguments and the contents of the participants argument must be the same as the ones given to xOpenEnable.

Generic: XkReturn xOpenDisable(Protl hlp, Protl hlpType, Protl llp, Part *participants)

Specific: XkReturn udp_opendisable(Protl self, Protl hlp, Protl hlpType, Part *participants)

2.2.4 xOpenDisableAll

Used by high-level protocol hlp to inform low-level protocol llp that all previous openEnables made by hlp should be removed.

Generic: XkReturn xOpenDisableAll(Protl hlp, Protl llp)

Specific: XkReturn udp_opendisableall(Protl self, Protl hlp)

2.2.5 xOpenDone

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

Note that the hlpType argument is not required in the generic call because that value was saved in the Sessn object at the time of the xOpenEnable call.

Generic: XkReturn xOpenDone(Protl hlp, Protl llp, Sessn session)

Specific: XkReturn udp_opendone(Protl self, Protl llp, Sessn session, Protl hlpType)

2.2.6 xCloseDone

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

Generic: XkReturn xCloseDone(Sessn session)

Specific: XkReturn udp_closedone(Sessn self)

2.2.7 xDemux

Used by low-level session lls to pass message message to the high-level protocol that created it. The high-level protocol demux routine should find the appropriate session, creating it if necessary, and xPop the message to the session. See Section 2.3.2 for guidelines on when session creation is appropriate.

Generic: XkReturn xDemux(Protl hlp, Sessn lls, Msg *message)

Specific: XkReturn udp_demux(Protl self, Sessn lls, Msg *message)

2.2.8 xCallDemux

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

Generic: XkReturn xCallDemux(Protl hlp, Sessn lls, Msg *request, Msg *reply)

Specific: XkReturn udp_calldemux(Protl self, Sessn lls, Msg *request, Msg *reply)

2.2.9 xControlProtl

Used by one protocol to act upon another protcol (llp) for retrieving information or for setting processing parameters. The operation code opcode identifies the action; buffer is a character buffer from which an argument is retrieved and/or into which a result is placed; and length is the length of the buffer. Returns an integer that indicates the length in bytes of the information which was written into the buffer, or -1 to indicate an error. 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 11.

Generic: int xControlProtl(Protl llp, int opcode, char *buffer, int length)

Specific: int udp_controlprotl(Protl self, int opcode, char *buffer, intlength)

2.2.10 xClose

Decrements the reference count of a Sessn, calling the session's close function *only* if the reference count is zero.

Generic: XkReturn xClose(Sessn session)

Specific: XkReturn udp_close(Sessn self)

2.2.11 xPop

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

The hdr argument is passed directly to the protocol-specific routine. It is typically used to pass the header (which the demux routine used to find the session) to the session's pop routine.

Generic: XkReturn xPop(Sessn hls, Sessn lls, Msg *message, void *hdr)

Specific: XkReturn udp_pop(Sessn self, Sessn lls, Msg *message, void *hdr)

2.2.12 xCallPop

When a synchronous (RPC-like) protocol is demuxing a message to an asynchronous protocol, xCallPop can be used to allow the upper protocol to return a message. This reply message may be the same one passed to the synchronous protocol via xCallDemux.

Generic: XkReturn xCallPop(Sessn hls, Sessn lls, Msg *request, void *hdr, Msg *reply)

Specific: XkReturn udp_callpop(Sessn self, Sessn lls, Msg *request, void *hdr, Msg *reply)

2.2.13 xPush

Used by a high-level protocol that opened session lls to pass a message 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 xControlProtl and xControlSessn operations. The message handle may also take one of three special values: a return value of XMSG_NULL_HANDLE indicates a successful push to a protocol which does not generate handles, XMSG_ERR_HANDLE indicates general failure, and XMSG_ERR_WOULDBLOCK indicates that a session in non-blocking mode would normally have blocked the push.

Generic: XkHandle xPush(Sessn lls, Msg *message)

Specific: XkHandle udp_push(Sessn self, Msg *message)

2.2.14 xCall

Similar to xPush except that a reply message may be returned through the argument reply. 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 Section 3.2.1).

Generic: XkReturn xCall(Sessn lls, Msg *request, Msg *reply)

Specific: XkReturn udp_call(Sessn self, Msg *request, Msg *reply)

2.2.15 xControlSessn

Used by one session to act upon another session (lls) for retrieving information or for setting processing parameters. The operation code opcode identifies the action; buffer is a character buffer from which an argument is retrieved and/or into which a result is placed; and length is the length of the buffer. Returns an integer that indicates the length in bytes of the information which was written into the buffer, or -1 to indicate an error. There are two "classes" of operations: standard ones that may be implemented by sessions of more than one protocol, and protocol-specific ones. A full discussion of control operation codes is in Section 11.

Generic: int xControlSessn(Sessn lls, int opcode, char *buffer, int length)

Specific: int udp_controlsessn(Sessn self, int opcode, char *buffer, int length)

2.2.16 xGetParticipants

Used by one session to retrieve the participant list of another session, lls.

Generic: Part *xGetParticipants(Sessn lls)

Specific: Part *udp_getparticipants(Sessn self)

2.2.17 xDuplicate

Increments the reference count of session. This can be used to create a permanent handle on session from a temporary handle, or to create a new equivalent handle from an existing handle. For a full discussion of session reference counts, see the *x*-kernel Tutorial [8].

Generic: XkReturn xDuplicate(Sessn session)

Specific: XkReturn udp_duplicate(Sessn self)

2.3 Graph Manipulation Operations

Unlike the previous set of operations, which protocols and sessions 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 xCreateProtl

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

The use of xCreateProtl outside of initialization—for example, to dynamically load new protocols—is not supported at this time.

```
Protl xCreateProtl(ProtlInitFunc func, char *name, char *instName, int *traceVar,
                   int downc, Protl *downv)

typedef void (*ProtlInitFunc)(Protl self)
```

2.3.2 xCreateSessn

Called by protocol llp 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 protocols hlp, hlpType and llp. The new session's up pointer is set to hlp (this is where upward-bound messages through this session will be delivered). The count downc indicates how many lower level sessions this session will use. An array of lower sessions themselves is passed as downv. Sessions which use no lower sessions may pass zero for downc and NULL for downv. The initialization function pointer func may be null; otherwise this function should fill in the interface function pointers in the Sessn structure. These pointers are initialized to default (usually null) functions by the system initialization code.

```
Sessn xCreateSessn(SessnInitFunc func, Protl hlp, Protl hlpType, Protl llp,
                   int downc, Sessn *downv)

typedef void (*SessnInitFunc)(Sessn self)
```

2.3.3 xDestroySessn

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

```
XkReturn xDestroySessn(Sessn session)
```

2.3.4 xGetProtlByName

Returns a capability for (pointer to) a protocol object given its mnemonic name. See the discussion of graph.comp in Section 12.

```
Protl xGetProtlByName(char *name)
```

2.3.5 xSetSessnDown

Sets the indexth member of self's down vector to be session. It increments the Sessn field numdown as a side effect.

```
XkReturn xSetSessnDown(Sessn self, int index, Sessn session)
```

2.3.6 xGetProtlDown

Returns the indexth member of self's down vector. Returns ERR_PROTL if the index is larger than the down vector.

```
Protl xGetProtlDown(Protl self, int index)
```

2.3.7 xGetSessnDown

Returns the indexth member of self's down vector. Returns ERR_SESSN if the index is larger than the down vector.

```
Sessn xGetSessnDown(Sessn self, int index)
```

2.3.8 xMyProtl

Returns the myprotl pointer of self.

```
Protl xMyProtl(Sessn self)
```

2.3.9 xSetUp

Resets the up pointer of session to hlp. The up pointer of a session is initialized in xCreateSessn, so xSetUp is only used for extraordinary manipulation of the session graph.

```
void xSetUp(Sessn session, Protl hlp)
```

2.3.10 xGetUp

Returns the up pointer of session.

```
Protl xGetUp(Sessn session)
```

2.3.11 xHlpType

Returns the hlpType argument that was used to create session.

```
Protl xHlpType(Sessn session)
```

2.4 Utility Operations

2.4.1 xIsProtl

Returns true if object is a protocol; returns false if object was either never initialized or has been badly clobbered.

```
bool xIsProtl(Protl object)
```

2.4.2 xIsSessn

Returns true if object is a session; returns false if object was either never initialized or has been badly clobbered.

```
bool xIsSessn(Sessn object)
```

2.4.3 xIsValidProtl

A protocol created with xCreateProtl is kept in a system map and removed when the protocol is destroyed. xIsValidProtl can be used to determine whether a random Protl handle (protocol) is in this map and can thus be used safely.

```
bool xIsValidProtl(Protl protocol)
```

2.4.4 xIsValidSessn

A session created with xCreateSessn is kept in a system map and removed when the session is destroyed. xIsValidSessn can be used to determine whether a random Sessn handle (session) is in this map and can thus be used safely.

```
bool xIsValidSessn(Sessn session)
```

2.4.5 xPrintProtl

Displays some information about the state of protocol.

void xPrintProtl(Protl protocol)

2.4.6 xPrintSessn

Displays some information about the state of session.

void xPrintSessn(Sessn session)

2.5 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.

2.5.1 Initializing a Protocol

At system boot time, the *x*-kernel calls xCreateProtl for each protocol configured into the kernel (see Section 12). xCreateProtl, in turn, calls the protocol's init routine (where for a protocol named 'yap', this initialization routine must be named yap_init). The work generally done by this routine is illustrated by an example protocol in the *x*-kernel Tutorial [8].

2.5.2 hlp and hlpType

The operations xOpen, xOpenEnable, and xOpenDisable take two high-level protocols, hlp and hlpType. hlp is the protocol to which the new lower session should route incoming messages. The lower protocol uses hlpType to determine which messages the new session should handle. For example, when eth_open 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 Sections 4.3 and 12.3). The lower protocol passes hlp and hlpType down to xCreateSessn.

Most protocols use their self pointer as both hlp and hlpType when making these calls. Virtual protocols (see below) are the exception.

2.5.3 Protocol Realms

Although the *x*-kernel defines a single interface for all protocols, not all protocols are created equal. 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 into which a protocol falls defines both 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 protocols' 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 symmetric, and we strongly encourage such designs. Knowing that any low-level protocols you may use are symmetric enhances your 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.

2.5.4 Anchor Protocols

Anchor protocols sit either at the top or the bottom of a protocol stack and provide an interface between the *x*-kernel and the system in which the *x*-kernel is embedded. Top-level anchor protocols look like an *x*-kernel protocol from the bottom, but provide an Application Programmer Interface to the *x*-kernel. Bottom-level anchor protocols (e.g., device drivers) look like a protocol from the top, but typically interface with the lower levels of the surrounding system or with network hardware.

Writing anchor protocols involves careful synchronization of external threads with *x*-kernel threads and objects (see Section 7.4.3).

2.5.5 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 hlp. 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).

2.6 Default Operations

Since many protocols' UPI operations look very similar, the *x*-kernel provides some library operations that do much of the standard work of some of the operations. Many protocols can call these default operations, or at the very least, these default routines can serve as a template for writing the corresponding protocol-specific routine.

2.6.1 defaultOpenEnable

Binds key to an Enable object with hlp and hlpType. If a previous binding exists for the given key and protocols, the reference count of that Enable object will be increased. defaultOpenEnable will fail if a previous binding exists for this key that does not match the protocols.

```
XkReturn defaultOpenEnable(Map map, Protl hlp, Protl hlpType, void *key)
```

2.6.2 defaultOpenDisable

Undoes the effect of a previous defaultOpenEnable. Returns failure if no appropriate Enable object exists (e.g., if nothing exists for the given key or if the protocols don't match the saved values in the Enable object).

```
XkReturn defaultOpenDisable(Map map, Protl hlp, Protl hlpType, void *key)
```

2.6.3 defaultOpenDisableAll

Removes all Enable objects bound in map with protocol hlp. If func is non-zero, it is called with the (key, Enable *) pair for each Enable object in the map before it is removed.

```
XkReturn defaultOpenDisableAll(Map map, Protl hlp, DisableAllFunc func)
typedef void (*DisableAllFunc)(void *key, Enable *e)
```

2.6.4 defaultVirtualOpenEnable

Designed to be used by virtual protocols. In addition to the binding performed by defaultOpenEnable, an xOpenEnable is performed on each lower protocol in the null-terminated array llp (using participants), causing the lower protocols to deliver packets for hlpType to the virtual protocol self. If any of these xOpenEnables fail, defaultVirtualOpenEnable backs out of the entire operation. Assumes that the passive map is keyed on hlpType.

```
XkReturn defaultVirtualOpenEnable(Protl self, Map map, Protl hlp, Protl hlpType, Protl *llp,
                                  Part *participants)
```

2.6.5 defaultVirtualOpenDisable

Undoes the effect of a previous defaultVirtualOpenEnable.

```
XkReturn defaultVirtualOpenDisable(Protl self, Map map, Protl hlp, Protl hlpType, Protl *llp,
                                   Part *participants)
```

2.6.6 Usage

Figure 1 illustrates an example of how a protocol might simplify its enable/disable routines using these operations.

Figure 1: Using default routines

```
static XkReturn
yapOpenEnable(Protl self, Protl hlp, Protl hlpType, Part *p)
{
    long   key;
    PState *ps = self->state;

    key = getRelProtNum(hlpType, self));
    return defaultOpenEnable(ps->passive_map, hlp, hlpType, &key);
}

static XkReturn
yapOpenDisable(Protl self, Protl hlp, Protl hlpType, Part *p)
{
    long   key;
    PState *ps = self->state;

    key = getRelProtNum(hlpType, self));
    return defaultOpenDisable(ps->passive_map, hlp, hlpType, &key);
}

static XkReturn
yapOpenDisableAll(Protl self, Protl hlp)
{
    PState *ps = self->state;

    return defaultOpenDisableAll(ps->passive_map, hlp, NULL);
}
```

3 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.

3.1 Type Definitions

Messages, which are the *x*-kernel's abstract data type for network packets, are defined by the Msg structure. Loosely speaking, this structure consists of a tree of buffers that collectively hold the bytes contained in the message. The leftmost buffer in this tree is called the *header stack* because it holds the headers that are pushed onto the front of a packet. This data structure is fairly complex, however, and so we do not describe it in this document. The interested reader is referred to a companion report [4]. In addition, there is a MsgWalk structure that is used by msgWalkNext to traverse the buffers that make up a message (see Section 3.3.12). This structure is also defined in [4]. The fields of neither structure should not be directly accessed by the protocol developer.

3.2 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.

Messages that are newly created "own" the header stack, and can write into that space efficiently using msgPush. See Section 3.4 for more information about message stacks.

3.2.1 msgConstructEmpty

Initializes a message structure with a data length set to zero. The user must provide a pointer to valid memory.

```
void msgConstructEmpty(Msg *message)
```

3.2.2 msgConstructBuffer

Copies data from a user buffer (buffer) into an uninitialized message structure. The message data area, of size length, is allocated and a copy is performed. This constructor is used when the data buffer already exists. Use msgConstructAllocate when you will not have the opportunity to fill the buffer until after it has been created.

```
void msgConstructBuffer(Msg *message, void *buffer, int length)
```

3.2.3 msgConstructAllocate

Allocates a data area of size length and associates the area with the uninitialized message structure message. A pointer to the data area is returned. A device driver might use this constructor, handing the pointer to the device as a place to put down an incoming packet.

```
char *msgConstructAllocate(Msg *message, int length)
```

3.2.4 msgConstructCopy

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

```
void msgConstructCopy(Msg *message, Msg *original_msg)
```

3.2.5 msgConstructInplace

An uninitialized message structure is constructed with a direct reference to the buffer specified. A function appropriate for freeing the buffer when the message is destroyed must be provided. The msgConstructInplace function is recommended only for limited use, such as within device drivers.

```
void msgConstructInplace(Msg *message, char *buffer, int length, MsgCIPFreeFunc freefunc)
typedef void *MsgCIPFreeFunc)(void *, int);
```

3.2.6 msgDestroy

Logically frees message. Data portions of the deallocated message are freed if there are no other outstanding references to them.

```
void msgDestroy(Msg *message)
```

3.2.7 msgRefresh

Allocates a data area of size length and associates the area with the initialized message structure message. This is equivalent to (but can be faster than) doing a msgDestroy to message, followed by a msgConstructAllocate. This function should be used only when message is valid.

```
char *msgRefresh(Msg *message, int length)
```

3.2.8 msgAssign

The assignment of msg_2 to msg_1 means that msg_1 will refer to the same data that msg_2 currently points to. No data copying is involved. This is equivalent to doing a msgDestroy to msg_1, followed by a msgConstructCopy. This function should be used only when both messages are valid; copying to an uninitialized structure should be done with msgConstructCopy.

```
void msgAssign(Msg *msg_1, Msg *msg_2)
```

3.3 Manipulation Operations

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

3.3.1 msgLength

Returns the number of bytes of data in message.

```
int msgLength(Msg *message)
```

3.3.2 msgTruncate

Truncates the data in message to the given length. An attempt 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 *message, int newLength)
```

3.3.3 msgBreak

Removes length bytes from the front of original_msg and assigns them to fragment_msg. 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.

```
void msgBreak(Msg *original_msg, Msg *fragment_msg, int length)
```

3.3.4 msgJoin

Assigns (in the same sense as msgAssign) to new_msg the concatenation of message fragment1 to the front of fragment2. 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. One common use of msgJoin is to attach a fragment to the end of a larger message, in which case the first two arguments are the same (the larger message) and the third argument is the fragment.

```
void msgJoin(Msg *new_msg, Msg *fragment1, Msg *fragment2)
```

3.3.5 msgPush

Used to prepend space for a header to the front of a message. Returns a pointer to contiguous buffer of length bytes that is logically attached to the front of message. Typically, a header is then copied into this buffer.

```
char *msgPush(Msg *message, int length)
```

3.3.6 msgPop

Used to remove a header from the front of a message. Returns a pointer to a contiguous buffer of length bytes that contains the data that was at the front of message and removes length bytes from the front of the message.

```
char *msgPop(Msg *message, int length)
```

3.3.7 msgPeek

Used to examine a header at the front of a message. Returns a pointer to a contiguous buffer of length bytes that contains the data at the front of message. The message remains unchanged.

```
char *msgPeek(Msg *message, int length)
```

3.3.8 msgDiscard

Used to remove and discard a header of length length from the front of a message. msgDiscard is faster than msgPop since it doesn't have to worry about making the returned buffer contiguous.

```
void msgDiscard(Msg *message, int length)
```

3.3.9 msgSetAttr

Associates an attribute of length bytes with name and attaches it to message message. Setting an attribute overrides any previous attribute with the same name. Message attributes are used to communicate ancillary properties of messages from a protocol to a session, or between protocols.

The only name supported at this time is 0. Attempting to set an attribute with another name will result in an XK_FAILURE return value.

```
XkReturn msgSetAttr(Msg *message, int name, void *attribute, int length)
```

3.3.10 msgGetAttr

Retrieves an attribute previously attached to message message with name. If no attribute has been associated with name, 0 will be returned.

```
void *msgGetAttr(Msg *message, int name)
```

3.3.11 msgWalkInit

Intializes the context cxt for message, as required by msgWalk.

```
void msgWalkInit(MsgWalk *cxt, Msg *message)
```

3.3.12 msgWalkNext

Walks the tree structure of buffers that hold the message data, and returns a pointer to the next chunk of data in the message. Also sets length to the number of bytes in that chunk. Argument cxt maintains the context for the message traversal, so that msgWalk knows how far through the tree it got on the last invocation.

```
char *msgWalkNext(MsgWalk *cxt, int *length)
```

3.3.13 msgWalkDone

Destroys the context cxt used by msgWalkNext.

```
void msgCleanUp(Msg *message)
```

3.3.14 msgCleanUp

Frees unnecessary resources allocated to message.

```
void msgCleanUp(Msg *message)
```

3.3.15 msgShow

Shows information about message. Only valid when compiling in DEBUG mode.

```
void msgShow(Msg *message)
```

3.3.16 msgStats

Prints statistics about message. Only valid when compiling with OPTION_MSG_STATISTICS defined.

```
void msgStats(MsgWalk *message)
```

3.4 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.

When a protocol passes a message to an adjacent protocol (via xPush, xDemux, etc.) its view of the message becomes invalid. The contents of the message after such an operation 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 either the msgAssign or the msgConstructCopy operation before passing the message on to another protocol.

Note that although a protocol which constructs a message invalidates its view of the message by performing a UPI operation involving that message, it is still responsible for destroying the message.

The stack ownership is a hidden variable in the message library implementation that affects whether or not storage is automatically allocated on msgPush operations. The stack ownership is affected by several message library operations, particularly msgAssign, msgJoin, msgPeek, and msgConstructCopy. The user is referred to the source code for the details of the ownership rules.

Message attributes passed between protocols should consist of exportable data, i.e., not pointers. Adherence to this convention will ensure that the protocol can be in used in a multi-address space environment.

4 Participant Library

Participant lists identify members of a session and are used for opening connections. An upper protocol interested in establishing a connection constructs a participant list and passes it to the lower protocol as a parameter of an open routine. The lower protocol then extracts 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 stack of void * pointers to address components and the lengths of each component. The component pointers are pushed or popped onto the stack by utility functions.

4.1 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 {
    struct {
        void  *ptr;
        int   len;
    } arr[PART_MAX_STACK];
    int       top;
} PartStack;

typedef struct {
    int       len;
    PartStack stack;
} Part;
```

4.2 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.

4.2.1 partInit

Initialize participant list participants of number entries.

```
void partInit(Part *participants, int number)
```

4.2.2 partPush

Pushes address addr, pointing to length bytes, onto the stack of participant. A length of 0 indicates a “special-value” pointer (e.g., ANY_HOST) whose value as a pointer should be preserved across protection boundaries (and not dereferenced). See Section 4.4.

```
void partPush(Part participant, void *addr, int length)
```

4.2.3 partPop

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

```
void *partPop(Part participant)
```

4.2.4 partStackTopByteLen

Returns the number of bytes pointed to by the top element of the stack of participants. Returns 0 if the stack element was pushed with a length field of zero (i.e., a "special-value" pointer). Returns -1 if there are no elements on the stack.

```
int partStackTopByteLen(Part participants)
```

4.2.5 partLength

Returns the number of entries in participant list participants.

```
int partLength(Part *participants)
```

4.3 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.

The *x*-kernel reconciles these two approaches by maintaining a table of relative protocol numbers. (See Section 12.3 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.

```
ProtId relProtNum(ProtI hlp, ProtI llp)
```

This operation returns the protocol number of the high-level protocol relative to the low-level protocol, or -1 if no such binding has been configured in the protocol tables. 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.

Two other operations provide an alternate query interface. The operation

```
ProtId protTblGetId(char *protocolName)
```

returns the protocol ID number for the named protocol. This ID number can be used with

```
ProtId relProtNumById(ProtId hlpId, ProtI llp)
```

which has the same semantics as relProtNum, except that the high-level protocol is identified by its ID number rather than by the ProtI object itself. This interface can be useful when you need to determine relative protocol numbers, but do not have the appropriate ProtI objects in scope.

4.4 Usage Rules

By convention, active participant lists (those used in xOpen) 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.

Figure 2: Using the participant list

```
/* protocol invoking xOpen on low-level protocol llp */
{
    Part p[2];

    ...
    /* set participant addresses before calling low-level protocol's open */
    partInit(p, 2);
    partPush(p[0], &ServerHostAddr, sizeof(IPhost));  /* remote */
    partPush(p[0], &ServerPort, sizeof(long));         /* remote */
    partPush(p[1], ANY_HOST, 0);                       /* local  */
    partPush(p[1], &ClientPort, sizeof(long));         /* local  */
    xOpen(self, self, llp, p);
    ...
}

llp_open(Protl self, Protl hlp, Protl hlpType, Part *p)
{
    /* get participant addresses within low-level protocol's open */
    remoteport = (long *)partPop(p[0]);
    localport  = (long *)partPop(p[1]);
    ...
}
```

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, the constant pointers ANY_HOST and ANY_PORT can be used to specify "wildcard" values. For example, if you want to open UDP with a specific local port, but don't care which local host number is used, you 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. Similarly, the pointer ANY_PORT could be used for protocols that use ports on their stacks. Protocols that support wildcards indicate this in their manual page.

Figure 2 illustrates how a protocol that is about to invoke xOpen on a low-level protocol initializes the participant list, and then how the low-level protocol extracts that information from the participant list.

Notice in this example how the high-level protocol pushes two items (a host address and a port number) onto each participant's address stack, but the low-level protocol pops off only one item. This is because the low-level protocol does not interpret the first item (the host address); it just passes it on to its low-level protocol. Also note that when using participants that have been passed from other protocols, you 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.

In general, passive participant lists (those used in xOpenEnable) contain only the local participant, with no remote participant specified. This indicates that an upper protocol is willing to accept connections from any remote participant, as long as the connection is addressed to the correct local participant. Protocols which provide different semantics for their openEnable participants will indicate this explicitly in their manual page in the Appendix.

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. This type is defined by the underlying platform and is opaque to the protocol programmer.

5.2 Event Operations

5.2.1 evSchedule

Schedules an event that executes function func with argument arg after delay usec microseconds; usec 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 func. Note that even after an event fires and a thread has been scheduled to 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(EvFunc func, void *arg, unsigned usec)
typedef void (*EvFunc)(Event event, void *arg)
```

Function func 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 func 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 func 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 event)
```

5.2.3 evCancel

Cancels event 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 event)
```

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.

```
bool evIsCancelled(Event event)
```

Figure 3: Repeating events

```
foo_init()
{
    ...
    evDetach(evSchedule(func, arg, INTERVAL));
}

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

5.2.5 evDump

Displays a ps-style listing of *x*-kernel threads when the *x*-kernel is compiled with DEBUG mode. The address of the entry function, the thread state (pending, scheduled, running, finished, or blocked), the time relevant to the thread state, and flags (detached or cancelled), are displayed for each thread controlled by *x*-kernel monitor. The meaning of the time entry varies according to the state. For pending threads, the time is the time until it will be scheduled; for other states it is the time the thread has spent in that state. The time is reset on each transition, i.e., it is not cumulative.

void evDump(void)

5.3 Usage Rules

5.3.1 Repeating Events

Each event that is scheduled executes at most one time. Repeating events are programmed as illustrated in Figure 3.

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. An example is given in Figure 4.

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.

Figure 4: Cancellable events

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

        foo_timeout(Event self, void *arg)
        {
    PState *state = (PState *)arg;

            ...
            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);
        }
```

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 message headers (e.g., addresses, port numbers) into capabilities for (pointers to) *x*-kernel objects (e.g., Protl, Sessn, Enable).

6.1 Type Definitions

The map library defines two data structures: MapElement and Map. A Binding is a pointer to a MapElement. A map element is a table of bindings, where each binding is given by the pair <external key, internal id>. An external key is a variable length byte string, which typically is constructed from various fields in a message header. An internal id is a fixed-sized identifier (e.g., a 32 or 64-bit memory address) which is a pointer to a protocol or session object.

```
typedef struct mapelement {
        struct mapelement*next;
        void              *externalkey;
        void              *internalid;
} MapElement, *Binding;

typedef struct {
        int               nEntries;
        int               keySize;
        MapElement        *cache;
        MapElement        *freelist;
```

```
        MapElement      **table;
        XkReturn        (*resolve)();
        Binding         (*bind)();
        XkReturn        (*unbind)();
        XkReturn        (*remove)();
} *Map;
```

6.2 Map Operations

6.2.1 mapCreate

Creates a map with table_len elements in it. External keys bound in this map are keySize bytes long. The maximum value for the key size is MAX_MAP_KEY_SIZE, currently 100 bytes. Programmers should normally use sizeof(structuretype) as the key size to facilitate platform independence. Note that maps never overflow, but they perform best if table_len is chosen so that the map is at most 50-80% full. Returns 0 if the map could not be created.

```
Map mapCreate(int table_len, int keySize)
```

6.2.2 mapBind

Adds a binding of external key to internal id to map. The binding will be done with keySize bytes of what key 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. A return value of ERR_BIND indicates that the key is already bound in the map to a different id. If the key is already bound to the same id, that binding is returned.

```
Binding mapBind(Map map, void *key, void *id)
```

6.2.3 mapResolve

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

```
XkReturn mapResolve(Map map, void *key, void **id)
```

6.2.4 mapRemoveBinding

Removes binding bind from map. Returns XK_FAILURE if the item is not in the map.

```
XkReturn mapRemoveBinding(Map map, Binding bind)
```

6.2.5 mapRemoveKey

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

```
XkReturn mapRemoveKey(Map map, void *key)
```

6.2.6 mapClose

Destroys 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 callback function func. Each call to mapForEach puts the external key and its internal id into arguments passed the function func. The third argument passed to func is the supplied value arg. As long as the flag MFE_CONTINUE is set in the callback function's return value and there are unprocessed keys, mapForEach will continue to call func.

If the flag MFE_REMOVE is set in the return value of the callback function, mapForEach will remove the entry from the map after the user function returns and before it is called with the next map entry. This is the only correct way to remove the "current" map entry during a mapForEach operation. If the user callback function attempts to remove the "current" entry directly (via mapRemoveBinding or mapRemoveKey), the result is unpredictable and may result in system crashes.

It is currently possible to remove entries other than the "current" entry from within the callback function. However, we strongly discourage such use as its correctness depends on implementation details that may change in future versions of the *x*-kernel.

New map entries added in the middle of a mapForEach iteration may or may not show up during that iteration. Map manipulations within a mapForEach user function are generally not recommended.

MFE_REMOVE and MFE_CONTINUE are binary flags which may be combined using bitwise OR. The order in which keys are returned depends on the internal structure of the map.

```
void mapForEach(Map map, MapForEachFun func, void *arg)
```

```
typedef int MapForEachFun(void *key, void *id, void *arg)
```

6.3 Usage Rules

6.3.1 Map Modifications During mapForEach

It is no longer permissible to directly remove the "current" entry in a mapForEach callback function, as it was in *x*-kernel version 3.2. When adapting a version 3.2 protocol, care should be take to remove such illegal map modifications.

Illegal map modifications often appear in timeout handlers closing active sessions. Such code can be fixed easily because it is customary to save the binding of an active session in the binding field of the Sessn and for functions that close sessions to check this field and perform a mapRemoveKey only if it is non-zero. The fix is to modify the timeout handler to (a) reset binding to zero, (b) call the session-closing function, and (c) return the flag MFE_REMOVE in addition to any other flags that may have been returned by the mapForEach callback function. This ensures that the map entry for the active session that is being closed remains in the map until the mapForEach callback function returns.

6.3.2 External Keys

Maps are used to bind a variable length external key to an internal id of type int. The size of the external key is given as an argument when a particular map is created. All external keys bound using this map are expected to be of this size. It's important that you 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) to fail to map to the same value in the *x*-kernel.

6.3.3 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 keys to Enable objects (Section 2.1.2), thereby allowing a protocol to create a session when a message that is part of a new connection arrives. Typically, a protocol binds an active key to a session in its xOpen routine, and a passive key to an enable object in its xOpenEnable routine. These bindings are then used in the protocol's xDemux routine.

7 Thread Library

The *x*-kernel uses a "thread-per-message" model of computation, and provides primitives for synchronizing threads. The following operations affect thread scheduling. Of these, only semWait 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 of which protocol programmers need be aware 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 sem with a count of count. 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 *sem, int count)
```

7.2.2 semWait

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

```
void semWait(Semaphore *sem)
```

7.2.3 semSignal

The current thread decrements the use count for semaphore sem. 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 *sem)
```

7.3 Delay

Delays the current thread for at least msec milliseconds. This is not a thread primitive, but a library routine built on top of semWait/semSignal. Note that the argument is in milliseconds, while the time argument to evSchedule (Section 5.2.1) is in microseconds.

```
void Delay(int msec)
```

7.4 Usage Rules

7.4.1 Scheduling and Preemption

The currently executing thread gives up control by either terminating or executing a semWait 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-to-protocol operations are considered to have the potential to cause a thread switch, and all data structures must be "secured" before calling such operations.

7.4.2 Blocking

Although the *x*-kernel advocates a "thread-per-message" model and it provides primitives for blocking threads, 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 not be able to proceed, it should put the message in a protocol-dependent queue and return. Later, another thread can pick up the message from 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 waiting 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.

7.4.3 External Threads

Where the *x*-kernel is embedded in another operating system, there may be asynchronous threads representing device drivers or user requests that want to enter the *x*-kernel. These threads must, in general, acquire the *x*-kernel master lock (i.e., enter the *x*-kernel monitor) with xk_master_lock before performing any *x*-kernel operations, including other thread synchronization operations. (This isn't necessary for normal *x*-kernel threads because threads started by evSchedule acquire the master lock automatically when they start running.) Unless a call is explicitly documented otherwise, threads may not make *x*-kernel system or library calls without holding the master lock.

A thread acquires and releases the master *x*-kernel lock with the following operations.

```
void xk_master_lock(void)

void xk_master_unlock(void)
```

Note that normal protocols should not use these operations. The only place that they are meaningful is in anchor protocols, such as device drivers, and application-level interfaces, that have to transition between the *x*-kernel and the host OS. Also note that this interface is not part of the official *x*-kernel interface; it is internal to the current implementation of the *x*-kernel.

7.4.4 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.

7.4.5 Multiprocessor Support

Version 3.3 of the *x*-kernel is MP-safe, although probably not MP-performant. This is because all threads executing in the *x*-kernel must first acquire a master lock; i.e., the *x*-kernel is currently implemented as a single monitor. Research projects at the Swedish Institute of Computer Science and The University of Massachusetts have been investigating the addition of finer-grain locks.

8 Trace Library

The *x*-kernel provides two different facilities for tracing protocol execution. The first, which is described in this section, supports the conditional printing, in printf format, of statements taking from zero to six variables. Every protocol should make use of the trace facililty described in this section. The second, which is described in the next section, supports the collection of fine-grain trace data, and the storage of this data to files, where it can later be analyzed. Protocols use this more advanced facility only when they are being instrumented for detailed performance analysis.

8.1 Type Definitions

The current value of the trace 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 12). 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_ERRORS	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$n$ macros take n arguments (where $0 <= n <= 6$) in addition to the variables tracevar, tracelevel, and formatstring. tracevar is a name associated with the protocol or subsystem 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 "trace" prepended and "p" appended; e.g., udp has trace variable traceudpp. In addition to protocol tracing, there are *x*-kernel trace variables for subsystems: e.g., init, processswitch, protocol, processcreation, event, msg and ptbl. These are defined in the file xkernel/include/xk_debug.h.

Note that the trace facility automatically supplies a newline at the end of the trace message, therefore the supplied format string need not. Also, the trace facility prepends "trace" to the tracevar argument passed in. Thus, the first argument must be the protocol name with only "p" appended; e.g., udpp for udp. Because of this prepending, 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);
```

will print the trace message if the *x*-kernel was built in DEBUG mode (see Section 12 and if TR_ERRORS <= traceudpp.

8.2.2 xTraceP, xTraceS

The xTracePn and xTraceSn macros function much the same way as the xTrace$n$ macros, except that they take a Protl or Sessn as their first parameter (instead of a trace variable) and they print the protocol instance name before the rest of the trace statement. This turns out to be very useful when reading an *x*-kernel trace where several protocols were interleaving trace statements. We recommend using the xTracePn and xTraceSn macros whenever you have an appropriate Protl Sessn in scope, using the xTrace$n$ macros only when there is no such Protl or Sessn available.

xTracePn(Protl protocol, int tracelevel, char *formatstring, args, ...) xTraceSn(Sessn session, int tracelevel, char *formatstring, args, ...)

8.2.3 xIfTrace, xIfTraceP, xIfTraceS

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();
```

xIfTraceP and xIfTraceS are the analogous operations, taking a Protl or Sessn instead of a trace variable.

xIfTraceP(Protl protocol, int tracelevel) xIfTraceS(Sessn session, int tracelevel)

8.3 Usage Rules

Trace statements are macros which are only active in DEBUG mode (see Section 12). If you are writing a new protocol, you should insert trace statements. Even though there will be no bugs left after you release your protocol, it may help others in debugging their protocols. Don't delete these very helpful debugging statements when you are done.

The trace levels listed in Section 8.1 are in increasing order of severity. When an *x*-kernel runs with tracing enabled, trace statements associated with a trace variable will print if their trace level is at least as severe as the value of the trace variable. For example, if the TCP trace variable is set to TR_GROSS_EVENTS, this will cause TCP trace statements with trace levels of TR_GROSS_EVENTS, TR_ERRORS and TR_ALWAYS to be displayed. To display all TCP trace statements, you would set the TCP trace variable to have the value TR_FULL_TRACE.

9 Data-Trace Library

In addition to the trace facilities that print information to standard output, as described in the previous section, the *x*-kernel also provides a facility for saving detailed trace information about protocol execution to disk. This data can later be processed by various protocol-specific analysis tools. We anticipate most protocols using the trace facility described in the previous section, rather than the facility given in this section.

This data tracing facility supports operations for creating and managing circular trace buffers, writing trace entries to a buffer, saving traces to a file, and appending "postamble" information to trace files..

9.1 Type Definitions

The data tracing facility defines three data structures: dt is the main object associated with a trace (it manages the trace buffers and output file); dthdr keeps track of the numbers and sizes of trace buffers written to the output file; and dtpost manages the postamble list (postamble buffers are written to the trace file after the trace has completed).

```
typedef struct dt_object_struct {
        char          *buffer;
        char          *current;
        char          *last;
        char          *traceName;
        int           fileSize;
        dthdr         fileHdr;
        dtpost        *post;
        int           numPost;
        dtCloseFunc   closeFunc;
        void          *closeArg;
        struct dt_object_struct *next;
} dt;

typedef struct dt_filehdr_struct {
        int           version;
        int           bufferSize;
        int           numberBuffers;
        int           lastBufferIdx;
        int           lastBufferSize;
} dthdr;

typedef struct dt_postamble_struct {
        char          *buffer;
        int           size;
        struct dt_postamble_struct *next;
} dtpost;
```

9.2 Operations

9.2.1 dtCreateTraceObj

Creates and initializes a trace object with name traceName. The traceName and instName fields are also used to create the name of the trace output file. If instName is NULL, the output file is "traceName.dt"; otherwise, it is "traceName_instName.dt" (substituting the appropriate values for the variable names).

The logsize parameter specifies the size of the trace buffer in bytes, and fileSize states the maximum length of the trace file in terms of trace buffers (e.g., logsize = 10000 and fileSize = 3 means a trace buffer of approximately 10KB and a maximum trace file size of 30KB).

dt *dtCreateTraceObj(char *traceName, char *instName, int logsize, int fileSize)

Note that both the trace buffer and trace file are circular. When the trace buffer is full, it will be flushed to disk; when the trace file is full, the next trace buffer written will overwrite the first one in the file.

A list of all trace objects created by any protocol is maintained by the datatrace tool. The newly created trace object is put at the end of this list.

9.2.2 dtTrace

The dtTrace$n$ macros take n arguments in addition to a pointer to a dt object. The effect of all of them is to save the trace variables given as arguments to the trace buffer, and advance the buffer pointer. When the trace buffer becomes full, it is flushed to disk and the buffer pointer is reset to the start of the trace buffer.

void dtTrace$n$(dt *dtobj, args,...)

9.2.3 dtTraceBuf

The dtTraceBuf macro can be used instead of dtTrace$n$. It copies a single buffer, pointed to by buf and of length len, to the trace buffer, and advances the buffer pointer. When the trace buffer becomes full, it is flushed to disk and the buffer pointer is reset to the start of the trace buffer.

dtTraceBuf(dt *dtobj, char *buf, int len)

9.2.4 dtFlushTraceObj

Flushes the data in the buffers to the data file. Also flushes the postamble data if flush_post is non-zero.

void dtFlushTraceObj(dt *dtobj, int flush_post)

9.2.5 dtRegisterCloseFunc

Associate closefunc with trace object dtobj. Function closefunc is invoked with argument closearg when dtClose() is called.

void dtRegisterCloseFunc(dt *dtobj, dtCloseFunc closefunc, void *closearg)

9.2.6 dtClose

This function first calls the function registered with dtobj by dtRegisterCloseFunc(), if there is one. It then removes the trace object from the trace object list, flushes the trace buffer to disk, and frees all storage associated with the object.

void dtClose(dt *dtobj)

9.2.7 dtCloseAll

Invokes dtClose() on all trace objects. This function should be called at the end of the program.

void dtCloseAll()

9.2.8 dtAppendPostAmble

Adds a buffer to the trace object dtobj, which is flushed to the end of the file when dtClose() is called. Assumes that the buffer has been preallocated. The buffer is placed at the end of the postamble list.

XkReturn dtAppendPostAmble(dt *dtobj, char *buffer, int size)

9.2.9 dtInsertPostAmble

Adds a buffer to the trace object dtobj, which is flushed to the end of the file when dtClose() is called. Assumes that the buffer has been preallocated. The buffer is placed at the beginning of the postamble list.

XkReturn dtInsertPostAmble(dt *dtobj, char *buffer, int size)

9.2.10 dtPostAmbleLocation

Returns the offset from the beginning of the file to the beginning of the postamble information.

long dtPostAmbleLocation(dthdr *FileHdr)

9.2.11 dtGetTraceObj

Returns the trace object that was created with name traceName.

dt *dtGetTraceObj(char *traceName)

9.2.12 dtGetTopTraceObj

Returns the first trace objects in the list of trace objects.

dt *dtGetTopTraceObj()

9.2.13 dtLoadXObjRomOpts

This routine would typically be called in a protocol's initialization routine, if the protocol supports tracing. See Section 9.3 for more information.

void dtLoadXObjRomOpts(Protl prot)

9.3 Usage Rules

A romfile entry can be used to create a trace object for a protocol that supports tracing. The protocol's initialization routine should include a call to dtLoadXObjRomOpts(); when this function is invoked, the romfile is scanned looking for entries that bear the name of that protocol and that have meaning to the datatrace facility.

A romfile entry to create a trace object for the IP protocol would look like:

```
ip   trace   name=ip_trace   logsize=10000   filesize=3;
```

The first argument in the romfile entry must be the protocol name, and the second is "`trace`". The "`name`" argument is optional; if not specified, the trace object is given no name. The above romfile entry would result in a call to dtCreateTraceObj() with the specified parameters when the dtLoadXObjRomOpts() function was invoked.

10 Utility Routines

10.1 Storage

10.1.1 xMalloc

Essentially the same as the Unix malloc routine. Causes an *x*-kernel abort if no storage is available; therefore, it has no error return value.

```
char *xMalloc(int size)
```

The *x*-kernel provides a macro, *X_NEW*, that can be used to allocate space of a certain type.

```
#define X_NEW(type) (type *)xMalloc(sizeof(type))
```

10.1.2 xFree

Frees previously allocated memory.

```
int xFree(char *buf)
```

10.2 Time

The *x*-kernel uses a time structure that is the same as that of Unix.

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

10.2.1 xGetTime

Sets time to the current time of day.

```
void xGetTime(XTime *time)
```

10.2.2 xAddTime

Sets result to the sum of time_1 and time_2. Assumes time_1 and time_2 are in standard time format (i.e., does not check for integer overflow of the usec value).

```
void xAddTime(XTime *result, XTime time_1, XTime time_2)
```

10.2.3 xSubTime

Sets result to the difference of time_1 and time_2. The resulting value may be negative.

```
void xSubTime(XTime *result, XTime time_1, XTime time_2)
```

10.3 Panic Conditions

10.3.1 xAssert

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

xAssert(bool exp)

Note that xAssert statements are macros which are only active in DEBUG mode (see Section 12). In OPTIMIZE mode, xAssert and trace statements go away completely. You should keep this in mind to avoid bugs that show up only in OPTIMIZE mode. For example, the statment:

```
xAssert(mapResolve(map, key, &p) == XK_SUCCESS);
```

will have no effect in OPTIMIZE mode. You should be careful to separate the operation and the check of the return code, as follows.

```
res = mapResolve(map, key, &p);
xAssert(res == XK_SUCCESS);
```

10.3.2 xError

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

xError(char *ErrorString)

10.4 Byte Order: ntohs, ntohl, htons, 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)
```

10.5 Checksum

10.5.1 inCkSum

Calculates a 16-bit 1's complement checksum over buffer (of length length) and message, returning the bit complement of the sum. length should be even and the buffer must be aligned on a 16-bit boundary. length may be zero.

u_short inCkSum(Msg *message, u_short *buffer, int length)

10.5.2 ocsum

Returns 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)

10.6 Strings to Hosts

Utility routines exist for converting from string representations of IP and Ethernet addresses to their structural counterparts and vice-versa.

10.6.1 ipHostStr

Returns a pointer to a string with a "dotted-decimal" representation of IP host host (e.g., "192.12.69.1"). This string is in a static buffer, so it must be copied if its value is to be preserved.

char *ipHostStr(IPhost *host)

10.6.2 str2ipHost

Interprets str as a "dotted-decimal" representation of an IP host and assigns the fields of host accordingly. The operation fails if str does not seem to be in dotted-decimal form.

XkReturn str2ipHost(IPhost *host, char *str)

10.6.3 ethHostStr

Returns a pointer to a string with a representation of Ethernet host host (e.g., "8:0:2b:ef:23:11"). This string is in a static buffer, so it must be copied if its value is to be preserved.

char *ethHostStr(ETHhost *host)

10.6.4 str2ethHost

Interprets str as a six-hex-digit-colon-separated representation of an Ethernet host and assigns the fields of host accordingly. The operation fails if str does not seem to be in the correct format.

XkReturn str2ethHost(ETHhost *host, char *str)

10.7 Host Name Service

A simple way of mapping host name strings to host IP addresses is provided via rom file entries (see Section 12.4) and the interface function xk_gethostbyname.

During *x*-kernel startup, rom file lines beginning with the string "dns" are parsed into name and address components and added to the host name table. E.g.:

```
dns umbra 192.12.69.97
```

The host name must be less than 64 characters in length.

10.7.1 xk_gethostbyname

This function will look up a hostname and return its IP address in addr. The name must be an exact match to a rom file entry; no substrings are allowed. If the name is not found, the return code indicates failure.

XkReturn xk_gethostbyname(char *name, IPhost *addr)

10.8 ROM file parsing utilities

When writing a protocol that provides user-configurable ROM file options, you can make use of the ROM file parsing utilities to process the ROM file entries. To use these utilities:

1. Write separate routines to handle each ROM option your protocol will support. These routines should be of the following type:

   ```
   typedef XkReturn (*ProtlRomOptFunc)(Protl protl, char **fields, int numFields,
                                        int lineNumber, void *arg)
   ```

The ROM file parsing code will call this handler routine when it finds an appropriate line in the ROM file. The number of fields on that line and the fields themselves will be placed in numFields and fields, respectively. The line number is provided to allow the handler to produce error messages if desired.

If the handler returns XK_FAILURE, the parsing code will print a generic "ROM file format error" message, specifying the name of the protocol and the line number.

2. Create an array of ProtlRomOpt structures which bind option names to their handling functions. There is one ProtlRomOpt structure for each ROM option.

```
typedef struct {
        char               *name;      /* name of the option as specified in ROM file */
        int                minFields;  /* minimum number of fields for this option */
        ProtlRomOptFunc    func;       /* handler function */
} ProtlRomOpt;
```

3. Somewhere in your protocol's initialization code, call findProtlRomOpts.

```
XkReturn findProtlRomOpts(Protl protocol, ProtlRomOpt *opts, int numOpts, void *arg)
```

This routine scans through the ROM file, looking for lines where the first field matches either the protocol name or the full instance name of protocol (e.g., if the protocol instance is ethdrv/SE0, ROM file entries with either ethdrv/SE0 or ethdrv would match). When such a match is found, the array opts, of ProtlRomOpts, is scanned. If the second field of the line matches the name field of one of the opts entries, or if the name field of one of the opts entries is the empty string, the ProtlRomOptFunc for that option is called with the protocol, all ROM fields on that line, the number of fields on that line, the line number and the user-supplied argument.

If the first field of a ROM line appears to match protocol, but none of the supplied opts entries matches the second field, an error message will be printed and XK_FAILURE will be returned. The rest of the ROM entries will not be scanned. This same behavior results from the ProtlRomOptFunc returning XK_FAILURE and from a ROM line with too few fields for its associated handler.

As an example, consider a protocol which supports two ROM options, a port option and an mtu option, both of which take single integer parameters. The example in Figure 5 shows how this protocol would interface with the romopt parsing utilities.

Figure 5: Interfacing with the romopt parsing utilities

```
static XkReturn readMtu(Protl, char **, int, int, void *);
static XkReturn readPort(Protl, char **, int, int, void *);

static ProtlRomOpt opts[] = {
    { "mtu", 3, readMtu },
    { "port", 3, readPort }
};

static XkReturn
readMtu(Protl self, char **arr, int nFields, int line, void *arg)
{
    PState *ps = (PState *)self->state;

    return sscanf(arr[2], "%d", &ps->mtu) < 1 ? XK_FAILURE : XK_SUCCESS;
}

static XkReturn
readPort(Protl self, char **arr, int nFields, int line, void *arg)
{
    PState *ps = (PState *)self->state;

    return sscanf(arr[2], "%d", &ps->port) < 1 ? XK_FAILURE : XK_SUCCESS;
}

foo_init(Protl self)
{
    ...
    findProtlRomOpts(self, opts, sizeof(opts)/sizeof(ProtlRomOpt), 0);
    ...
}
```

11 Control Operations

Control operations are used to perform arbitrary operations on protocols and sessions, via the xControlProtl and xControlSessn operations described in Sections 2.2.9 and 2.2.15. xControlProtl and xControlSessn 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.

The opcode field in the control operations specifies the operation to be performed on the protocol or session. There are two "classes" of operations: standard ones that may be implemented by more than one protocol, and protocol-specific ones.

11.1 Standard Control Operations

11.1.1 Operations Common to Both Protocols and Sessions

These operations can be performed on both protocols and sessions.

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 could 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 protocol 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 its lower protocol 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).

11.1.2 Session-Only Operations

These operations can be performed on sessions only.

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.

FREERESOURCES

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

SETNONBLOCKINGIO

Treats the buffer as a pointer to an int (non-zero == TRUE). This operation is interpreted by sessions which do output buffering. Such sessions may block threads executing an xPush until sufficient buffer space is available to hold the outgoing message. If SETNONBLOCKINGIO with value TRUE is performed on such a session, a thread which would normally block in such a situation returns with an XMSG_ERR_WOULDBLOCK message handle instead.

11.2 Protocol-Specific Control Operations

While all protocols support the control operations enumerated above, it is not uncommon for any given protocol to also support a collection of protocol-specific opcodes. These opcodes can be associated with either both the protocol's session and protocol objects, or with just its session objects. These opcodes are defined relative to an identifier that has been assigned to each protocol (in the file include/upi.h). 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 being written (i.e., until it's been defined in upi.h), a set of temporary ids, TMP0_CTL, TMP1_CTL, ... TMP4_CTL, can be used.

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

11.3 Forwarding Control Operations

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

12 Configuring a Kernel

This section describes how to configure and build an *x*-kernel. You will need to substitute the pathname where your system's *x*-kernel tree resides for /usr/xkernel in the following.

The *x*-kernel configure and build procedure is the same, regardless of whether you are building a user_level, standalone, or simulator *x*-kernel. For simplicity, we explain how to build a user_level kernel. Substitute stand_alone or simulator for user_level in the pathnames that follow to build standalone and simulator kernels, respectively.

12.1 Build Directory

The *x*-kernel user must set up a "build directory" in which to construct an instance of the *x*-kernel. Build directories are usually created within a user's home directory.

Each build directory can support one *x*-kernel configuration at a time. The contents of three types of files determine an *x*-kernel configuration. They are:

- A graph.comp file specifies the collection of protocols that are to be included in the kernel and the relations between them.
- Protocol table files (prottab) define the number space for protocols to identify each other.
- ROM files specify runtime options, such as the IP address of the host machine on which an *x*-kernel will be run.

The graph.comp file must reside in the build directory. Protocol table files and ROM files are not required to be in the build directory; later, we show how to specify the locations of these files. Note that the graph.comp file is read in during the build phase, and so represents an *x*-kernel's static configuration. The protocol table files and ROM files are scanned at runtime, and so allow dynamic configuration of the *x*-kernel.

Directory /usr/xkernel/user_level/build/Template contains samples of common graph.comp and ROM files. The graph.comp file should be copied from this directory to your build directory. Also, the appropriate Makefile for your platform must be copied from this directory to the build directory; it should be renamed Makefile, and made writable. The sample ROM file found in this directory may be copied to a directory from which you intend to run the *x*-kernel; more on this in Section 13.

For the purpose of the remaining discussion, we assume you are configuring a kernel so as to implement and evaluate protocol ASP (A Simple Protocol), the example protocol used in the *x*-kernel Tutorial [8].

12.2 Specifying a Protocol Graph

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

The first two sections—device drivers and protocols—describe the protocol graph to be configured into the *x*-kernel. The only difference between the two sections is that drivers in the first section are initialized directly from the *x*-kernel boot thread, whereas protocols in the protocol section are initialized from a distinct protocol initialization thread. For the device drivers and platforms in this distribution, this distinction is of no consequence and device drivers may be configured in either the first or the second section.

Device drivers and protocols are described by the same types of entries, as illustrated by the following example.

```
name=asp files=asp dir=asp 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 that implement the protocol. Files are specified without extensions. The dir and files fields are not used in the common case where you want to link in protocol object code from the public system object area (/usr/xkernel/protocols); they are used only when you want to compile and link code from your private build area. If a files entry exists but no dir entry is specified, the current directory (i.e., the build directory) is assumed. If a dir entry exists without a files entry, the files field defaults to a single .c file with the protocol's name.

The protocols field indicates the protocols directly below the current protocol in the graph, that is, the protocols upon which this protocol depends. 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 tha 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 traceaspp.

Multiple instantiations of protocols are supported by using a "/" character after the protocol name, and then adding a unique suffix. In the following example, two instantiations of "asp" are indicated, one over "ip" and one over "eth," and both are used by the "prt" protocol. In this example, each instance suffix for the "asp" protocol is the name of the protocol below the instance, but this is just a convention; any distinct string could be used as an instance suffix. Note that only the first of multiple instantiations should have dir, files, or trace fields.

```
name=asp/ip files=asp dir=asp protocols=ip trace=TR_MAJOR_EVENTS;
name=asp/eth protocols=eth;
name=prt files=prt dir=prt protocols=asp/ip,asp/eth trace=TR_ERRORS;
```

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

```
@;
#
# You can specify protocol tables to be read in at boot time.
#
prottbl=/usr/xkernel/etc/prottbl.std;
prottbl=./prottbl.local;
#
# You can specify subsystem tracing for messages and protocol operations
# (see file include/xk_debug.h for a list of subsystem trace variables).
#
name=msg        trace=TR_GROSS_EVENTS;
name=protocol   trace=TR_MAJOR_EVENTS;
#
# You can specify the name of the ROM file to be used; it will be read
# during "make compose" and incorporated into the xkernel runtime image.
#
romfile=romfile.asp;
#
# You can specify romfile contents (see section "ROM options").
#
romopt shepherd threads 8;
```

The graph.comp file is read by an *x*-kernel utility program called compose. This utility 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.

12.3 Protocol Tables

The *x*-kernel runtime environment always includes a protocol table that defines the number space protocols use to identify each other. The *x*-kernel builds a table of protocol numbers by reading configuration files at runtime, and provides an operation for protocols to query this table to determine the protocol corresponding to a protocol number. This frees protocol developers from having to embed explicit protocol numbers in the protocol code itself.

The reason that the protocol table is needed may not be obvious. After an *x*-kernel protocol receives an incoming data message and finishes processing it, the message must be passed up to the appropriate higher-level protocol. To do this, the protocol must determine the high level protocol to which the message belongs; for example, the IP protocol must decide if a received message should be passed up to UDP or TCP. Each *x*-kernel message carries in its message headers fields containing the numbers of the protocols that sent it, and these determine which protocols should process the message on the receiving host. Thus, the protocol holding the message extracts the protocol number of the next high level protocol from the message header, queries the protocol table, and passes the message on to the indicated protocol.

If the graph.comp file does not include any non-standard protocols, one of the default protocol tables in /usr/xkernel/etc may safely be used.

The following is an example protocol table file.

```
#
# prottbl
#
# This file describes absolute protocol id's and gives relative
# protocol numbers for those protocols that use them.
#

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
```

Each protocol has an entry of the form:

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

where idNumber uniquely identifies each protocol.

There are two ways for a protocol to define its protocol number space. The first technique uses an *explicit numbering* scheme; the protocol explicitly indicates which protocols may be configured above it and the relative number that should be used for each higher-level protocol. Use of this numbering scheme is indicated by the presence of the optional *hlp* list after the protocol's *idNumber*.

The second technique uses an *implicit numbering* scheme; 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 protocol number relative to BLAST. Protocol ID numbers are 4-byte quantities in the *x*-kernel, so protocols using implicit numbering have a 4-byte field in their headers for the upper protocol number.

Implicit numbering 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 implicit numbering scheme.

The *x*-kernel is distributed with two useful protocol table files: prottbl.std and prottbl.nonstd. These files are located in /usr/xkernel/etc. The first of these two files, prottbl.std, contains the standard protocol numbers; e.g., TCP is known as protocol "6" relative to IP, IP is known as protocol "x0800" relative to the ethernet, and so on. This file should be used when there is no danger of interfering with the standard protocol suite on the platform you are using. The second file, prottbl.nonstd, uses non-standard protocol numbers. You will want to use this file when there is a danger of interfering with the platform's native protocol stack; e.g., when both the *x*-kernel and the native stack are running and getting packets diverted their way by a packet filter.

All *x*-kernels must load at least one protocol table, and all protocols configured in a kernel 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. 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
#

asp  1000

ip   2  {
    asp  200
}
```

Here, we define our new protocol ASP 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.

12.3.1 ROM options

The protocol table files used by a particular instance of the *x*-kernel can be specified in either the graph.comp file (Section 12.2), or in a ROM file (see Section 12.4). The ROM file format is:

```
prottbl FILENAME1
prottbl FILENAME2
...
```

In both cases, the named protocol tables are loaded at runtime.

By default, the protocol table allows multiple upper protocols to use the same protocol number relative to a single lower protocol. To restrict this behavior, so that each upper protocol must have a unique relative protocol number, the `unique_hlps` ROM option may be used.

```
prottbl unique_hlps on
```

12.4 ROM Files

ROM files allow specification of runtime options for protocols and various subsystems. When a protocol instance or *x*-kernel subsystem initializes, it typically scans a list of user-provided options in the ROM file to see if it should adjust its default parameters for that particular instantiation. ROM options are used for a variety of purposes, such as providing initial values for databases, specifying numbers of network shepherd threads, and providing IP gateway information.

Each ROM file entry consists of a single line. The first field in each line specifies the particular protocol or subsystem that should interpret that line. The rest of the fields are specific to that particular protocol or subsystem. Comments can be added following a #. For example, given the following ROM file:

```
#
# Example ROM file
#

simeth port 1234

arp 192.12.69.49 192.12.69.1 1234
arp 192.12.69.45 192.12.69.1 9876

prottbl /usr/xkernel/etc/prottbl.nonstd
```

the SIMETH protocol will interpret the first line, the ARP protocol will interpret the second and third lines, and the protocol table subsystem will interpret the last line.

The exact method for indicating where the *x*-kernel should find its ROM files is specific to the individual platforms and is documented for each platform in Section 12.6.

Protocols that provide ROM file configurable options will describe the format of these options in their man pages in Appendix A.

12.5 Build Procedure

Once you have edited the graph.comp file to include all protocols and device drivers to be configured into the *x*-kernel, an instance of the *x*-kernel can be built. Execute the following steps. (The protocol table and ROM files can be specified and even changed at a later time because they are read at runtime.)

1. Put /usr/xkernel/bin/BINTYPE and /usr/xkernel/bin in your search path, where BINTYPE is one of sunos-sparc, solaris-sparc, osf1-alpha, linux-alpha, linux-x86 or irix-mips.

   These must occur before /bin and /usr/bin. This allows use of the version of make distributed with the *x*-kernel (GNU make v. 3.66), which is included in the BINTYPE directory, rather than the standard Unix make.

2. Modify the Makefile in the build directory. The variable XRT in this Makefile must be a path to the root of the *x*-kernel source tree; e.g., /usr/xkernel. The *x*-kernel uses a trace package to generate debugging information; to enable the tracing facility, set the Makefile variable HOWTOCOMPILE to DEBUG. To obtain accurate performance timings, variable HOWTOCOMPILE should be assigned OPTIMIZE. This causes all trace of tracing code to be eliminated from the kernel.

3. Type: make compose

   If this is the first time make compose has been run, you may see what appear to be error messages about missing files, such as Makefile.local and DEPS/Makedep.*. These warnings can be ignored, since these files will be created by the running of make compose.

4. Type: make depend
5. Type: make

Object files will be placed in a subdirectory of the OBJS directory, whose name reflects the chosen configuration and platform (e.g., UL-DEBUG-sunos-sparc). Object files are stored similarly throughout the *x*-kernel hierarchy. The final *x*-kernel executable (xkernel) will be placed in your build directory.

Steps 3 through 5 must be repeated whenever you change the graph.comp file. If the Makefile is changed, only step 5 must be repeated. Changes to the protocol tables and ROM files do not require rebuilding tke *x*-kernel.

12.6 Examples

The *x*-kernel source tree contains some ready-to-use configuration files to help you build any of the different types of *x*-kernels. This section contains locations and descriptions of these files. All of the configurations specify a protocol stack that includes ETH, ARP, VNET, IP, and ICMP, plus some set of higher level protocols

12.6.1 User-Level with Simulated Ethernet

A user_level *x*-kernel will usually be configured to use the Unix socket facility to send and receive from the network (see Section 13.1). Example graph.comp and ROM files for building and running such a kernel that includes the TCP/IP protocol stack can be found in /usr/xkernel/user_level/build/Template/. Configuration files for other protocol stacks can be found subdirectories of /usr/xkernel/user_level/build/Template/; e.g., example_rpc shows how to configure an RPC stack, and example_msp shows how to configure a stack that includes the MSP and SWP protocols.

Note that in this graph.comp file, the lowest protocol in the protocol stack is the SIMETH driver. Refer to Section 13.1 for an explanation of the ROM file's simeth and arp options.

12.6.2 User-Level with Direct Ethernet Access

An *x*-kernel that uses a simulated device driver like SIMETH can communicate only with other *x*-kernels; it is not possible to exchange messages with a "native" application since an *x*-kernel configured with SIMETH encapsulates the messages it sends in a UDP datagram. To send raw ethernet packets over the network, one needs to configure a kernel that includes a protocol that interacts directly with the device driver of the host OS. Unlike SIMETH, which is supported on all Unix platforms, these protocols are platform dependent. The current distribution includes three such protocols: IRIXFDDI and IRIXETH for IRIX, and ETHPKT for Linux. In all three cases, root access is required to run a kernel with one of these protocols configured in.

Example graph.comp and rom files for building and running a kernel that includes ETHPKT are given in /usr/xkernel/user_level/build/Template/example_ethpkt. Notice that the graph.comp file specifies that protocol table prottbl.nonstd be used. It does this so that all protocols (most importantly, IP) are assigned nonstandard protocol numbers; this prevents messages designated for the *x*-kernel from being acted upon by the machine's native protocol stack, and vice versa.

12.6.3 Simulator

Directory /usr/xkernel/simulator/build/Template/example contains the configuration files needed to build and run the simulator. The graph.comp file is the only one needed to compile the simulator; the others must be in the working directory at runtime.

The files residing in the example directory configure a simulated network consisting of two Ethernets connected by a point-to-point link. Each Ethernet has two hosts, for a total of four. One host on each Ethernet runs the traffic protocol to simulate background traffic from TCP connections. The two other hosts run megtest, which uses TCP to stream one megabyte of data from one host to the other. Three flavors of TCP are configured into the simulator: rtcp (TCP Reno), ttcp (TCP Tahoe), and vtcp (TCP Vegas). Any of these TCPs can be run on the traffic and megtest hosts.

The simulator treats the graph.comp file differently than the user_level and standalone *x*-kernels do. The simulator does not set up its protocol graph using the graph.comp; it only uses it to decide what protocols it must include in the executable. At runtime, the simulator creates the protocol graph using the xsim.data file. Note that in the example graph.comp file, megtest is configured over TCP Reno, but the xsim.data file runs it over TCP Vegas.

There is only one ROM file, and this file contains information for all of the hosts in the simulation. The example ROM file specifies the gateway to which a host will send when its IP datagram is addressed to a machine residing on another network. Note that the ROM entries begin first with the name of the protocol (ip) and then the name of the host which uses that entry (e.g., h1n0 – host 1 on network 0).

Most of the difficulting in configuring the simulator is how to specify the network you want to simulate. This specification is given in the file xsim.data, which is described elsewhere [1].

13 Running a Kernel

This section describes the procedure for running a user_level simulator *x*-kernel. A user_level *x*-kernel runs in Unix user space, and usually uses a Unix socket interface (or other OS-specific interfaces, such as ETHPKT) to send and receive messages on the network. A simulator *x*-kernel also runs in user space, but doesn't use the network hardware at all; instead, it simulates traffic between hosts on a configurable network in virtual time.

13.1 Unix User Level (SunOS/Solaris/OSF/Irix/Linux)

The behavior of an *x*-kernel running as a user task depends on the "device drivers" configured into the kernel. There are two categories of *x*-kernel device drivers: those that send real network packets (e.g., ETHPKT in Linux and IRIXETH in Irix), and those that send encapsulated network packets (e.g, SIMETH or SIMFDDI). Real-packet drivers use platform-specific methods to access network devices, and are relatively straightforward to configure and use. (Be sure to see the manual page for the individual drivers in Appendix A.)

13.1.1 Simulated Drivers

Configuring the encapsulated-packet drivers can be confusing. (We refer to encapsulated-packet drivers as *simulated drivers*, and instances of *x*-kernels using them as *simulated hosts*.) Simulated drivers sit at the bottom of a protocol stack, just like a standard device driver. Instead of sending packets directly to the device, however, they use the Unix socket interface (and thus the Unix implementations of UDP and IP) to send and receive packets. For example, if you implement IP and UDP *within* a user_level *x*-kernel, then the UDP packets produced by the *x*-kernel are, in turn, encapsulated in real UDP packets. This means that protocols and programs built on top of UDP in the *x*-kernel can only talk to their peers in other *x*-kernels; they cannot communicate with "real" versions of those protocols running on a Unix machine, for example.

Since a user_level *x*-kernel with a simulated driver uses a connectionless UDP socket as its transport mechanism, more than one such *x*-kernel can be run on a single workstation. Because of this flexibility, the local IP address used by each kernel (the *simulated IP address*) is decoupled from the IP address of the actual workstation on which it runs (the *real IP address*). Configuration files for a user_level *x*-kernel must therefore indicate not only which UDP port should be used by the simulated driver, but also the binding between the real and simulated IP addresses for each *x*-kernel.

Consider the following example ROM files (as described in Section 12.4) for two user_level *x*-kernels.

```
% cat client/rom

simeth      port    3050
#
#           Sim. IP addr    Real IP addr     Real UDP port
#
arp         128.10.5.54     192.12.69.1      3050
arp         128.10.5.23     192.12.69.1      3051


% cat server/rom

simeth      port    3051
#
#           Sim. IP addr    Real IP addr     Real UDP port
#
arp         128.10.5.54     192.12.69.1      3050
arp         128.10.5.23     192.12.69.1      3051
```

The simeth entries indicate the real UDP port number which each simulated host will use to 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 the same real processor (192.12.69.1) and use different UDP port numbers: 3051 and 3050.) Note that the name of the ethernet protocol appears exactly as it does in the graph.comp file.

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 *x*-kernel runs, and the fourth field is the *x*-kernel's UDP port number. 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. Since ARP broadcasts are infeasible for simulated hosts, each *x*-kernel must be configured with an arp entry for each of its peers.

See the manual page in Appendix **??** for more information on configuring a specific simulated driver.

13.1.2 Running

As the result of configuring a kernel (Section 12), a file named xkernel should exist in your build directory.

While in this directory, you should create a sub-directory for each *x*-kernel instance to be tested. For example, if you intend to start up client and server instances of a user_level *x*-kernel, create two subdirectories, e.g., client and server. In each subdirectory, create a file named rom, an example of which can be copied from /usr/xkernel/user_level/build/Template. The ROM files should contain configuration information as described in Section 12.4, and in the man pages for protocols and device drivers in Appendix A.

Each simulated host runs as a separate Unix process. To run multiple *x*-kernels using a windowing user interface, you should start each process in a separate window. For each simulated host, open a shell command window, cd to the sub-directory that contains that host's rom file (e.g., cd client) and type ../xkernel. Use DELETE or CTRL-C to stop an *x*-kernel.

13.2 Simulator

An executable called xkernel will reside in the build directory after following the procedures described in Section 12. To run the simulation, change to a directory containing the xsim.data, rom, and protocol table files, and invoke the executable.

Note that the *x*-kernel executable created by the simulator is of a different order than the user_level executable. For the latter, each invocation of the executable corresponds to exactly one host instance. However, invoking the simulator runs the entire simulation, which may potentially include thousands of hosts communicating over a variety of networks.

13.3 Running Test Suites

Most protocols distributed with the *x*-kernel come with a test protocol in the protocols/test directory. These protocols typically send a number of round trips for messages of various sizes and report the total time for the test. The most common test protocols are configured into the protocol library, but some may have to be copied and compiled directly in a user's build area. Test protocols compiled into the protocol graph start up automatically with the rest of the protocols.

The behavior of test protocols can be modified by various command-line and ROM file options. See the man page for the TEST protocol in Appendix A.

13.4 Troubleshooting

Many of the problems encountered when running an *x*-kernel turn out to be configuration problems. Setting the debugging variable traceprotocol to TR_EVENTS or higher can be very helpful in identifying problems. A few common symptoms and some things you might want to check if these symptoms occur are:

- The *x*-kernel aborts before the first protocol's init routine is called.

Your *x*-kernel may have been configured without a protocol table. Make sure your graph.comp or ROM files mention at least one protocol table and that the specified protocol table exists and is readable.

- The *x*-kernel hangs in arp_init.

  ARP's initialization routine will not return until it has discovered the binding for its local IP address. If the *x*-kernel hangs in arp_init, ARP is probably sending out RARP requests which are not being answered. Multiple warning messages of the form:

  ARP: Could not get my IP address for interface eth (still trying)

  are an indicator of this problem.

  If you are running the sunos simulator, you must have an ARP binding in your ROM file for your local host (see Section 13.1). On other platforms, an ARP binding for the local host is not necessary if another host on your network will respond to RARP broadcasts. If you do not have such a host (or if it is not responding for whatever reason), adding a local binding to your ROM file should fix the problem.

- Messages sent out on one host are never received on the destination host.

  Check your ROM file. If it contains an initial binding for the destination host which is incorrect, the destination host will not see packets from the sending host.

- An xOpen hangs for a while and then fails.

  An open may fail if ARP cannot resolve the IP address of the destination (turning ARP tracing on can help identify this problem). ARP requests should never be sent on the sunos simulator platform. If you are running on the sunos platform and you see ARP requests being sent, check the ROM file on the sending host and make sure there is an ARP binding for the destination host.

- The *x*-kernel aborts before a specific protocol's init routine is called.

  If the *x*-kernel cannot find a protocol number for your protocol in any of its tables, it will abort before calling that protocol's initialization routine. You will need to add an entry for your protocol in one of the tables (see Section 12.3).

- Messages get to the destination host but never make it up to the appropriate protocol.

  Make sure that the source and destination hosts are running with identical protocol table entries for the protocol in question. If the numbers are different, messages won't get to the appropriate protocol on the destination host.

14 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, instead put the source files in /usr/xkernel/user_level/platforms/PLATFORM/drivers, where PLATFORM is one of sunos, solaris, osf1, irix, or linux. You should put any public .h files in /usr/xkernel/inclue/prot. Finally, you need to update the prottbl files in /usr/xkernel/etc to include protocol numbers for your new protocol, and /usr/xkernel/include/upi.h to include a base control op number for your protocol.

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-help@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 the Appendix. The source files for these man pages are in /usr/xkernel/doc/manual/protocols.

References

[1] L. S. Brakmo, A. C. Bavier, and L. L. Peterson. *x-Sim User's Manual (Version 1.0)*. Network Systems Research Group, Department of Computer Science, University of Arizona, Jan. 1996.

[2] N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[3] D. Mosberger. *Map Library Design Notes*. Network Systems Research Group, Department of Computer Science, University of Arizona, Jan. 1996.

[4] D. Mosberger. *Message Library Design Notes*. Network Systems Research Group, Department of Computer Science, University of Arizona, Jan. 1996.

[5] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[6] L. L. Peterson. *Getting Started with the x-kernel*. Network Systems Research Group, Department of Computer Science, University of Arizona, Jan. 1996.

[7] L. L. Peterson and B. S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers, San Fransico, CA, 1996.

[8] L. L. Peterson, B. S. Davie, and A. C. Bavier. *x-kernel Tutorial*. Network Systems Research Group, Department of Computer Science, University of Arizona, Jan. 1996.

A Protocol Specifications

This appendix describes each of the protocols currently available in Version 3.3 of the *x*-kernel. (Additional protocols are available in Version 3.2.) 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 (pointer to) 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), the CONTROL realm (existing only to allow control operations), or the ANCHOR realm (interfacing with the host system).

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). In the case of control operations that take multiple arguments, a set of types is given. Non-primitive types are generally defined in the protocol's .h file.

EXTERNAL INTERFACE

A description of 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 if the protocol fails to work as advertised.

A.1 ARP

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;`}

`ARP_FOREACH`: This is a kludge to allow non-broadcast device drivers, such as SIMETH, to simulate broadcast without having to keep their own tables of reachable hosts. When the ARP_FOREACH control operation is invoked, ARP will call-back the invoking protocol once for each binding in its table.

Input: ArpForEach == { `void *arg; ArpForEachFunc f;` }

Output: none

typedef int (**ArpForEachFunc**) (*ArpBinding* *, *void* *);

`ETH_REGISTER_ARP`: ARP invokes this control operation on its lower protocol at initialization time so the driver knows which protocol to use if it has to invoke an ARP_FOREACH. This is not pretty.

Input: `XObj`

Output: none

CONFIGURATION

```
name=arp protocols=eth;
```

AUTHORS

Larry Peterson and Norm Hutchinson

A.2 ASP

NAME

ASP (A Simple Protocol)

SPECIFICATION

L. Peterson and B. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers, San Francisco, CA (1996).

SYNOPSIS

ASP is an example protocol that supports an unreleable message delivery service, where the two end-points of an ASP channel are identified by a pair of ports.

REALM

ASP is in the ASYNC realm.

PARTICIPANTS

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

CONFIGURATION

```
name=asp protocols=ip;
```

AUTHORS

Larry Peterson and Andrew Bavier

A.3 BID

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.

AUTHOR

Ed Menze

A.4 BIDCTL

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.

AUTHOR

Ed Menze

A.5 BLAST

NAME

BLAST (RPC Blast Micro-Protocol)

SPECIFICATION

S. O'Malley and L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems 10*, 2 (May 1992), 110–143.

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 large message, fragments it into smaller packets, and sends them. The maximum packet size accepted by BLAST (as returned by the GETMAXPACKET control op) is the product of the maximum number of fragments handled by BLAST (16 by default) and the optimal packet size of BLAST's lower protocol. Blast is tuned for the local area networks and should not be used across the Internet.

The receiver gathers all of the packets and sends a NACK if it has reason to believe (through time-outs or other considerations) that a packet has been dropped. BLAST can handle any number on outstanding messages between two hosts (buffer space permitting, of course). The protocol is bidirectional; 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 time-out occurs or the higher level protocol that sent the message notifies BLAST that it can free the message (through a FREERESOURCES control op.) Users of blast are strongly encouraged to free messages as soon as possible. The sender knows which BLAST (BLAST can be instantiated more than once) and which message to free because when a push was performed blast writes into a message attribute attached by CHAN (or some other high level protocol) a pointer to itself and a 32 bit integer ticket which uniquely identifies the message.

Because the sending BLAST may time-out and release a message before all fragments have been received, BLAST is not reliable. It is, however, very persistent.

BLAST performance is critically dependant upon the time-out strategy used and the initial values of those timers. As mentioned earlier the sender uses a timer to free resources after a set interval has occurred. Tuning this timer for use with higher level protocols which do not explicitly free resources is very difficult. For applications which do free resources this time-out interval has no effect on performance unless it is set to too small a value. The receiver sets a timer whenever a fragment from a new packet arrives. The only purpose of this timer is to detect the drop of the last fragment. This timer is set to some constant times the number of fragments in the message. If this timer expires to early this is detected by the code and the constant is increased by a factor of two. After a NACK is set to the round trip time plus some constant times the number of fragments. The purpose of this time is to generate a new NACK if the original NACK or retransmitted segments are lost.

REALM

BLAST is in the ASYNC realm.

PARTICIPANTS

BLAST neither removes nor adds anything to the participant stacks.

CONTROL OPERATIONS

`FREERESOURCES`: Free the storage associted with the message handle passed as argument. The handle should be a value returned by xPush. (protocol and session).

Input: `xmsg_handle_t`

Output: none

`CHAN_RETRANSMIT`: This is CHAN's way of asking BLAST if it should go ahead and retransmit the message. BLAST returns true (1) if and only it has received no NACK's for this message since the message was sent or the last time CHAN_RETRANSMIT was called. The idea being that CHAN should not retransmit while BLAST is in the process of sending the message.

Input: none

Output: `0 or 1`

`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.

AUTHORS

Sean O'Malley and Ed Menze

A.6 CHAN

NAME

CHAN (RPC Channel Micro-Protocol)

SPECIFICATION

S. O'Malley and L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems 10*, 2 (May 1992), 110–143.

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 Birrell-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.

CHAN must know several things about the transport protocol used to actually send the message. This information is represented in the following structure:

```
typedef struct {
    XObj transport;
    int  ticket;
    int  reliable;
    int  expensive;
    unsigned int timeout;
} chan_info_t;
```

This structure is defined in the CHAN session state and a pointer to it is attached as an attribute to each outgoing message. Before the message is send CHAN zero's out all fields of the structure. When xPush returns CHAN assumes that some lower level protocol may have filled in the fields.

If transport has been defined CHAN will perform a FREERESOURCES control operation on transport when the current message has been acked. If the lower level protocol is reliable CHAN will never retransmit the entire message and will not start a timer. If the lower level protocol is expensive CHAN will not retransmit the entire message when it times out. It simply requests an ACK. The timeout field is ignored for the moment. If transport has been defined CHAN will invoke a CHAN_RETRANSMIT control operation on it before retransmitting. If this control operation returns 0 CHAN will not retransmit the body of the message. This allows a lower level protocol like BLAST to discourage CHAN from retransmitting while the message is still being sent.

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.)

Because CHAN affixes a pointer to the outgoing message it must be in the same address space as any transport protocol which will attempt to set the structure passed in the attribute.

AUTHOR

Sean O'Malley

A.7 ETH

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.

ETH provides support for IEEE 802.3 packet formats. An upper protocol registering with Ethernet type 0 is assumed to the recipient for all IEEE 802.3 packets. Conversely, a protocol using an Ethernet type smaller than the maximum IEEE 802.3 data size will have its packets sent using IEEE 802.3 format (i.e., with the packet length overwriting the type field.)

CONFIGURATION

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

ETH recognizes the following ROM options:

`eth/xxx mtu N`: Instantiation xxx of ETH should use an MTU of N (decimal). Default is 1500.

AUTHOR

Ed Menze

A.8 FDDI

NAME

FDDI (FDDI Protocol)

SYNOPSIS

This hardware-independent protocol provides the interface between the rest of the *x*-kernel protocols and the actual FDDI 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 FDDI protocol for each driver protocol.

REALM

FDDI is in the ASYNC realm.

PARTICIPANTS

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

CONTROL OPERATIONS

`MAC_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

FDDI driver protocols should include the file protocols/fddi/fddi_i.h which defines the interface between FDDI and the drivers.

FDDI 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.

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

FDDI will attach a pointer to an FDDIhdr as a message attribute for each outgoing message:

```
typedef struct {
    FDDIhost     dst;
    FDDIhost     src;
    u_short     type;
} FDDIhdr;
```

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

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

CONFIGURATION

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

FDDI recognizes the following ROM options:

`fddi/xxx mtu N`: Instantiation xxx of FDDI should an MTU of N.

AUTHOR

David Yates

A.9 ICMP

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

A.10 IP

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

`IP_GETPSEUDOHDR`: Fills the buffer with a partial IP pseudoheader, containing the source address, destination address, and the upper protocol type. The packet length field and the zero-block are both set to zero. (session only)

Input: none

Output: `IPpseudoHdr`

`IP_PSEUDOHDR`: Used by protocols that use the IP pseudoheader (e.g., TCP and UDP) to alert protocols between them and IP that they must not change the length of packets without worrying about the length field in the pseudoheader. IP itself simply absorbs this control operation and returns.

Input: none

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*, if one has been configured. The default gateway can be set with a rom file entry of the form:

```
ip    gateway     127.1.22.11
```

If no default getway has been configured, or the specified default gateway 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.

AUTHORS

Clinton Jeffery, David Kays and Ed Menze

A.11 MSP

NAME

MSP (Message Stream Protocol)

SPECIFICATION

L. Peterson and B. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers, San Francisco, CA (1996).

SYNOPSIS

MSP extends SWP to include exlicit connection setup/tear-down and flow control. MSP expects a higher level protocol to buffer incoming messages, and inform MSP as to the amount of available buffer space via the MSP_SETRCVBUFSIZE control op. This is similar to TCP's TCP_SETRCVBUFSPACE control op. The implementation of MSP is directly derived from SWP. Like SWP, MSP is a message-oriented protocol, rather than a byte-oriented protocol like TCP.

Because MSP does not implement congestion control, should the intial advertized flow control window be large enough, it is possible that an MSP source will send a large burst of packets upon startup. This is not unlike TCP's behavior before slow start was implemented. Even if MSP is being run over a single ethernet, is is possible for this initial burst to cause congestion-like losses. This is because when running on top of Unix using SIMETH, the UDP receiver buffer on the receiving host may overflow, analogous to the way router buffers overflow with a non-slow-started TCP. When this happens, the *x*-kernel prints sim_ether ERROR: Can't get next buffer, dropping incoming packet. MSP is robust, however, so it will eventually recover from these losses. An interesting exercise would be to add slow-start to MSP.

REALM

MSP is in the ASYNC realm.

PARTICIPANTS

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

CONTROL OPERATIONS

`MSP_SETRCVBUFSIZE:` Sets the receiver's buffer size to the specified number of bytes. This effectively opens the flow control window to this size.

Input: `int bufsize`

Output: none

CONFIGURATION

```
name=msp protocols=ip;
```

AUTHOR

Tim Newsham

A.12 SELECT

NAME

SELECT (RPC Select Micro-Protocol)

SPECIFICATION

S. O'Malley and L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems 10*, 2 (May 1992), 110–143.

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

A.13 SWP

NAME

SWP (Sliding Window Protocol)

SPECIFICATION

L. Peterson and B. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers, San Francisco, CA (1996).

SYNOPSIS

SWP implements reliable, ordered message delivery using the sliding window algorithm. It is a message-oriented protocol, rather than a byte-oriented protocol like TCP. SWP does not support explicit connection setup or flow control. (See MSP for these features.) This implementation of SWP is loosely based on the one given in the book, but it is much more robust and complete.

REALM

SWP is in the ASYNC realm.

PARTICIPANTS

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

CONTROL OPERATIONS

`SWP_SET_SWS`: Sets the sending window size for this session.

> Input: `int sws`
>
> Output: none

CONFIGURATION

`name=swp protocols=ip;`

AUTHOR

Tim Newsham

A.14 TCP

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.

This implementation of TCP supports input and output buffering. Output buffers are contained within TCP. If the amount of data sent and unacknowledged by the peer reaches the output buffer size, TCP will block subsequent *xPush*'s (or will return XMSG_ERR_WOULDBLOCK in the case of non-blocking I/O.)

TCP provides support for users to work with finite input buffers. TCP will limit the amount of input data sent to its upper protocol via *xDemux* to the size of the input buffer. When data have been consumed from the user's input buffer, free buffer space must be signalled to TCP via a TCP_SETRCVBUFSPACE call (see below.) If a user does not wish to use input buffering, a control message signalling an empty buffer should be sent in response to each *xDemux*.

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

`TCP_GETFREEPORTNUM`: Returns an unused TCP port number. This port number will not be given out to subsequent TCP_GETFREEPORTNUM calls until it is released with TCP_RELEASEPORTNUM. This allows an opener to separate reservation of free ports from the actual open operation, if desired. (protocol only)

Input: none

Output: long

`TCP_RELEASEPORTNUM`: Releases a TCP portnumber previously acquired with TCP_GETFREEPORTNUM. (protocol only)

Input: long

Output: none

`TCP_SETRCVBUFSPACE`: Tells TCP how many bytes in the receive queue are free. (session only)

Input: u_short

Output: none

`TCP_SETRCVBUFSIZE`: Tells TCP the size of the TCP user's receive queue. (session only)

Input: u_short

Output: none

`TCP_GETSNDBUFSPACE`: Asks TCP for the number of free bytes its send queue. (session only)

Input: none

Output: u_short

`TCP_SETSNDBUFSIZE`: Tells TCP to change its send queue to the indicated size (session only)

Input: u_short

Output: none

`TCP_SETOOBINLINE`: Tells TCP whether users wants urgent data to be delivered inline (non-zero == yes.) (session only)

Input: int

Output: none

`TCP_GETOOBDATA`: reads the urgent data (exactly one byte), returning 1 on a successful read or returning 0 if data was either read already or was not received yet (the OOB notification may precede the actual reception of the OOB data)

Input: none

Output: char

`TCP_OOBPUSH`: send a msg in urgent mode

Input: Msg *

Output: char

`TCP_OOBMODE`**:** TCP uses this to tell the user of TCP that it has urgent data present, i.e., TCP does an xControl() call on its parent — THIS IS AN UPCALL! The first void pointer (args[0]) is of type XObj and is a pointer to the TCP session that invoked this operation. The second pointer (args[1]) is of type u_int and is the value of the urgent data mark. The oobmark indicates that the "oobmark-th" byte in the receive queue is the oobdata (or will be the oobdata.)

Note: all protocols using TCP without having OOB data delivered in-band must be prepared to accept this upcall.

Input: void *args[2]

Output: none

CONFIGURATION

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

AUTHORS

Norm Hutchinson, Herman Rao, and David Mosberger-Tang

A.15 TEST

NAME

TEST (instantiated as 'chantest', 'udptest', etc.)

SPECIFICATION

The test protocol, usually running a simple a "ping-pong" test of the protocol below it, for various message lengths and number of round trips.

SYNOPSIS

Transport test protocols run in one of two roles, either as "client" or as "server." In most cases, the client will send a message to the server and wait for a reply before sending the next message. There are no provisions for retransmission: if the protocol below the test protocol drops a message, the test will fail.

CONFIGURATION

When the test protocol instantiates, it can determine which role it should assume in several ways. Command line parameters can be used to cause the same kernel to run as the server on one machine and as the client on another. The server should be started up with a "-s" flag:

```
xkernel -s
```

The client side must be told the host address of the server peer (note that on the sunos platform, this should be the address of the simulated IP host.) This can be done with the "-c" command line option, e.g.:

```
xkernel -c192.12.69.54
```

The number of round trips for each packet size can be set with the "trips" flag:

```
xkernel -trips=10000
```

The test protocols all use the common trace variable `prottest` which can be set in the third section of graph.comp:

```
@;
...
name=udptest    protocols=udp;
@;
name=prottest   trace=TR_EVENTS;
```

If you set a trace level when you declare the test protocol in the second section of graph.comp, it will be ignored.

CAVEATS

Remember that if you are using *simeth* you must use the name of the *simulated* host when you invoke the client, not the real host.

A.16 UDP

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_PORT, UDP will select an unused local port.

CONTROL OPERATIONS

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

Input: none

Output: none

`UDP_DISABLE_CHECKSUM:` Cause the session to not use checksums on its outgoing packets. (session only)

Input: none

Output: none

`UDP_GETFREEPORTNUM:` Returns an unused UDP port number. This port number will not be given out to subsequent UDP_GETFREEPORTNUM calls until it is released with UDP_RELEASEPORTNUM. This allows an opener to separate reservation of free ports from the actual open operation, if desired. (protocol only)

Input: none

Output: long

`UDP_RELEASEPORTNUM:` Releases a UDP portnumber previously acquired with UDP_GETFREEPORTNUM. (protocol only)

Input: long

Output: none

CONFIGURATION

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

AUTHORS

Larry Peterson and Sean O'Malley

A.17 VCHAN

NAME

VCHAN (Channel Virtual-Protocol)

SPECIFICATION

S. O'Malley and L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems 10*, 2 (May 1992), 110–143.

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

A.18 VDELAY

NAME

VDELAY (Virtual Delay Protocol)

SPECIFICATION

Delays outgoing packets. Used to exercise the ability of other protocols to keep the pipe full.

SYNOPSIS

VDELAY sessions delay packets for a fixed number of milliseconds to simulate end-to-end latency. VDELAY is designed to simulate propogation delay, not delays due to queuing and congestion. The high-level protocol can set the delay (measured in milliseconds); the default is 25ms. VDELAY has no other effects on outgoing packets.

REALM

VDELAY is in the ASYNC realm.

PARTICIPANTS

VDELAY passes participants to the lower protocols without manipulating them.

CONTROL OPERATIONS

`VDELAY_SETDELAY`: Sets the delay for this session to the specified number of milliseconds.

> Input: `int delay`
>
> Output: none

`VDELAY_GETDELAY`: Returns the current delay for this session.

> Input: none
>
> Output: `int delay`

CONFIGURATION

VDELAY can be configured between any two ASYNC protocols. It is commonly configured between the protocol you want to test (e.g., MSP) and IP.

AUTHOR

Ed Menze

A.19 VNET

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

A.20 VDROP

NAME

VDROP (Virtual Drop Protocol)

SPECIFICATION

Throws away occasional incoming packets. Used to exercise the recovery mechanisms of other protocols.

SYNOPSIS

VDROP sessions throw away incoming packets at regular intervals. By default, this interval is set in a somewhat random fashion at session creation time, though it can be set explicitly on a per-protocol basis via a ROM option (see CONFIGURATION below) or on a per-session basis via a control operation.

VDROP has no effect on outgoing packets.

VDROP should probably allow sessions to have more interesting distributions of drop intervals than "once every N packets."

REALM

VDROP is in the ASYNC realm.

PARTICIPANTS

VDROP passes participants to the lower protocols without manipulating them.

CONTROL OPERATIONS

`VDROP_SETINTERVAL`: Sets the drop interval for this session. An interval of 1 drops every packet, an interval of 2 drops every other packet, etc. An interval of zero indicates that VDROP is disabled for that session. (session only)

Input: `int interval`
Output: none

`VDROP_GETINTERVAL`: Returns the current drop interval for this session. (session only)

Input: none
Output: `int interval`

CONFIGURATION

VDROP recognizes the following ROM options:

`vdrop/xxx interval N`: Instantiation *xxx* of VDROP will use N as the drop interval for all of its sessions.

AUTHOR

Ed Menze

A.21 VSIZE

NAME

VSIZE (Size Virtual-Protocol)

SPECIFICATION

S. O'Malley and L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems 10*, 2 (May 1992), 110–143.

SYNOPSIS

VSIZE is a virtual protocol that multiplexes messages through N lower-level protocols based on the size of the message being sent. By default, VSIZE determines the maximum packet size that each lower level protocol can handle by performing a GETOPTPACKET control operation on the first N-1 lower protocols (the last lower protocol is assumed to have an infinite maximum packet size). VSIZE sends each message using the lower level protocol with the smallest index whose optimum packet size is greater than the length of the message.

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 "largest message" protocol.

CONFIGURATION

VSIZE's lower protocols should be order by decreasing efficiency and increasing packet size.

VSIZE recognizes the following ROM options:

`vsize/xxx cutoff C1 C2`: Instantiation *xxx* of VSIZE should use a cutoff length of C1 bytes for its first down protocol and a cutoff value of C2 bytes for its second down protocol. This control operation allows the user of VSIZE to override the GETOPTPACKET. Note this operation does not check to see if the specified cutoff value is less than the maximum packet size of the lower level protocol.

AUTHOR

Ed Menze

B Device Drivers

This appendix describes device drivers currently implemented in the *x*-kernel. The descriptions are in the same format as those in Appendix A. Note that these drivers are platform-dependent.

B.1 ETHPKT

NAME

ETHPKT (Raw Ethernet Driver (Linux platform))

SPECIFICATION

ETHPKT provides direct interaction with an ethernet device through a Linux SOCK_PACKET socket.

SYNOPSIS

Each instantiation of ETHPKT is associated with a single ethernet device. ETHPKT has the ability to re-map and block ethernet types to allow an *x*-kernel to coexist with the normal TCP/IP stack.

REALM

ETHPKT is in the ANCHOR realm, supporting the ethernet driver interface described in ETH.

PARTICIPANTS

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

CONTROL OPERATIONS

`GETMYHOST`: Returns the six byte hardware address for the ethernet device.

> Input: none
>
> Output: ETHhost*

`ETH_SETPROMISCUOUS`: Enables promiscuous mode for the ethernet device.

> Input: none
>
> Output: none

EXTERNAL INTERFACE

ETHPKT adheres to the external interface defined by ETH.

CONFIGURATION

ETHPKT requires no lower protocol. The default network device is "eth0", which corresponds to the primary ethernet adapter in the host machine.

```
name=ethpkt;
```

ETHPKT recognizes the following ROM options:

`ethpkt/xxx block type`:

Instantiation `xxx` of ETHPKT will block all ethernet packets of the given `type` from being processed by the driver. The `type` should be specified in hex and should be in network byte order. There is no limit to the number of `block` options.

`ethpkt/xxx device name`:

Instantiation `xxx` of ETHPKT will use the given network device. This is the name used internally by the Linux kernel. If no option is provided the default is "eth0".

`ethpkt/xxx remap realtype bogustype`:

Instantiation `xxx` of ETHPKT will re-map all outgoing ethernet packets of type `realtype` to `bogustype`. The reverse will be done to all incoming packets. The `realtype` and `bogustype` should be specified in hex and should be in network byte order. There is no limit to the number of `remap` options. Although subsequent operations on previously mapped ids have no effect.

After an incoming packet has had its ethernet type field re-mapped it is subject to being blocked from the `block` option.

This method of changing ethernet types allows different mappings for each instantiation of ETHPKT. If this is not required, the *x*-kernel protocols tables could be changed to achieve the same result.

Example `graph.comp` and `rom` files for using ETHPKT can be found in `/usr/xkernel/user_level/build/Template/example_ethpkt`.

AUTHOR

Mason Katz

B.2 IRIXETH

NAME

IRIXETH (Raw Ethernet Driver (IRIX platform))

SPECIFICATION

IRIXETH is a user-space *x*-kernel ethernet driver that sends and receives messages using IRIX raw sockets.

SYNOPSIS

IRIXETH places and receives packets directly on the wire using the SGI raw socket interface. Using raw sockets is a priviledged operation, so the user must be root or the running xkernel must be owned by root and have the suid bit set.

REALM

IRIXETH is in the ANCHOR realm, supporting the ethernet driver interface described in ETH.

PARTICIPANTS

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

CONTROL OPERATIONS

`MAC_REGISTER_ARP`: Used by an ARP instantiation to register itself with its corresponding SIMETH driver. IRIXETH has no need of this and simply consumes the control operation.

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

Output: none

`MAC_DUMP_STATS`: If IRIXETH or PACKET_STATS have been defined when the module is compiled (the default), this causes the driver to print out relevant statistics such as packets sent and received, broadcasts sent, errors, etc.

Input: none

Output: none

EXTERNAL INTERFACE

IRIXETH adheres to the external interface defined by ETH.

CONFIGURATION

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

IRIXETH recognizes the following ROM options:

`irixeth mmmm nnnn`: This instantiation of irixeth should use IRIX raw socket send port mmmm and receive port nnnn. There must be such a line for each instantiation of IRIXETH in the *x*-kernel.

AUTHORS

Jim Doyle

B.3 IRIXFDDI

NAME

IRIXFDDI (Raw FDDI Driver (IRIX platform))

SPECIFICATION

IRIXFDDI is a user-space *x*-kernel FDDI driver that sends and receives messages using IRIX raw sockets.

SYNOPSIS

IRIXFDDI places and receives packets directly on the wire using the SGI raw socket interface. Using raw sockets is a priviledged operation, so the user must be root or the running xkernel must be owned by root and have the suid bit set.

REALM

IRIXFDDI is in the ANCHOR realm, supporting the FDDI driver interface described in FDDI.

PARTICIPANTS

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

CONTROL OPERATIONS

`MAC_REGISTER_ARP`: Used by an ARP instantiation to register itself with its corresponding SIMFDDI driver. IRIXFDDI has no need of this and simply consumes the control operation.

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

Output: none

`MAC_DUMP_STATS`: If IRIXFDDI_STATS or PACKET_STATS have been defined when the module is compiled (the default), this causes the driver to print out relevant statistics such as packets sent and received, broadcasts sent, errors, etc.

Input: none

Output: none

EXTERNAL INTERFACE

IRIXFDDI adheres to the external interface defined by FDDI.

CONFIGURATION

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

IRIXFDDI recognizes the following ROM options:

`irixfddi mmmm nnnn`: This instantiation of irixfddi should use IRIX raw socket send port mmmm and receive port nnnn. There must be such a line for each instantiation of IRIXFDDI in the *x*-kernel.

AUTHORS

David Yates and Erich Nahum

B.4 SIMETH

NAME

SIMETH (Simulated Ethernet Driver (SunOS, Solaris, OSF/1, Linux, and IRIX platforms))

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 simulated *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 "native" 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.)

REALM

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

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 adheres to the external interface defined by ETH.

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 (sunos platform), Erich Nahum, David Yates, and Jim Doyle (irix platform).

B.5 SIMFDDI

NAME

SIMFDDI (Simulated FDDI Driver (IRIX platform))

SPECIFICATION

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

SYNOPSIS

Each instantiation of SIMFDDI is associated with a specific Unix UDP port and simulates an FDDI driver for a single interface. SIMFDDI transmits outgoing messages by sending to other UDP ports and presents UDP messages received on its port as incoming FDDI packets. Note that since messages sent from one IRIX *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 "native" peers is not possible with this driver.

The mapping between Unix UDP ports and SIMFDDI fddi addresses is very simple. The six bytes of SIMFDDI fddi 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 SIMFDDI 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 SIMFDDI, each with its own UDP port, to simulate a multihomed host.

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

REALM

SIMFDDI is in the ANCHOR realm, supporting the FDDI driver interface described in FDDI.

PARTICIPANTS

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

CONTROL OPERATIONS

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

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

Output: none

`MAC_DUMP_STATS`**:** If SIMFDDI_STATS or PACKET_STATS have been defined when the module is compiled (the default), this causes the driver to print out relevant statistics such as packets sent and received, broadcasts sent, errors, etc.

Input: none

Output: none

EXTERNAL INTERFACE

SIMFDDI adheres to the external interface defined by FDDI.

CONFIGURATION

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

SIMFDDI recognizes the following ROM options:

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

AUTHORS

David Yates and Erich Nahum

Index