x-Kernel Programmer's Manual
(Version 3.2)

Edwin F. Menze III
Hilarie K. Orman [*]

January 23, 1996

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 several environments: as a user program on a Sun SparcStation running SunOS4.1, as a user program on SGI systems running IRIX 4.x or 5.x, as a task on a Mach3.0 system (on some platforms), within the Mach3.0 microkernel (on some platforms), or as a standalone kernel on the DEC Alpha. Protocols can be moved among the different environments without modification. This document assumes that the reader is already familiar with the *x*-kernel's object-oriented infrastructure for implementing protocols.

[*]This work supported in part by National Science Foundation Grants CCR-8811423, IRI-9015407, and CCR-9102040, Sun Microsystems Inc., Hewlett-Packard Company, DARPA Contract DABT63-91-C-0030, and NCSC University Research Grant MDA904-92-C-515

Contents

1 Introduction

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

Version 3.2 is a *protocol implementation environment* that can be embedded in any platform, including both hardware and software platforms. By defining all the interfaces between the protocol and the host environment, the *x*-kernel completely isolates the protocol from the underlying operating system. As a result, protocol source code can be moved from one platform to another without modification.

This manual covers the *x*-kernel functionality, the protocol library, and the tools for building and running *x*-kernels on various platforms. Sections 2–9 define the uniform protocol interface and the various libraries that make up the *x*-kernel. Sections 10 and 11 give additional guidelines about using these routines. Finally, Sections 12–17 describe procedures for maintaining, configuring, running, and installing the *x*-kernel.

1.1 Acknowledgements

The *x*-kernel was originally the concept of Norman C. Hutchinson at the University of British Columbia and Larry L. Peterson of the University of Arizona. Several people at the University of Arizona have made contributions to the current release, including Peter Druschel, Hasnain Karampurwala, Ed Menze, David Mosberger-Tang, Sean W. O'Malley, and Hilarie Orman. The Alpha standalone system was developed by David Mosberger-Tang as a starting point for the Scout operating system. The Irix port was done at the University of Massachusetts by Erich Nahum and David Yates. Many people have contributed protocols to the *x*-kernel as noted in appendix A.

1.2 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
Tucson, AZ 85721

We can be reached by electronic mail at:

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

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

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.

1.3 Copyright Notice

2 Uniform Protocol Interface

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 XObj

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

```
typedef struct xobj {
    XObjType          type;
    char              *fullName;    /* the name given in graph.comp, e.g., "ethdrv/SE0" */
    char              *name;        /* the protocol name, e.g., "ethdrv" */
    char              *instName;    /* the instance name, e.g., "SE0" */
    char              *state;       /* Readable/writable – Owned by the session/protocol */
    Bind              binding;      /* Readable/writable */
    int               rcnt;         /* readable */
    int               id;           /* readable */
    /* Interface functions */
    XOpenFunc         open;         /* protocols only */
    XCloseFunc        closedone;    /* protocols only */
    XOpenEnableFunc   openenable;   /* protocols only */
    XOpenDisableFunc  opendisable;  /* protocols only */
    XOpenDoneFunc     opendone;     /* protocols only */
    XDemuxFunc        demux;        /* protocols only */
    XCallDemuxFunc    calldemux;    /* protocols only */
    XShutDownFunc     shutdown;     /* protocols only */
    XPopFunc          pop;          /* sessions only */
    XCallPopFunc      callpop;      /* sessions only */
    XPushFunc         push;         /* sessions only */
    XCallFunc         call;         /* sessions only */
    XCloseFunc        close;        /* sessions or protocols */
    XControlFunc      control;      /* sessions or protocols */
    XDuplicateFunc    duplicate;    /* sessions only */
    int               numdown;      /* readable */
                      /* down info is owned by the protocol for its sessions */
    int               downlistsz;
    unsigned int      idle;
    struct xobj       *down[8];
    struct xobj       **downlist;
    struct xobj       *myprotl;     /* readable – a session is an instance of this protocol */
    struct xobj       *up;          /* for sessions only */
```

```
    struct xobj    *hlpType;
} *XObj;
```

If you think of the *x*-kernel as implementing protocol and session graphs, then each *XObj* represents a node in one of the graphs. Furthermore, an object's *down* vector represents graph edges; they contain pointers to the *XObjs* that are below the object in the protocol or session graphs. The fields *myprotl* and *up* link a session to the protocols that own and created the session, respectively.

2.1.2 Enable Objects

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

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

2.1.3 Return Values

Most routines have a return value type of *xkern_return_t*, which is either *XK_SUCCESS* or *XK_FAILURE*. Routines that return type *XObj* have a failure value of *ERR_XOBJ*. Some message handling routines use type *xmsg_handle*; see section 2.2.6. Severe error conditions will result in console error messages and the termination of the *x*-kernel.

2.1.4 Function Types

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

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

2.2 Protocol-Protocol Operations

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

2.2.1 xOpen

The *xOpen* function is used by high-level protocol *hlpRcv* to actively open a session associated with low-level protocol *llp* on behalf of high-level protocol *hlpType*. Typically, *hlpRcv* and *hlpType* refer to the same protocol (see section 2.5.2.) The *participants* argument is a list of addresses for each participant in the communication. For this, and all calls returning type *XObj*, a return value of *ERR_XOBJ* indicates failure, and this must be checked by all callers before using the return value.

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

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

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

2.2.2 xOpenEnable

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

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

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

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

2.2.3 xOpenDisable

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

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

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

2.2.4 xOpenDisableAll

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

Generic: *xkern_return_t* **xOpenDisableAll** (*XObj* hlpRcv, *XObj* llp)

Specific: *xkern_return_t* **udp_opendisableall** (*XObj* self, *XObj* hlpRcv)

2.2.5 xOpenDone

Used by low-level protocol to inform a high-level protocol (*hlpRcv*) that a session (*s*) has now been created corresponding to an earlier *xOpenEnable* on behalf of *hlpType*. The "up" pointer of the session is set to the *hlpRcv* protocol. This is useful for virtual protocols; this operation can create a path that bypasses the virtual protocol.

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

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

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

2.2.6 xPush

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

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

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

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

2.2.7 xCall

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

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

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

2.2.8 xPop

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

The *hdr* argument is passed through the infrastructure 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: *xkern_return_t* **xPop** (*XObj* s, *XObj* lls, *Msg* *msg, *void* * hdr)

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

2.2.9 xCallPop

When a synchronous (RPC-like) protocol will be 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: *xkern_return_t* **xCallPop** (*XObj* s, *XObj* lls, *Msg* *msg, *void* * hdr, *Msg* *rmsg)

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

2.2.10 xCloseDone

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

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

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

2.2.11 xClose

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

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

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

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

2.2.12 xControl

Used by one *XObj* to act upon another *XObj* for retrieving information or for setting processing parameters. The operation code *opcode* identifies the action; *buf* is a character buffer from which an argument is retrieved and/or into which a result is placed; and *len* is the length of the buffer. The value returned is the length of the valid data in the buffer; the value -1 is used to indicate failure.
There are two "classes" of operations: standard ones that may be implemented by more than one protocol, and protocol-specific ones. A full discussion of control operation codes is in section 10.

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

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

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

2.2.13 xDemux

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

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

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

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

2.2.14 xCallDemux

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

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

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

2.2.15 xDuplicate

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

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

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

2.2.16 xShutDown

Called once for each protocol object to indicate imminent death of the *x*-kernel. A protocol may choose to take actions to shut down in a graceful manner. In particular, boundary protocols may choose to sever their connections to the outside world in an orderly manner on receipt of this call.

Protocols should generally not invoke this operation on other protocols or on themselves. It is typically invoked by the *x*-kernel infrastructure itself.

Generic: *xkern_return_t* **xShutDown** (*XObj* protl)

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

2.3 Graph Manipulation Operations

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

2.3.1 xCreateSessn

Called by 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 *hlpRcv*, *hlpType* and *llp*. The new session's up pointer is set to *hlpRcv* (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 both *downc* and *downv.* The initialization function pointer *f* may be null; otherwise this function should fill in the interface function pointers in the *XObj* structure. These pointers are initialized to default (usually null) functions by the system initialization code.

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

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

2.3.2 xCreateProtl

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

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

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

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

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

2.3.3 xDestroy

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

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

2.3.4 xGetProtlByName

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

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

2.3.5 xSetDown

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

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

2.3.6 xGetDown

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

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

2.3.7 xSetUp

Resets the up pointer of session *s* 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** (*XObj* s, *XObj* hlp)

2.3.8 xGetUp

Returns the up pointer of session *s*.

XObj **xGetUp** (*XObj* s)

2.3.9 xHlpType

Returns the *hlpType* argument that was used to create session *s*.

XObj **xHlpType** (*XObj* s)

2.4 Utility Operations

2.4.1 xIsProtocol

Returns true if the *XObj* is a protocol.

bool **xIsProtocol** (*XObj*)

2.4.2 xIsSession

Returns true if the *XObj* is a session.

bool **xIsSession** (*XObj*)

2.4.3 xIsXObj

Returns true if the *XObj* is either a session or a protocol (i.e., if this returns false the *XObj* was either never initialized or has been badly clobbered.)

bool **xIsXObj** (*XObj*)

2.4.4 xIsValidXObj

XObjects created with xCreateSessn or xCreateProtl are kept in a system map and removed when the object is destroyed. *xIsValidXObj* can be used to determine whether a random XObject handle is in this map and can thus be used safely.

bool **xIsValidXObj** (*XObj*)

2.4.5 xPrintXObj

Displays some information about the state of the *XObj*.

bool **xPrintXObj** (*XObj*)

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 xCreateProtocol for each protocol configured into the kernel (see section 12). xCreateProtocol, in turn, calls the protocol's init routine, where for a protocol named 'yap', this initialization routine must be named 'yap_init'. Generally, this routine does the following work.

```
void
yap_init(self)
    XObj self;
{
    /* fill in protocol-specific operations */
    self->open = yap_open;
    self->demux = yap_demux;
    . . .

    /* create and initialize protocol-specific state, including maps */
    self->state = xMalloc( sizeof(struct yap_state) );
    . . .

    /* get handle for, and register with, lower-level protocol */
    llp = xGetDown(self, 0));
    partInit(&part, 1);
    partPush(part, ANY_HOST, 0);
    xOpenEnable(self, self, llp, &part);
}
```

2.5.2 hlpRcv and hlpType

The operations *xOpen*, *xOpenEnable*, and *xOpenDisable* take two high-level protocols, *hlpRcv* and *hlpType*. *hlpRcv* is the protocol to which the new lower session should route incoming messages. The lower protocol uses *hlpRcv* as the hlp argument to *xCreateSessn*. The lower protocol uses *hlpType* to determine which messages the new session should handle. For example, when *ethOpen* is called with IP as the *hlpType*, ETH knows that the new session will deal with packets that have the IP ethernet type. The lower protocol typically determines the number that corresponds to *hlpType* by using it in a call to *relProtNum* (see section 13.1.)

Most protocols use their *self* pointer as both *hlpRcv* and *hlpType* when making these calls, though 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. Our experience is that protocols can be classified into different categories, which we call realms. Chances are, any protocol you write falls into one of the following realms. In some cases, the realm 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 one may use are symmetric enhances one's 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 (see appendix C.) 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 7.5.3.

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 *hlpRcv*. This allows arbitrary chains of virtual protocols to insert their sessions between the upper and lower conventional sessions while still passing "type information" from the upper protocol to the lower protocol.

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

2.6 Default Operations

Since many protocols' UPI operations look very similar, the *x*-kernel provides some library operations which do much of the standard work of some of the operations. Many protocols can call these default operations and greatly simplify their own implementations of these routines.

2.6.1 defaultOpenEnable

Binds *key* to an enable object with *hlpRcv* and *hlpType*. If a previous binding exists for the given key and *hlp*s, 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 *hlp*s.

xkern_return_t **defaultOpenEnable** (*Map* m, *XObj* hlpRcv, *XObj* 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 *hlp*s don't match the saved values in the enable object.)

xkern_return_t **defaultOpenDisable** (*Map* m, *XObj* hlpRcv, *XObj* hlpType, *void* *key)

2.6.3 defaultOpenDisableAll

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

xkern_return_t **defaultOpenDisableAll** (*Map* m, *XObj* hlpRcv, *DisableAllFunc* f)

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 *p*, causing the lower protocols to deliver packets for *hlpType* to the virtual protocol *self*. If any of these *xOpenEnable*s fail, *defaultVirtualOpenEnable* backs out of the entire operation.

Assumes that the passive map is keyed on *hlpType*.

xkern_return_t **defaultVirtualOpenEnable** (*XObj* self, *Map* m, *XObj* hlpRcv, *XObj* hlpType, *XObj* *llp, *Part* *p)

2.6.5 defaultVirtualOpenDisable

Undoes the effect of a previous *defaultVirtualOpenEnable*.

xkern_return_t **defaultVirtualOpenDisable** (*XObj* self, *Map* m, *XObj* hlpRcv, *XObj* hlpType, *XObj* *llp, *Part* *p)

2.6.6 Usage

Here is an example of how a protocol might simplify its *enable/disable* routines using these operations:

```
static xkern_return_t
yapOpenEnable( XObj self, XObj hlpRcv, XObj hlpType, Part *p )
{
    long    key;
    PState  *ps = self->state;

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

static xkern_return_t
yapOpenDisable( XObj self, XObj hlpRcv, XObj hlpType, Part *p )
{
    long    key;
    PState  *ps = self->state;
```

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

static xkern_return_t
yapOpenDisableAll( XObj self, XObj hlpRcv )
{
    PState  *ps = self->state;

    return defaultOpenDisableAll(ps->passive_map, hlpRcv, 0);
}
```

3 Participant Library

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

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

3.1 Type Definitions

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

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

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

3.2 Participant List Operations

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

3.2.1 partInit

Initialize participant list *p* of *N* participants.

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

3.2.2 partPush

Pushes address *addr*, pointing to *len* bytes, onto the stack of participant *p*. 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 3.4 below.

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

3.2.3 partPop

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

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

3.2.4 partStackTopByteLen

Returns the number of bytes pointed to by the top element of the stack of participant *p*. 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* p)

3.2.5 partLen

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

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

3.2.6 partExternalize

Takes a participant list *p* and externalizes it into *buf* in a form appropriate for transmission across a protection boundary, essentially dereferencing all the pointers except special-value pointers pushed with length == 0. The *int* pointed to by *len* holds the size of the buffer going in and holds the number of bytes written to the buffer on return. If the buffer is not large enough, *partExternalize* will fail.

xkern_return_t **partExternalize** (*Part* * pList, *void* * buf, *int* * len)

3.2.7 partInternalize

Takes an externalized participant buffer *buf* resulting from *partExternalize* and recreates the participants in the uninitialized participant list *p*.

void **partInternalize** (*Part* * p, *void* * buf)

3.2.8 partExtLen

Reports the number of participants in an externalized participant buffer *buf* resulting from *partExternalize*. This can be useful for allocating the appropriate sized participant list to pass to *partInternalize*.

int **partExtLen**(*void* * buf)

3.2.9 GETPARTICIPANTS

The participant list used to create a session can be retrieved via the GETPARTICIPANTS control operation. See section 10.1.2.

3.3 Relative Protocol Numbers

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

Version 3.2 of the *x*-kernel reconciles these two approaches by maintaining a table of relative protocol numbers. (See Section 13.1 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:

long **relProtNum**(*XObj* hlp, *XObj* 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

long **protTblGetId**(*char* *protocolName)

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

long **relProtNumById**(*long* hlpId, *XObj* llp)

which has the same semantics as *relProtNum*, except that the high-level protocol is identified by its ID number rather than by the XObject itself. This interface can be useful when one needs to determine relative protocol numbers, but does not have the appropriate XObjects in scope.

3.4 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.

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

The following example 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.

```
{
    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);
    ...
}


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

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

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 appendix A.

4 Message Library

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

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

4.1 Type Definitions

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

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

4.2 Constructor/Destructor Operations

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

Messages that are newly created "own" the stack space and can write into that space efficiently using "msgPush". See the section on Usage (4.4) for more information about stacks.

4.2.1 msgConstructEmpty

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

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

4.2.2 msgConstructBuffer

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

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

4.2.3 msgConstructAllocate

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

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

4.2.4 msgConstructCopy

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

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

4.2.5 msgAssign

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

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

4.2.6 msgConstructInplace

The uninitialized message *this* 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 (e.g. within device drivers); it might not be compatible with later implementations of the message library that will provide highly efficient buffer management.

void **msgConstructInplace** (*Msg** this, *char* *buffer, *long* length, *Pfv* freefunc);

4.2.7 msgConstructAppend

The uninitialized message *this* is initialized to prepare for appending data with *msgAppend*. The message can be used in any message operation, but *msgPush* will cause additional space to be allocated. A pointer to the beginning of the new message buffer is returned in the *buffer* parameter.

void **msgConstructAppend** (*Msg** this, *long* length, *char* **buffer)

4.2.8 msgDestroy

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

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

4.3 Manipulation Operations

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

4.3.1 msgLen

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

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

4.3.2 msgTruncate

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

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

4.3.3 msgChopOff

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

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

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

4.3.4 msgJoin

Assigns (in the same sense as *msgAssign*) to *this* message the concatenation of messages *m1* to the front of *m2*. This operation is used to reassemble fragments into a larger message. The first argument must be a valid message. The arguments need not refer to distinct messages.

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

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

4.3.5 msgPush

Prepends data pointed to by *hdr* to the front of message *this*. The supplied function *store* will be called with the *hdr* pointer, a pointer to the message data area where the header data can be copied, the *hdrLen* value indicating the length of the data, and the *arg* pointer. The last argument is not interpreted by the *msgPush* function; its semantics are determined entirely by the user store function.

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

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

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

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

4.3.6 msgPop

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

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

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

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

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

4.3.7 msgPopDiscard

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

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

4.3.8 msgAppend

Appends data pointed to by *tail* to the end of message *this*. The supplied function *store* will be called with the *tail* pointer, a pointer to the message data area where the data can be copied, the *tailLen* value indicating the length in bytes fo the data, and the *arg* pointer. If it is necessary to allocate new space to hold the data, the *newlength* parameter designates the minimum amount of space for the allocation. See *msgConstructAppend*, also.

void **msgAppend** (*Msg** this, *MStoreFun* store, *void* *tail, *long* tailLen, *void* *arg, *long* newlength);

4.3.9 msgPeek

Used to examine (and possibly remove) a header from the front of a message. Returns a pointer to a buffer of *len* bytes that contains the data at the front of message *msg*. When *discard* is TRUE, the operation has the side-effect of removing *len* bytes from the front of the message; otherwise, a pointer to the front of the message is returned, but the message remains unchanged.

xkern_return_t **msgPeek** (*Msg* *msg, *long* len, *bool* *discard);

4.3.10 msgPoke

Used to add a header to the front of a message. Returns a pointer to contiguous buffer of *len* bytes that is logically attached to the front of message *msg*. Typically, a header is then copied into this buffer.

xkern_return_t **msgPoke** (*Msg* *msg, *long* len);

xkern_return_t **msgSetAttr** (*Msg* *this, *int* name, *void* *attribute, *int* len);

4.3.11 msgSetAttr

Associates an *attribute* of *len* bytes with *name* and attaches it to message *this*. 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.

xkern_return_t **msgSetAttr** (*Msg* *this, *int* name, *void* *attribute, *int* len);

4.3.12 msgGetAttr

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

*void ** **msgGetAttr** (*Msg* *this, *int* name);

4.3.13 msgForEach

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

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

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

4.4 Usage Rules

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

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 the *msgAssign* or *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, msgPop*, 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-adress space environment.

5 Event Library

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

5.1 Type Definitions

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

5.2 Operations

5.2.1 evSchedule

Schedules an event that executes function *f* with argument *a* after delay *t* usec; *t* may equal 0. A handle to the event is returned, and this can be used to cancel the event at some later time. When an event fires, a new thread is created to run function *f*. Note that even after an event fires and a thread had been scheduled to handle 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* f, *void** a, *int* t)
typedef void (* **EvFunc**) (*Event*, *void* *)

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

5.2.2 evDetach

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

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

5.2.3 evCancel

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

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

5.2.4 evIsCancelled

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

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

5.2.5 evDump

Displays a *ps*-style listing of *x*-kernel threads in *x*-kernel's 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 follows:

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


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

5.3.2 Cancellable Events

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

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


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

5.3.3 Event Granularity

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

6 Map Library

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

6.1 Type Definitions

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

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

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

6.2 Operations

6.2.1 mapCreate

Creates a map with *nEntries* elements in it. External ids bound in this map are *keySize* bytes long. The maximum value for the key size is MAX_MAP_KEY_SIZE, currently 100 bytes. Programmers will normally use sizeof(structuretype) as the keysize to facilitate platform independence. Note that maps never overflow, but they perform best if *nEntries* is chosen so that the map is at most 50-80% full.

If the *keySize* parameter is any negative number, the resulting map will support variable length keys. The functions *mapVarBind* and *mapVarResolve* should be used with such a map.

Returns 0 if the map could not be created.

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

6.2.2 mapResolve

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

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

6.2.3 mapVarResolve

This is almost identical to *mapResolve*. Note that the resolution will be done with *len* bytes of what the *ext* identifier points to, where *len* is a parameter to this call. The map used in this call must have been created with a negative key size (see section 6.2.1).

xkern_return_t **mapVarResolve** (*Map* Map, *void* *ext, *int* len, *void* **ptr);

6.2.4 mapBind

Adds a binding of external id *ext* to internal id *intern* to map *Map*. Note that the binding will be done with *keysize* bytes of what the *ext* identifier points to, where *keysize* is the parameter that was used in the *mapCreate* call. The return value uniquely identifies this binding; it can later be given as an argument to *mapRemoveBinding*. A return value of ERR_BIND indicates that the external identifier is already bound in the map to a different internal id. If the external identifier is already bound to the same internal id, that binding is returned.

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

6.2.5 mapVarBind

This is almost identical to *mapBind*. Note that the binding will be done with *len* bytes of what the *ext* identifier points to, where *len* is a parameter to this call. The map used in this call must have been created with a negative key size (see section 6.2.1).

Bind **mapVarBind** (*Map* Map, *void* *ext, *int* len, *void* * intern);

6.2.6 mapRemoveBinding

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

3.1 Compatibility note: Same as 3.1 *mapunbindbinding*.

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

6.2.7 mapUnbind

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

3.1 Compatibility note: Same as 3.1 mapremove.

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

6.2.8 mapClose

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

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

6.2.9 mapForEach

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

If the flag *MFE_REMOVE* is set in the return value, *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 recommended way to remove the "current" map entry during a *mapForEach* operation. If the user function tries to remove the "current" entry directly (via mapRemove or mapUnbind), *mapForEach* may fail to find all of the map entries.

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. A return value of 0 negates both flags.

The order in which keys are returned depends on the internal structure of the map.

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

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

6.3 Usage Rules

6.3.1 External Keys

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

6.3.2 Active and Passive Maps

Protocols generally maintain two maps: an active map and a passive map. Active maps are used to map keys found in incoming messages into the session that will process the message. Thus, the active map holds information about the set of currently active connections. Passive maps are used to bind identifiers to *Enable* objects (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 *open* routine, and a passive key to an enable object in its *openEnable* routine. These bindings are then used in the protocol's *demux* operation, according to the general pattern illustrated in figure 1.

Figure 1: Example of using maps in demux

```
static xkern_return_t
yap_demux(self, lls, msg)
    XObj self;
    XObj lls;
    Msg *msg;
{
    Enable  *e;

    /* get protocol-specific state from self */
    pstate = self->state;

    /* extract the header from the message */
    msgPop(msg, yapHdrLoad, &hdr, HLEN, 0);

    /* make an active key from fields in the header (and other info) */
    /* then see if it is in active map; if found, call pop */
    activeid.localport = hdr.dstport;
    activeid.remoteport = hdr.srcport;
    activeid.lls = lls;
    if (mapResolve(pstate->activemap, &activeid, &s) != XK_FAILURE)
    {
        xkr = xPop(s, msg, lls, &hdr);
        return xkr;
    }

    /* make a passive key from fields in the header (and other info) */
    /* then look for it in the passive map */
    /* if found, create a new session, */
    /* save a binding to it in the active map, */
    /* and call opendone and pop */
    passiveid = hdr.dstport;
    if (mapResolve(pstate->passivemap, &passiveid, &e) != XK_FAILURE)
    {
        s = yapCreateSessn(self, e->hlpRcv, e->hlpType, &activeid);
        xDuplicate(lls);
        s->binding = mapBind(pstate->activemap, &activeid, (int)s);
        xOpenDone(e->hlpRcv, s, self);
        xkr = xPop(s, lls, msg, &hdr);
        return xkr;
   }

    /* don't know how to process this message, so drop it */
    return;
}
```

Notice two things about the example. First, in the case where a new session is created, it is necessary to call *xDuplicate* on the lower-level session to make the reference we hold to it permanent (see Sections 2.2.15 and 11). Second, it is useful to save the binding of an active key to the newly created session in the session object itself. (The same thing is done when a session is created in the protocol's *open* routine). This is because it will be necessary to be able to remove the binding from the active map when the session is closed, and this is often easier or more efficient if the binding itself is available. Finally, the active key in this example includes both fields found in the message header and the lower-level session that called xDemux. Using the lower-level session as part of the active key is fairly common; it helps to disambiguate protocol header fields that are not unique across all lower-level connections.

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

7.2 Synchronization Operations

7.2.1 semInit

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

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

7.2.2 semWait

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

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

7.2.3 semSignal

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

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

7.3 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* is in microseconds.

void **Delay** (*int* msec)

7.4 Locking Operations

The *x*-kernel provides a readers/writers lock synchronization type, *ReadWriteLock*. This type is currently built on top of the *x*-kernel *Semaphore* type. Although originally developed for the machNetIPC protocols, the lock can be used for more general synchronization. The *ReadWriteLock* implementation favors writers (i.e., if queues of readers and writers both exist, writers will always be favored.)

Note that we expect these locking operations to be used in a multiprocessor environment, although we have not yet used them for this purpose. Protocol developers working in an MP environment should attempt to adhere to this interface. However, we consider locking on an MP to be an unresolved issue, and so we are eager to hear about experiences using these operations in such an environment.

7.4.1 rwLockInit

Initializes the lock for subsequent operations.

void **rwLockInit** (*ReadWriteLock* *)

7.4.2 rwLockDestroy

All threads waiting on this lock will be released and their locking operations will return *XK_FAILURE*. When both the reader and writer queue are empty, the user-supplied RwlDestroyFunc will be called with a pointer to the lock and the *void** argument.

void **rwLockDestroy** (*ReadWriteLock* *, *RwlDestroyFunc*, *void* *)

typedef void (***RwlDestroyFunc**) (*ReadWriteLock* *, *void* *);

7.4.3 readerLock

Acquires a reader lock. Many readers may hold the lock at one time, but no writers will acquire the lock while at least one reader holds the lock. Will return *XK_FAILURE* if the lock was destroyed before the lock was acquired.

xkern_return_t **readerLock** (*ReadWriteLock* *)

7.4.4 readerUnlock

Releases a reader lock. It is an error to call *readerUnlock* without holding a reader lock.

void **readerUnlock** (*ReadWriteLock* *)

7.4.5 writerLock

Acquires a writer lock. While a writer lock is held, no other reader or writer may acquire the lock. Will return *XK_FAILURE* if the lock was destroyed before the lock was acquired.

xkern_return_t **writerLock** (*ReadWriteLock* *)

7.4.6 writerUnlock

Releases a writer lock. It is an error to call *writerUnlock* without holding a writer lock.

void **writerUnlock** (*ReadWriteLock* *)

7.5 Usage Rules

7.5.1 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-protocol operations are considered to have the potential to cause a thread switch, and all data structures must be "secured" before calling such operations.

7.5.2 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 (message) not be able to proceed, it should put the message in a protocol-dependent queue and return. Later, another thread can pick the message up out of the queue and continue processing it.

For example, when an incoming thread/message arrives in IP and discovers that it is just one fragment of a larger datagram, rather than blocking the thread and 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.

7.5.3 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 interface, 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.

7.5.4 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.

7.5.5 Multiprocessor Support

To date, we do not have significant experience running the *x*-kernel on a multiprocessor. We have introduced locking operations that are useful in certain situations on a uniprocessor, and while these operations might extend to an MP environment, we cannot be certain that they will. Moreover, we certainly expect that the implementation of both the locking and the semaphore operations will need to change in an MP environment, even if the interface remains the same.

It should be noted that the *x*-kernel, as currently implemented, 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. Our experience suggests, however, that this implementation is reasonably efficient for a small number of processors.

8 Trace Library

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

8.1 Type Definitions

The current value of the tracing variable *tracevar* is used to control whether or not a particular trace operation takes place. The trace variable values can be set at system build time; see section 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_ERROR	mild warnings
TR_MAJOR_EVENTS	open, close, etc.
TR_GROSS_EVENTS	the coarsest tracing level
TR_ERRORS	serious non-fatal errors; some residual event traces
TR_ALWAYS	normally only used during protocol development

8.2 Operations

8.2.1 xTrace

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

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

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

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

For example:

```
int traceudpp;

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

8.2.2 xTraceP

The *xTracePn* macros function much the same way as the *xTracen* macros, except that they take an XObject as their first parameter (instead of a trace variable) and they print the protocol instance name of the XObject 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* macros whenever you have an appropriate XObject in scope, using the *xTracen* macros only when there is no such XObject available.

xTraceP*n*(*XObj*, *int* tracelevel, *char* *formatstring, args, ...)

8.2.3 xIfTrace, xIfTraceP

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();
```

There is also the analogous operation taking an XObject instead of a trace variable:

xIfTraceP (*XObj*, *int* tracelevel)

8.2.4 xAssert

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

xAssert (*bool* exp)

8.2.5 xError

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

xError (*char* * ErrorString)

8.3 Usage Rules

Trace and assert statements are macros which are only active in DEBUG mode (see section 12). In OPTIMIZE mode, debugging and trace statements go away completely. One 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. One should be careful to separate the operation and the check of the return code:

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

If you are writing a new protocol, you should insert trace statements. Even though there will never be bugs left after you release your protocol, it may help others in debugging their protocol which sits on top of, or below, yours. Don't delete these very helpful debugging libraries 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, one would set the TCP trace variable to have the value TR_FULL_TRACE.

9 Utility Routines

9.1 Storage: xMalloc and xFree

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

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

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

9.2 Time

The *x*-kernel uses the same time structure as Unix:

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

9.2.1 xGetTime

Sets **t* to the current time of day:

void **xGetTime** (*XTime* *t)

9.2.2 xAddTime

Sets the *res* to the sum of *t1* and *t2*. Assumes *t1* and *t2* are in standard time format (i.e., does not check for integer overflow of the usec value.)

void **xAddTime** (*XTime* *res, *XTime* t1, *XTime* t2)

9.2.3 xSubTime

Sets the *res* to the difference of *t1* and *t2*. The resulting value may be negative.

void **xSubTime** (*XTime* *res, *XTime* t1, *XTime* t2)

9.3 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)
```

9.4 Checksum

9.4.1 inCkSum

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

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

9.4.2 ocsum

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);

9.5 Strings to Hosts

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

9.5.1 ipHostStr

Returns a string with a "dotted-decimal" representation of IP host *h* (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* *h)

9.5.2 str2ipHost

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

xkern_return_t **str2ipHost** (*IPhost* *h, *char* *str)

9.5.3 ethHostStr

Returns a string with a representation of Ethernet host *h* (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* *h)

9.5.4 str2ethHost

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

xkern_return_t **str2ethHost** (*ETHhost* *h, *char* *str)

9.6 Network Mask Operations

A table of network masks is maintained for use by protocols that handle IP addressing. A mask can be added to the table by specifying the host and the mask. The network mask for the local host can be set a startup time via a ROM option A.

int **netMaskAdd** (*IPhost* *iph, *IPhost* ***newMask**)

Using the host as input, the network mask can be accessed from the table using the finding function. *void* **netMaskFind**(*IPhost* *mask, *IPhost* *host)

The broadcast predicate has the value one if the IP address is a broadcast address, otherwise it returns the value zero. The mask is determined from the table is there is an appropriate entry, otherwise the default forms based on the upper bits of the address are used.

int **netMaskIsBroadcast**(*IPhost* *h)

Comparisons for equality for the networks or subnetworks to which two hosts belong are available:

int **netMaskSubnetsEqual**(*IPhost* *h1, *h2)

int **netMaskNetsEqual**(*IPhost* *h1, *h2)

9.7 Host Name Service

A simple way of mapping host name strings to host IP addresses is provided via rom file entries (see section 13.2) 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.

9.7.1 xk_gethostbyname

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

xkern_return_t **xk_gethostbyname** (*char* *name, *IPhost* *addr);

9.8 ROM file parsing utilities

When writing a protocol that provides user-configurable ROM file options, one 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 xkern_return_t (***XObjRomOptFunc**)
   (*XObj* protl, *char* **fields, *int* numFields, *int* lineNumber, *void* *arg)

The ROM file parsing code will call this handler 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 XObjRomOpt structures which bind option names to their handling functions. There is one XObjRomOpt 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 */
        XObjRomOptFunc  func;       /* Handler function */
} XObjRomOpt;
```

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

   xkern_return_t **findXObjRomOpts**(*XObj*, *XObjRomOpt* *, *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 the XObject (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 of XObjRomOpts is scanned. If the second field of the line matches the name field of one of the XObjRomOpts, or if the name field of one of the XObjRomOpts is the empty string, the XObjRomOptFunc for that option is called with the object, 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 the XObject but none of the supplied RomOpts 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 XObjRomOptFunc 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. This example shows how this protocol would interface with the romopt parsing utilities:

```
static xkern_return_t readMtu( XObj, char **, int, int, void * );
static xkern_return_t readPort( XObj, char **, int, int, void * );


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


static xkern_return_t
readMtu( XObj 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 xkern_return_t
readPort( XObj 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( XObj self )
{
    ...
    findXObjRomOpts(self, opts, sizeof(opts)/sizeof(XObjRomOpt), 0);
    ...
}
```

10 Control Operations

Control operations are used to perform arbitrary operations on protocols and sessions, via the *xControl* operation described in section 2.2.12. *xControl* returns 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 *xControl* 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.

10.1 Standard Control Operations

10.1.1 Protocol and Session Operations

These operations can be performed on both sessions and protocols.

GETMYHOST, GETMYHOSTCOUNT

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

GETMAXPACKET, GETOPTPACKET

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

RESOLVE, RRESOLVE

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

10.1.2 Session Operations

GETPEERHOST, GETPEERHOSTCOUNT

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

GETMYPROTO, GETPEERPROTO

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

GETPARTICIPANTS

> Treats the buffer as a simple array of bytes. The lower session creates a participant list with data that was used to open the session. This participant list is then written into the control buffer using *partExternalize* (see section 3.) The caller will have to run *partInternalize* on the buffer before it can be used as a participant list.
>
> Sessions typically implement this control op by first executing it on their lower session. If their protocol uses information on participant stacks, they then run *partInternalize* on the buffer, push their protocol-specific information on the participant stacks, and then re-externalize the participant list into the buffer.

FREERESOURCES

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

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(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.

10.2 Protocol-Specific Control Operations

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

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

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

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

10.3 Forwarding Control Operations

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

11 Reference Counting XObjs

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

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

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

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

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

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

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

11.1 References

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

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

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

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

11.2 Reference counts

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

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

11.2.1 Counting External References

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

> *xCreateSessn(...)*
>
> > The newly created session has an initial reference count of 0 (it was 1 in v3.1.)
>
> *xOpen(...)*
>
> > The session returned by invoking the lower protocol's open routine has its reference count incremented before it is returned to the caller of *xOpen*, indicating that the caller now has a permanent reference to the session. (*xOpen* did not manipulate reference counts in v3.1.)
>
> *xClose(XObj s)*
>
> > Decrements the reference count of the session, indicating that a permanent reference to the session has been released. The lower protocol's close operation is called *only* if the session's new reference count is zero. (*xClose* always invoked the lower protocol's close operation in v3.1.)
>
> *xDuplicate(XObj s)*
>
> > By default, increments the session's reference count, indicating that a new permanent reference to the session has been created. If the session has its own duplicate function, that is called instead. (*xDuplicate* did not exist in v3.1.)

11.2.2 Counting xPops

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

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

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

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

11.3 Internal vs. External Reference Counts

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

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

11.4 Session Caching Strategies

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

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

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

- Destroy the session if inactive.

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

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

- Garbage collector on a per-session basis

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

```
foo_demux()
        if ( active session does not exist ) {
                if ( openEnable exists ) {
```

```
                        if ( cached session exists ) {
                                evCancel(sstate->gc);
                                sstate->gc = 0;
                        } else {
                                s = foo_open();
                        }
                        xOpenDone(s, ...);
                } else {
                        drop packet; return;
                }
        }
        xPop()

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

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

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

- Garbage collector on a per-protocol basis

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

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

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

foo_demux()
        if ( active session does not exist ) {
                if ( openEnable exists ) {
                        if ( cached session exists ) {
                                move sessn from cached map to
                                active map
                        } else {
                                s = foo_open();
```

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

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

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

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

11.5 Modifying v3.1 protocols

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

- `foo_open` should not increment the reference count

  v3.1 protocols often followed the format:

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

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

```
foo_open()
        if (session s does not exist) {
```

```
                s = xCreateSessn();
                fill in state;
        }
        return s;
```

- `foo_close` should not decrement the reference count

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

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

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

- *xPop* should not be bypassed

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

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

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

12 Configuring a Kernel

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

12.1 Specifying a Protocol Graph

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

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

The first two sections, device drivers and protocols, describe the protocol graph to be configured into your kernel. 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 in the example:

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

The first field gives the protocol's name. The rest of the fields are optional and may occur in any order. The dir and files fields describe the names and locations of the source files used to build the protocol. Files are specified without extensions. These two fields should not be used in the common case where one wants to link in protocol object code from the public system object area. The dir and files fields should only be used when one wants to compile and link code from one's private build area instead. 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.

Some newer protocols have been built with a stub compiler described in the University of Arizona tech report TR 94-26, available through ftp and WWW. The input files in this case have the extension `.usit`, and require the `usit` and `usc` programs as part of the build procedure. To obtain the code (if it is not part of your distribution), use ftp to cs.arizona.edu to access the directory xkernel/usc.

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

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

Multiple instantiations of protocols are supported by using a "/" character after the protocol name to add an extension. In the following example, two instantiations of "yap" are indicated, one over "ip" and one over "eth," and both are used by the "prt" protocol. In this example, the instance suffixes for the "yap" protocol are the names of the protocol below the instance, but this is just a convention – they could have any distinct string as an instance suffix.

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

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

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

```
@;
#
# You can also specify protocol tables to be read in at boot time.
#
prottbl=/usr/xkernel/etc/prottbl;
prottbl=./prottbl.local;
#
#subsystem tracing for messages and protocol operations
#see the include file 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 romfile to be used;
# the data will be read during "make compose" and incorporated
# into the xkernel runtime image
#
romfile=romfile.yap;
#
# 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* which generates startup code to build the protocol graph and set up the described configuration. The protocol graph is built bottom-up; when a protocol's initialization function is called, the lower level-protocols have already been initialized.

12.2 Build Procedure

A given instance of the *x*-kernel is built in a working directory. Each working directory can support one *x*-kernel configuration at a time. Working directories may be organized within the *x*-kernel source tree (e.g., as subdirectories of xkernel/build) or outside of the source tree (e.g., in an *x*-kernel user's home directory.)

For the purpose of the following discussion, we assume the user is configuring a kernel so as to implement and evaluate protocol YAP (yet another protocol.) A user configuring a kernel that contains only existing protocols should ignore all references to YAP.

1. Put xkernel/bin/BINTYPE and xkernel/bin in your search path, where BINTYPE is one of `sunos-sparc`, `mach3-mips`, `mach3-intelx86`, `irix-mips`, or `scout-alpha`. These must be before /bin and /usr/bin. This will allow use of the version of make distributed with the *x*-kernel (GNU make v. 3.66) rather than the standard Unix make.

2. Create a new working directory to hold your kernel.

3. Change to your new working directory.

4. Copy the appropriate Makefile for your platform from xkernel/build/Template into your working directory under the name Makefile:

   cp xkernel/build/Template/Makefile.sunos Makefile

5. Make the Makefile writable:

   chmod u+w Makefile

6. Edit the Makefile. Select an appropriate HOWTOCOMPILE. Generally, you want DEBUG until you are ready to run serious performance tests, then change it to OPTIMIZE. (See Section 8.)

   Check to see that the variable XRT in your Makefile is a path to the root of the xkernel source tree.

7. If you are including a new protocol in your kernel, then create (at least) a shell for that protocol; e.g., create a shell version of yap.c. The protocol shell should define an initialization routine; e.g., yap_init. You will also need to put an entry for your protocol in one of the protocol tables referenced by your graph.comp (see section 13.1.) If you are configuring a kernel that contains only existing protocols, then this step is not necessary.

8. Create a graph.comp in your build directory, as discussed in section 12.1, to specify the protocol graph for your kernel. Example graph.comps are located in the xkernel/build/Template directory.

   In steps 9, 10, and 11, you may see one or more compiler warning messages of the form:

   ```
   make[n]: fopen: [filename]: No such file or directory
   ```

   These messages are expected and can be ignored. The files referenced by these messages will be created in steps 10 and 11.

   The following two steps must be redone if graph.comp is later modified:

9. make compose

10. make depend

11. If you are the first person building a kernel of this configuration at your site (i.e., if you are building a kernel as part of the installation procedure), you will need to run an initialization target for the system areas:

    make system

    This will run the targets *allCompose*, *allDepend*, and *libs* for the configuration and platform indicated by your Makefile, building configuration files, dependency files, and object files for most of the *x*-kernel system. Again, you will get error messages about missing dependency files, but the *allDepend* target will generate the missing files. This will take a long time to run. The installer of the *x*-kernel should run this target for all configurations (DEBUG,OPTIMIZE) and platforms that will be used at the site.

12. make
    Object files will be placed in a subdirectory whose name reflects your configuration and platform (DEBUGsunos-sparc, OPTIMIZEmach3-mips, etc.) This is the same way in which object files are stored throughout the *x*-kernel hierarchy. The final *x*-kernel executable (`xkernel`) will be placed in your build directory.

12.2.1 Scout (Alpha Standalone)

As a standalone kernel, the Scout-based *x*-kernel can't read protocol tables or ROM files at boot time, so the information in these files must be loaded into the kernel during configuration.

Protocol table information is contained in the file `ptblData.c` in your build directory. By default, this file contains the contents of xkernel/etc/prottbl.std. To generate data from other protocol tables, use the `ptbldump` command:

```
ptbldump file1 [ file2 ... ] > ptblData.c
```

To load ROM information into your kernel, specify the name of your ROM file in the third section of graph.comp (see the example in section 12.1.) The `compose` program will read the rom file and dump its data to the file `compose_rom.c` in your build directory.

12.3 Debug versus Optimized Mode

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

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

12.4 Useful Make Targets

The previously mentioned make targets (*compose*, *depend*, *xkernel* (the default)) are the minimal set necessary to build an *x*-kernel. This section describes some additional useful make targets.

Some of these descriptions refer to "*x*-kernel system areas". These areas contain source and object code shared by all users of the *x*-kernel. Each platform has four main system areas:

prot:

> Code for the *x*-kernel protocols. (`/usr/xkernel/protocols`)

pi:

> Platform-independent code common to all platforms. (`/usr/xkernel/pi`)

pxk:

> Platform-specific code including device drivers and application programmer interfaces (APIs). (e.g., `/usr/xkernel/mach3`, `/usr/xkernel/sunos`, etc.)

user:

> Platform-specific user libraries that can be linked with user applications to access the *x*-kernel through one of the *x*-kernel APIs. (e.g., `/usr/xkernel/mach3/user`) Not all platforms have such libraries.

These targets are all executed from a build directory.

prot, pi, pxk, user, libs:

> Rebuilds any out-of-date source code in the indicated area. The target *libs* rebuilds code in the *prot*, *pi*, and *pxk* areas.

clean:

> removes object files in the build directory.

protClean, piClean, pxkClean, userClean, allClean:

> Removes object files, libraries and other generated files from the indicated area. The *allClean* target runs in all areas except the user area.

protCompose, piCompose, pxkCompose, allCompose:

> Rebuilds ComposeFiles lists for the indicated area. These files group object files together so the compose program knows which files to link in for which protocols. These files are generated by the *system* make target during system installation, but they are only updated when these targets are run explicitly. The *allCompose* target runs in all areas except the user area.

protDepend, piDepend, pxkDepend, userDepend, allDepend:

> Rebuilds the dependency lists in the indicated area. The *allDepend* target runs in all areas except the user area.

srcList, hdrList:

> Builds files in the local build directory listing all of the active *x*-kernel source files. These targets will do nothing if the files already exist. To force the files to be rebuilt, use the targets *SrcList* and *HdrList* instead.

grep:

> greps for the target G in all *x*-kernel source and header files produced by the *srcList* and *hdrList* targets.
> Example usage:

```
make grep G="xCreate"
```

tags:

> Creates an emacs tag file `TAGS` in the local build directory for all *x*-kernel source and header files produced by the *srcList* and *hdrList* targets.

12.5 Modifying Code in System Areas

In order to make the common case as fast as possible, the *x*-kernel build procedure normally doesn't check to see if any source code in the system areas needs to be recompiled. If you make a change to any of the system areas, that code will have to be recompiled explicitly from the build area using the targets for that area. For example, if you add a protocol to the system protocol area, in addition to modifying the protocol Makefile you should run these targets:

```
make protDepend
make protCompose
make prot
```

13 Running a Kernel

This section describes the procedure for running an *x*-kernel, including general instructions for setting up a run-time environment for your *x*-kernel, as well as instructions for running an *x*-kernel in specific platforms.

13.1 Protocol Tables

The *x*-kernel runtime environment always includes a protocol table. For each protocol, this table defines a numbering space for identifying other protocols. The *x*-kernel builds a table of protocol numbers by reading configuration files at boot time, and provides an operation for protocols to query this table (see section 3.3). This frees protocols from embedding explicit protocol numbers in the protocol code itself.

If you have not configured any non-standard protocols in your graph, you can safely use one of the default protocol tables in /usr/xkernel/etc and may skip to section 13.1.1.

The following is an example configuration file defining a protocol table:

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

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

```
udp         5
tcp         6
icmp        7
blast       8
sunrpc      9
```

Each protocol has an entry of the form:

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

where the idNumber uniquely identifies each protocol.

There are two ways for a protocol to define its 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 configuration files for building protocol tables, located in /usr/xkernel/etc: prottbl.std and prottbl.nonstd. 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., 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 (see section 12.1 for configuration examples.) If you need to configure your new protocol above an existing protocol which uses explicit numbering, you can augment the table for the existing protocol as in this example:

```
#
# Auxiliary protocol table
#

yap 1000
```

```
ip 2  {
        yap 200
}
```

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

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

13.1.1 ROM options

Protocol table configuration files can be specified in either graph.comp (see section 12) or in a ROM file (see section 13.2). The ROM file format is:

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

In both cases, the named protocol tables are loaded at run-time. Platforms for which loading protocol tables at run-time is impractical (e.g., inside the Mach kernel) provide platform-specific mechanisms to support protocol tables (see, for example, section 17).

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
```

13.2 ROM files

To support run-time options for protocols and various subsystems, the *x*-kernel provides the notion of a ROM file. 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 `#`. As an example:

```
#
# 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 providing ROM files is specific to the individual platforms and is documented for each platform in section 13.

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

13.2.1 ROM File Generators

The "rom" file used during *x*-kernel initialization has a cryptic format that can be difficult specify and maintain manually. The *genrom.sh* and *genrom2.sh* utility programs can ameliorate the problem by using a "rom" file template and expanding it into one more files for use with testing configurations. This is especially helpful for use with the *simeth* (see appendix B) and *simsimeth* (see appendix A) protocols, or in testing configurations where an *ARP* server is not available.

The *genrom.sh* utility generates rom files for the *x*-kernel from prototype files. It includes information about the addresses of hosts specified by name on the command line. For example, using the prototype rom file "romgen" with the three lines:

```
simeth
arp
dns
```

and the command `genrom.sh romgen umb lei`, two rom files are produced, rom.umb.1 and rom.lei.2 with the contents

```
rom.umb.1:
simeth 4724
arp 192.12.69.97 192.12.69.97 4724
arp 192.12.69.116 192.12.69.116 4725
dns umb 192.12.69.97
dns lei 192.12.69.116

rom.lei.2:
simeth 4725
arp 192.12.69.97 192.12.69.97 4724
arp 192.12.69.116 192.12.69.116 4725
dns umb 192.12.69.97
dns lei 192.12.69.116
```

The "dns" lines refer to the simulated IP addresses for *simeth* and *simsimeth* files. See the manual sections on simsimeth and simeth for detailed explanations of the simulated address numbers.

The *genrom2.sh* utility generates two rom files for the *x*-kernel from two prototype rom file files. This is generally used for developing rom files to be used between a machine running simeth and a machine running simsimeth. For example, given two prototype rom files, romgen1 and romgen2,

```
romgen1:
simeth
arp

romgen2:
ethdrv/SE0 priority 200
arp/lower
eth/upper mtu 1440
simsimeth
arp/upper
```

The command `genrom2.sh romgen1 romgen2 umbra moz` produces two rom files, rom.umbra.1 and rom.tch.2 with contents:

```
rom.umbra.1:
simeth 1690
arp 192.12.69.97 192.12.69.97 1690
arp 192.12.69.99 192.12.69.99 1691

rom.tch.2:
ethdrv/SE0 priority 200
arp/lower 192.12.69.97 192.12.69.97 1690
arp/lower 192.12.69.99 192.12.69.99 1691
simsimeth 1691
eth/upper mtu 1440
arp/upper 192.12.69.97 c0:0c:45:61:06:9a
arp/upper 192.12.69.99 c0:0c:45:63:06:9b
```

13.2.2 CAVEATS

Both genrom.sh and genrom2.sh work by running the Unix command "arp" to get the ethernet addresses of the machines. The target machines must be running in order to get this information. If they aren't running, a bogus ethernet number will be used as a substitute. There is a further complication in that you cannot get the ethernet address of the machine you are running on. To overcome this failure, the programs will try to run "arp" on another machine using "rsh". The default machine is "cheltenham" (this default should be changed for sites other than University of Arizona); if you do not have an account on that machine, you will need to set the environment variable ROMGEN_HOST to some other host on your network where you can execute the command `/usr/etc/arp` via the "rsh" command. It may be necessary to establish a `.rhosts` file on the ROMGEN_HOST for this to work properly.

To get the arp addresses into the routing table, each host is probed using "ping". Output from "ping" indicating whether the host is dead or alive will be printed on stdout.

It is important that the arp lines come after the simeth or simsimeth line they are related to. I.e., in the romgen2 file above, the arp/lower line generates arp entries appropriate to a real ethernet because there has been no previous mention of a simeth or simsimeth protocol. After the "simsimeth" line, the next arp line produces entries appropriate to the simsimeth protocol.

The port numbers used for encapsulating traffic sent over UDP are assigned at random. You can override this by assigning a base number in decimal using the environment variable ROMGEN_BASE_PORT.

Normally the simeth arp lines use the same address for the real and simulated host address. If you use the same host twice on the command line (e.g. `genrom.sh romgen umb umb`), then the second arp entry will use a host address of "192.12.69.1", and further redundancies in host names will simply result in incrementing the fourth component of the network address. This might cause clashes with other host names. You can change the base number used for assigning the fourth component using the environment variable ROMGEN_FAKE_IP_BASE.

Another problem occurs when one real host is used to simulate more than one test host. The "dns" lines that are generated must use a simulated host name. This is achieved by attaching a number to hostname, i.e. `genrom romproto umb umb` will result in dns records for "umb" and "umb2". Currently, this is only detected when the hostname occurs consecutively on the command line, i.e.

```
genrom romproto umb ush umb
```

would not be handled correctly; it would be necessary to edit the generated rom files to correct the dns lines for the second occurrence of "umb".

Only the first word on each line is examined in the prototype file; if you have "arp xxx yyyy" or "dns name number" lines, they will be interpreted exactly as if they were simply "arp" or "dns".

13.3 Unix user-space (SunOS/Irix)

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., NIT in SunOS or 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 B for specific configuration instructions.)

13.3.1 Simulated Drivers

Configuring the encapsulated-packet drivers can be a bit more confusing. (We refer to encapsulated-packet drivers as *simulated drivers*, and to *x*-kernels which use them as *simulated x-kernels* or just *simulators*. 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 simulated *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 simulator can only talk to their peers in other simulators; they cannot communicate with "real" versions of those protocols running stand-alone, for example, on a Unix machine.

Since a simulated *x*-kernel uses a connectionless UDP socket as its transport mechanism, more than one simulated *x*-kernel can be run on a single workstation. Because of this flexibility, the local IP address used by each simulator (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 simulated *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 simulated *x*-kernel.

Consider the following example ROM files (as described in section 13.2) for two simulated *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 simulator runs, and the fourth field is the simulator'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 *x*-kernels, each simulator must be configured with an arp entry for each of its peers.

See the manual page in appendix B for more information on configuring the particular simulated driver you are using.

13.3.2 Running

As the result of configuring a kernel (section 12), a file named xkernel should exist in your current directory (/usr/xkernel/build/yap).

While in this directory, create a sub-directory for each configuration you want to test. For example, if you are using a simulated *x*-kernel and want to start up two instances of the *x*-kernel, create two subdirectories, e.g., client and server. In each subdirectory, create a file named rom, containing configuration information as described above and in the man pages for protocols and device driver in appendices A and B.

Each simulated host runs as a separate Unix process. If you will be running multiple *x*-kernels and are using a windowing user interface, then you will probably want to 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 then type ../xkernel. To stop an *x*-kernel, type DELETE or CTRL-C.

13.3.3 SunOS caveats

We recommend building the sunos version of the *x*-kernel with a gcc earlier than 2.0, if possible.

13.3.4 Irix caveats

We recommend using the native IRIX C compiler for the multithreaded debugging support (not yet supported under GDB on IRIX), but a current version of GCC (2.3.3 and above) can also be used successfully. You may need to use GCC to build the utilities (compose etc), depending on which OS version you use.

13.4 Mach 3

As the result of configuring a kernel (section 12), a file named xkernel should exist in your current directory (/usr/xkernel/build/yap). You may optionally create a file named rom in this directory. This is the ROM file described in section 13.2. This is an example rom file for the mach3 platform:

```
ethdrv/SE0 priority 20
arp 192.12.69.88 8:0:20:1:51:E6
arp 192.12.69.89 8:0:20:1:50:40
```

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

The instance name for the ethdrv protocol indicates the Mach device to use as for network access. You can usually learn which device is the default for the machine by looking at the network initialization command (usually “ifconfig”) in the system initialization file in the /etc directory. At our site the devices are SE0 for the decstations and el0 for the PC platforms.

The ethdrv entry indicates the priority of the Mach packet filter used by the *x*-kernel. Priorities > 100 cause the *x*-kernel to grab ethernet packets and not allow the native Mach networking code to see them. This allows the *x*-kernel to fully communicate with other hosts, but makes it difficult to run *x*-kernel tests from a remote host.

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

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

  or

- using the non-standard protocol table (/usr/xkernel/etc/prottbl.nonstd). In this case, of course, the *x*-kernel protocols will only be able to communicate with other hosts running the *x*-kernel with the alternate protocol table.

Since the *x*-kernel reads from the ethernet device, it needs to run with root uid. You will need to either su to root before running it or run it setuid root. Some Unix servers will allow members of group *kmem* to read the device; in this case members of the group need not su to root.

See the man page for the Mach ethernet driver *ethdrv* in appendix A for additional configuration requirements.

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

13.4.1 ROM options

The number of input threads and buffers used by the ethernet drivers in the Mach3 platform can be set with the following ROM file options:

```
shepherd threads N
shepherd buffers M
```

There should be at least as many buffers as input threads.

13.5 Mach 3 *x*-kernel in the Mach kernel

See section 17 for a description of how to build an *x*-kernel within the Mach kernel and how to establish ROM file options for this platform. Be sure to see appendix C for information on configuring the device driver you wish to use.

13.6 Scout (Alpha standalone)

13.6.1 Booting

Currently, the standalone kernel can be booted via network only. The Alpha firmware supports booting from the network via BOOTP (to get the boot parameters) and TFTP (to download the boot file). That is, to boot the standalone kernel, you need to designate some machine (other than the one you want to boot) as your bootserver. This machine does *not* have to be an Alpha. Anything supporting BOOTP and TFTP should work. You'll need to consult the bootserver's system documentation to find out how to setup a boot server. Typically, this involves enabling the `bootpd` and `ftpd` daemons and adding an entry to the `/etc/bootptab` file. For example, we use the following entry to boot one of our Alpha machines:

```
piston.cs.arizona.edu:\
hd=/usr/local/boot:bf=xkernel.boot:ht=ethernet:ha=08002BBAAB3E:\
        hn:vm=rfc1048:ip=192.12.69.168:bs=auto:
```

To boot a kernel, you need to copy the file `xkernel.boot` that was generated during the build process into the bootserver's directory. This directory is defined via the `hd` capability and in the above example is `/usr/local/boot`[1]

Once a bootserver is setup, the Alpha machine can be booted from the console prompt. The boot command syntax is (square brackets indicate optional items):

`boot ez` [`-fl` *bootflags*] [`-fi` `"`*filename* `"`]

To boot the default file (defined via `bf` capability in `/etc/bootptab`), you simply type "`boot ez`." The *bootflags* can be any string of the characters described in the table below:

Flag	Description
`k`	kernel debugger: attaches to GDB
`i`	interactive boot: prompts for *x*-kernel "command" line options
`v`	verbose: prints some configuration options
`a`	reserved for future use
`d`	reserved for future use

The `-fi` option can be used to boot an alternative kernel. Any file in the bootserver's directory can be specified (relative pathname). Notice that the filename *must* be enclosed by double quotes.

Specifying *x*-kernel Command Line Options

After booting a kernel with the `i` bootflag, an *x*-kernel command line string can be entered at the `scout:` prompt. This is useful when running one of the test protocols (see page 215). For example, when running `udptest`, a typical command line is (user input in bold-face):

`scout:` **-testudp -c192.12.69.54 -trips=1000**

This command line requests that `udptest` be run as a client, that the server is running on the machine with IP address 192.12.69.54, and that each test should exchange 1000 messages.

13.6.2 Using GDB

GDB's remote debugging facility can be used to debug a standalone *x*-kernel over a serial line cable. Once GDB is attached to a kernel, debugging works just like debugging any other program. For example, it is possible to interrupt a running kernel, set breakpoints, inspect variables and so on.

For remote debugging, it is necessary to have an additional Alpha machine running OSF/1 (the *host* machine). This machine needs to be connected over a serial line cable to the machine that you wish to debug (the *target* machine). The serial line cable on the target machine *must* be connected to communications port 1 (aka `/dev/tty00`). On the host machine, it can be connected to either of port 1 or 2. For now, let us assume it is connected to port 1 on both machines. The serial line parameters are fixed to 38400 bps, 8-bits, no parity. GDB must be of version 4.12 or newer. Older versions of GDB cannot handle remote debugging of 64 bit architectures.

To debug a kernel, proceed as follows:

[1] For security reasons, many `bootpd` and `tftpd` programs place restrictions on the location of the boot directory. Be sure to check the appropriate system documentation first.

1. Boot the file `xkernel.boot` on the target machine, specifying bootflag `k`. The console on the target machine should display the message:

   ```
   kgdb: waiting for remote gdb to connect...
   ```

2. On the host machine, `cd` to the build directory in which you built the kernel (e.g., build/example).

3. On the host machine, startup GDB as follows:

   `$` **gdb xkernel**
   `(gdb)` **set remotebaud 38400**
   `(gdb)` **target remote /dev/tty00**

   NOTE: GDB is invoked with `xkernel`, *not* the bootable file `xkernel.boot`.

After entering the "`target remote`" command, GDB should return with the prompt. It is now possible to use the normal GDB commands to set breakpoints, inspect source code, and so on. The kernel can be resumed with the `continue` command.

Interrupting with GDB

If GDB is attached to a running kernel, it is possible to interrupt it by typing a single Ctrl-C in GDB (i.e., on the host system). It is often necessary to find the address at which the CPU was executing prior to the interruption. This can be achieved by executing GDB's `up` command until reaching the frame in file `alpha_locore.s`. In this frame, the stack pointer (GDB variable `$sp`) points to the saved CPU state. The saved state can be printed with the command:

```
p/x *((Alpha_Saved_State*)$sp)
```

The command will print the Alpha's CPU registers (except for the floating point registers). The program counter is stored in field `pc`. So to find the source line at which the CPU was executing, you can type:

```
list *((Alpha_Saved_State*)$sp)->pc
```

Attaching GDB to a Running Kernel

It is also possible to attach to a running kernel that has not been started with bootflag `k`. To drop a running kernel into GDB mode, all you need to do is send a single Ctrl-C character over the serial line. This is most easily done by a communications program such as Kermit. For example:

`$` **kermit**
`C-Kermit>` **set line /dev/tty00**
`C-Kermit>` **set speed 38400**
`C-Kermit>` **connect**

At this point, type Ctrl-C once. After that, the kernel should respond with a string looking similar to this:

```
$T050f:0000002000000000;1e:90beff0100fcffff;...
```

Now you can exit Kermit (typically by typing "Ctrl-\ c quit"). It is then possible to start GDB and attach it to the target system in the usual manner (as described above).

13.7 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.

13.8 Troubleshooting

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

- The *x*-kernel aborts before a 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.

- *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 13.3.) 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 can not 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 can't find a protocol number for your protocol in any of its tables, it will abort before calling the protocol's initialization routine. You will need to add an entry for your protocol in one of the tables (see section 13.1.)

- 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, put the source files in /usr/xkernel/PLATFORM/drivers instead of /usr/xkernel/protocols.

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

15 System Installation

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

The *x*-kernel is commonly used for both research and instruction. A typical scenario is for a researcher (instructor) to install the *x*-kernel for use by various network projects (students). Individual projects (students) then build and test kernels in private working directories. This section describes the work the researcher (instructor) needs to do to set up the *x*-kernel directory.

15.1 Unpacking the *tar* file

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

`tar xf` *file*

where *file* is the disk or tape file of the *x*-kernel distribution. This will create several subdirectories in `/usr/xkernel`. A description of the *x*-kernel directory hierarchy can be found in the README file at the top level of the *x*-kernel hierarchy.

15.2 About Compilers

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

We recommend building the sunos version of the *x*-kernel with a gcc earlier than 2.0, if possible.

15.3 Site-specific Configuration

As the first person installing the *x*-kernel at your site, you should look over the makefiles in /usr/xkernel/build/Template. They have many site-specific parameters referring to compilers, libraries and the like which should be set to reflect the environment at your site.

The *x*-kernel is distributed with two versions of the protocol number table in /usr/xkernel/etc, prottbl.std and prottbl.nonstd. Since several example graph.comp and rom files refer to a file named prottbl in that directory, you should probably copy or link the version most appropriate for your site to this name. See section 13.1 for a description of the difference between the two.

15.4 Building the Utility programs

The *x*-kernel requires a few utility programs. We distribute binary versions of these programs, so you probably don't need to rebuild these programs and can skip to the next section.

If you decide to rebuild the utility programs, you must first have gnumake built and installed. A copy of the source code and build instructions for gnumake version 3.66 can be found in util/make. After you have built gnumake, place the binary in the appropriate /usr/xkernel/bin/BINTYPE directory, where BINTYPE is one of: `sunos-sparc`, `mach3-mips`, `mach3-intelx86`, `irix-mips`, or `scout-alpha`.

Next, put `/usr/xkernel/bin/BINTYPE` and `/usr/xkernel/bin` in your search path. They should appear before `/bin` and `/usr/bin` in order to pick up gnumake before the standard Unix `make`.

To build the rest of the utility programs, make a new subdirectory of /usr/xkernel/build, and copy the appropriate makefile in /usr/xkernel/build/Template to Makefile in your new directory. Change to this directory, and run `make utilDepend` and `make utilInstall`.

15.5 Building and testing the system

To install the system, build a kernel as described in section 12, paying attention to the notes for system installers. You will want to configure a kernel that includes one or more of the test protocols (e.g., `udptest`). You should then be able to run your test kernel as described in section 13.

16 Installing MachNetIPC - A Network Extension of Mach3 IPC

By compiling a protocol graph with the machnetipc protocol and its supporting protocols, one can construct a server that runs in the Mach3 environment and turns the native IPC mechanism into one that extends across a set of machines. This can be used as replacement for the Mach program known as the "netmsgserver." MachNetIPC is faster than the netmsgserver and supports all Mach3 IPC semantics and the netname interface. In addition, the protocols can work over internetworked configurations. This release is a beta test release. We expect and welcome problem reports (see section 16.7). Our testing has established that the transmission of simple messages appears correct and robust; complex messages involving port transfers are correct and robust in so far as our environment has allowed testing. Our test platforms have been HP 720's running OSF/1 from Utah, DecStations, running MK74 and UX35 and higher, and Intel 486-based machines running MK82 and UX41.

The files in the mach3/api/netipc directory comprise the interface between the user tasks and the protocols that transport data. When they are composed with a protocol graph providing the transport and liveness services they require, the resulting xkernel image file can run as a task that supports the netnameserver interface for registering and retrieving Mach ports on other machines running the same software. NB this software will not interoperate with the netmsgserver distributed by CMU.

The implementation of MachNetIPC is described in the Usenix Proceedings of the Mach III Symposium, Santa Fe NM, April 1993.

16.1 Building MachNetIPC

To build a MachNetIPC service, begin by creating a "build" directory as described in 12.2. Copy the file mach3/api/netipc/graph.comp to that directory and configure the graph.comp file to refer to the name of the ethernet device for your system.

You can then type `make compose; make depend; make` to build an xkernel image file.

16.2 Running MachNetIPC

Application programs registering and using cross-machine services make use of the netname interface to MachNetIPC. The "nns" interface to MachNetIPC can operate either as the sole name server on a node, or in cooperation with a service that is already running (such as "snames" or "netmsgserver"), although this latter configuration slightly complicates the initial contact with the service. See the appendix section about "nns" for more information on how to use this service. The host administrator should decide whether to run MachNetIPC alone or in cooperation with other services.

The MachNetIPC service can be added to the boot time server startup script nanny.config in place of the "netmsgserver". Consult your local system administration guide for more information about server startup procedures. Note that the *x*-kernel must be started with a user id that has permission to access the master device port. If your site does not have a RARP server (an unusual circumstance), you may find it necessary to create a ROM file and start the xkernel image file from the directory with the rom file. See section 13.2 and the ARP man page in appendix A for details.

If the *x*-kernel terminates or is terminated, it can be restarted easily ("xkernel &").

16.3 Using MachNetIPC

You can use the xkernel image built with MachNetIPC in place of the older netmsgserver. The most common way of utilizing it takes advantage of the netname interface, which is supported by the netmsgserver and by the "snames" program. This supports registering a name string and a port for access by remote machines, and lookup of the port corresponding to the name. The ports can be used just like any other Mach port port (with respect to send and receive operations).

Documentation for the netname interface can be found in either the /usr/mach/man area on your system or in this *x*-kernel distribution in the mach3/api/netipc/netname_*.3 files.

CAVEAT: This release does not support using DNS hostnames for netname lookups. Use the full IP address of remote hosts. See also the protocol appendix section for the "nns" service.

There is an aspect of the netname interface that depends on having a broadcast address available. If one machine wants to look up a name without specifying the IP address of the destination, a broadcast query is used. If the machine offering the name is not accessible via broadcast, then the lookup will fail. See section 16.4 for information about setting the UDP port number for this service.

With the exception of broadcast name lookup, all MachNetIPC functions work transparently between machines with local area network or Internet connectivity.

16.4 Protocol Tables

The protocol table defines the UDP port number to use for broadcasts. We have set it to be the same as is used by the netmsgserver. If this causes problems at your site, you can change the entry at the nns line:

```
udp     5
{
        pmap    111
        nns     x7823
}
```

The illustration shows that the hex number "7823" is used.

If your site will not be using the broadcast mechanism for netname lookup, the protocol number is irrelevant.

16.5 Large Messages

There is no inherent limit on the total size of a message sent using Mach IPC. MachNetIPC has a default configuration that is limited to sending datagrams no larger than 64K. Programs that transfer large amounts of out-of-line data may be inconvenienced by this. We provide an alternate configuration that supports arbitrarily large messages. We also note that very large message transfers may adversely affect the performance of a host node, due to having to buffer the message in the *x*-kernel task while it is being transferred (this is worse for the receiver than the sender).

The file mach3/api/netipc/graph.comp.large can be used in place of the standard graph.comp to support large messages. It relies on a newly developed protocol (RAT) that implements a reliable datagram protocol over TCP.

If you wish to use RAT for very large messages (more than 1M byte or so), you will need to recompile several *x*-kernel files in a special manner. We recommend that you recompile these files in your local build directory in order to avoid conflicts with other users. The relevant files are:

```
xkernel/pi/msg.c
xkernel/pi/msg_internal.h
xkernel/mach3/pxk/malloc.c
xkernel/mach3/pxk/stack.c
```

The file msg.c contains the message tool implementation, which must be compiled with the MSG_NON_REC flag to work with large messages. The standard Mach malloc doesn't return and reuse large storage blocks very well, so we recommend compiling our distributed version of malloc.c for large messages. You might also wish to compile and use the stack.c that we provide; it limits cthread stacks to 64K and uses less virtual memory.

To compile special versions of these files:

- Copy (or link) these files to your build directory. Edit your Makefile in your build directory to indicate that these files will be locally compiled, by adding to the PRIVSRC and PRIVOBJ variables:

```
PRIVSRC = \
        msg.c    \
        malloc.c \
        stack.c

PRIVOBJ = \
        ./$(HOW)/msg.o    \
        ./$(HOW)/malloc.o \
        ./$(HOW)/stack.o
```

- Add -DMSG_NON_REC to the TMP_CPPFLAGS variable in your Makefile.
- Run `make depend` and `make`.

16.6 Access Control and Privacy

This release of MachNetIPC has no access control or privacy mechanisms. We have expectations of providing these services in 1994.

16.7 Reporting Problems

Problems can be reported to xkernel-bugs@cs.arizona.edu, but problems and discussion that are specific to MachNetIPC should go to xkernel-mnipc@cs.arizona.edu; if you wish to be a member of this list send a note to xkernel-mnipc-request. If you aren't sure which list to use for a problem, use xkernel-bugs.

17 Installing within a Mach3 Kernel

The *x*-kernel can be compiled as an integral part of the Mach3 kernel. This facility is provided for researchers developing highly efficient protocols and for obtaining timing results that are exclusive of Mach IPC and Unix server overhead. At the present time, there is no user interface for accessing the *x*-kernel services – the Mach3 kernel can boot, run an *x*-kernel timing test and print the results. A more usable interface will be provided in the future.

These are instructions for building a version of the *x*-kernel that resides in the Mach3 kernel. You should be familiar with the build instructions for the "out of kernel" version before starting. You should also be familiar with the procedure for building the Mach kernel and have a working environment set up.

The files distributed (and these directions) have been used with the MK82 Mach kernel for a DecStation 5000 and an Intel 486 PC.

Set your path to include the /usr/xkernel/bin and /usr/xkernel/bin/ARCH directories.

1. Place the *x*-kernel source tree in the mach source tree. In the src/mk/kernel directory, in either a shadow area or the primary source area, create a link named 'xkern' to the head of the *x*-kernel source tree.

2. Patch some mach kernel source files. The tree rooted at xkernel/mach3/machkernel is a parallel of the mach kernel source tree, containing modifications to support the *x*-kernel. These modifications are all conditional on *x*-kernel compilation flags, so you can use these modified files to build non-*x*-kernel Mach kernels if you desire. You can either go through these files one at a time and make the appropriate file-by-file substitutions, or you can set up a mach shadow directory and link/copy the directories in xkernel/mach3/machkernel to the shadow area. The latter should be the easier approach (and less intrusive to your original mach source tree.)

   Note that one can not transparently shadow the mk/kernel/conf directory. Since the directory xkernel/mach3/machkernel/conf only includes files that we have modified, it should be overlayed on top of a copy of an original mk/kernel/conf directory.

3. Compose an *x*-kernel graph. Create a directory xkern.local under src/mk/kernel. This directory will contain the files created by composing the graph.comp. Change to the new xkern.local directory. Copy or create your graph.comp here. To compose this graph.comp run:

   ```
   compose -f < graph.comp
   ```

   This must be explicitly done each time the graph.comp changes. The files traceLevels.c, protocols.c, protocols.h and protTbl.c will be created. The files that were created here are referenced by the 'conf/files' file for inclusion in the mach kernel.

4. In your xkern.local directory, build a compilable version of the protocol table by running the *x*-kernel utility program ptbldump:

   ```
   ptbldump /usr/xkernel/etc/prottbl > ptblData.c
   ```

   This file will be automatically included by an *x*-kernel source file.

5. In your xkern.local directory, create a compilable rom file by running the genrom.awk script:

```
awk -f /usr/x32/merge/bin/genrom.awk < rom > initRom.c
```

This file will be automatically included by an *x*-kernel source file.

6. Build a configuration including the *x*-kernel. If your configuration includes the string `+xk`, the *x*-kernel will be built in DEBUG configuration (with tracing enabled.) If the configuration includes the string `+xko`, the *x*-kernel will be built in OPTIMIZE configuration (with no tracing.)

7. If you have problems with kernels crashing because of storage exhaustion, you might want to adjust the zone block table in the Mach kernel file kalloc.c. See the *x*-kernel file xkernel/pi/msg_internal.h for the size of a message block; these account for most of the *x*-kernel storage use.

8. The machkernel will come up by default with both the Unix server and the *x*-kernel receiving and sending network packets (incoming packets are copied to both the *x*-kernel and the Unix server.) For accurate kernel-kernel timings of *x*-kernel protocols, boot the mach_kernel image (which doesn't start the user task) instead of the mach.boot image and configure the in-kernel driver (xklance) to not deliver incoming packets to the user task as described in Appendix A.

A Protocol Specifications

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

NAME

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

SPECIFICATION

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

SYNOPSIS

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

REALM

Indicates whether the protocol is in the ASYNC realm (supporting push, demux and pop), the RPC realm (supporting call, calldemux and callpop), 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) and whether the control operation is understood by `x_controlsessn` or `controlprotl` (or both) is specified. In the case of control operations that take multiple arguments, we give a set of types. Non-primitive types are generally defined in the protocol's `.h` file.

EXTERNAL INTERFACE

Interfaces not encapsulated within *x*-kernel operations

CONFIGURATION

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

AUTHORS

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

NAME

ARP (Address Resolution Protocol)

SPECIFICATION

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

SYNOPSIS

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

REALM

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

CONTROL OPERATIONS

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

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

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

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

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

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

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

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

`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

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

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

NAME

BIND (A Layered Encryption Key Manager)

DISTRIBUTION NOTE

This protocol will not be distributed via anonymous ftp. If interested, contact xkernel-bugs@cs.arizona.edu about obtaining this protocol.

SPECIFICATION

BIND sits on top of an instance of the KM protocol and maps a participant to a id which is then used by the KM to provide a key.

SYNOPSIS

As we have discovered in decomposing Kerberos, any binding between network addresses and keys can be very transient. The BIND protocol was designed to be inserted between CRYPT and KM to manage temporary bindings. Thus the KM data base will contain name to key bindings while the BIND data base will contain address to name bindings. CRYPT will open a BIND session just as it would a KM session. When the address to name mapping is set (through calling a KM_SET or BIND_SET control op) BIND will open a KM session using the name as its participant. When CRYPT performs a KM_RESLOVE on the BIND session it in turns perform a KM_RESLOVE on the corresponding KM session.
In this organization the KM data base is fairly stable and managed via a separate key management protocol suite. The address name bindings are set by performing a CRYPT_IN(or OUT)_SET on the appropriate session with the name which corresponds to this session. Note names in this version are fixed length byte strings.
Note that a KM_RESLOVE on BIND gets you a key from the KM data base while a KM_SET takes a name not DES key. This is not symmetric and exists only to support Kerberos.
Names are simply fixed length byte strings.
The keys file name is formed by concatenating the strings "bindings" with the instance name of the protocol (which may be null).

REALM

BIND is in the ASYNC realm.

PARTICIPANTS

BIND accepts participants in order to do a table lookup to find the key. The entire participant list is converted into a byte string which is then used to lookup the key.

CONTROL OPERATIONS

BIND_RESOLVE returns a copy of the name stored in that session. This name is the name which is associated with the participant list passed at open time. BIND can fail if no key has been set or that key has been invalidated.

BIND_SET sets the name for that session to the value passed.

BIND_INVALIDATE marks the name as invalid and removes the BIND session from the map. The session is not destroyed until all users of that session have closed it.

BIND_ISEMPTY succeeds if no name for this session has been defined.

BIND_ISVALID succeeds if there is a valid name for this session.

BIND_ISINVALID succeeds if a BIND_INVALIDATE operation has been performed in this session.

BIND also supports all of the KM control operations. The only two which should ever be used are:

KM_RESOLVE performs a KM_RESLOVE on the corresponding KM session. (ie it returns a DES key not a name).

KM_SET is identical to BIND_SET.

CONFIGURATION

BIND must be configured on top of KM and any protocol which uses KM can use BIND.

KEY FILES

BIND key files are identical to KM key files.

AUTHOR

Sean W. O'Malley

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 GETMAX-PACKET 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

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

NAME

CONFOUNDER

DISTRIBUTION NOTE

This protocol will not be distributed via anonymous ftp. If interested, contact xkernel-bugs@cs.arizona.edu about obtaining this protocol.

SPECIFICATION

The CONFOUNDER protocol prepends a 4 byte pseudo-random number to the front of each outgoing packet. It pops (and discards) 4 bytes from the front of each incoming packet.

REALM

CONFOUNDER is in the ASYNC realm.

PARTICIPANTS

CONFOUNDER passes participants to the lower protocols without manipulating them.

CONTROL OPERATIONS

CONFOUNDER recognizes the following control operations; all others are passed unchanged to the lower protocol or session.

GETMAXPACKET and GETOPTPACKET: The packet size returned by the lower protocol/session is diminished by 4, and passed upward.

IP_PSEUDOHDR: This control operation turns on the IP pseudoheader length-fixup flag, either for a session or the entire protocol. The control operation is also passed to the lower session or protocol. See IP (page 156) for an explanation of this kludge.

CONFIGURATION

CONFOUNDER expects to be configured on top of a transport protocol. The transport protocol must preserve packet boundaries (i.e. CONFOUNDER will not work on top of TCP).

Example of a graph.comp file:

```
---------------------------------
@;
name=simeth/0;
name=eth protocols=simeth/0;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet;
name=confounder protocols=ip;
name=udp protocols=confounder;
```

```
@;
prottbl = ../../../etc/prottbl.nonstd;
---------------------------------
```

RESTRICTIONS

CONFOUNDER always generates the same sequence of pseudo-random numbers.

AUTHORS

Hilarie Orman and Richard Schroeppel

NAME

The Phil Karn Magic Cookie Protocol

SPECIFICATION

The cookie protocol maintains a table of MD5 generated identifier strings based on remote IP addresses. Hosts rely on the cookies as a weak form of authentication that can be used as a filter to protect computationally expensive protocols, such as Diffie-Hellman key exchange.

Phil Karn presented the ideas for this scheme as part of his Photuris key management system describe in the IETF draft RFC series.

SYNOPSIS

There are two kinds of cookie messages: requests and responses. If no cookie is known when the session is started, a request is sent; the remote host replies with a cookie. The requesting host then attaches the cookie to the current message. The remote host removes the cookie, and if it matches, delivers the remainder of the message.

The local and remote cookies are attached to all outgoing messages using the response message type.

REALM

COOKIE is in the ASYNC realm.

PARTICIPANTS

COOKIE passes participants to the lower protocols without manipulating them.

CONTROL OPERATIONS

There are no control operations. All control operation requests are passed through unchanged.

CONFIGURATION

Example of a graph.comp file:

```
---------------------------------
@;
name=simeth/0;
name=eth protocols=simeth/0;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet;
name=cookie protocols=ip;
name=udp protocols=cookie;
name=udptest protocols=udp;
@;
```

```
prottbl = ../../../etc/prottbl.nonstd;
---------------------------------
```

DEPENDENCIES

COOKIE relies on the MD5 algorithm from the *x*-kernel library.

AUTHOR

Hilarie Orman

NAME

COOKIE_PHOTURIS - The Photuris Magic Cookie Protocol

SPECIFICATION

The COOKIE_PHOTURIS protocol implements the cookie part of the PHOTURIS protocol as described in Photuris draft 8.

On every active open a new cookie is generated which is attached to every outgoing message of this session. If a session is created by a passiv open the local cookie is generated using the address of the opening host and the random value stored in the protocol state. Every incomming message has to have a valid cookie except for messages of type 0 which are passed to the higher level protocol in any case.

Demultiplexing is done based on lls, Initiator_Cookie and Responder_Cookie.

Every message has to start with:

16 octets Initiator Cookie followed by 16 octets Responder Cookie followed by a one octed message type.

The cookies are removed before the message is passed to the higher level protocol.

REALM

COOKIE_PHOTURIS is in the ASYNC realm.

PARTICIPANTS

COOKIE_PHOTURIS passes participants to the lower protocols without manipulating them.

CONTROL OPERATIONS

`COOKIE_PHOTURIS_INITIATOR_COOKIE:` Retrieves the cookie of the initiator of the session. (session only)

Input: none

Output: Cookie of initiator.

Return Value: XK_SUCCESS if sucessfull.

`COOKIE_PHOTURIS_RESPONDER_COOKIE:` Retrieves the cookie of the responder of the session. (session only)

Input: none

Output: Cookie of responder.

Return Value: XK_SUCCESS if sucessfull.

`COOKIE_PHOTURIS_SET_RANDOM_VALUE:` Sets the random value used to generate cookies for a passive open. (protocol only)

Input: Random value of size integer.

Output: none

Return Value: XK_SUCCESS if sucessfull.

All other controll operations are passed to the lower level protocol.

CONFIGURATION

COOKIE_PHOTURIS should be configured above UDP.

For an example of a graph.comp file see the description of PHOTURIS.

DEPENDENCIES

COOKIE_PHOTURIS relies on the MD5 algorithm from the *x*-kernel library.

Shortcommings

Random numbers are generated using rand().

COOKIE_PHOTURIS doesn't keep a passive map. Therefore only one higher level protocol can be configured above COOKIE_PHOTURIS.

AUTHOR

Oliver Spatscheck

NAME

CRC-32 (32-bit Cyclic Redundancy Check)

SPECIFICATION

For outgoing messages, the CRC-32 protocol calculates a 32 bit checksum and pushes it onto the front of the message. For incoming messages, a 32 bit checksum is popped from the front of the message, and compared against the computed checksum of the rest of the message. No special action is taken when the compare fails; the shortened message is passed on as usual to the next higher protocol. If tracing is enabled, a message is printed when the compare fails. There is an option to use an implicit prefix at the front of each packet when computing the checksum; separate prefixes may be used for incoming and outgoing packets.

The checksum algorithm is the IEEE's CRC-32 algorithm. By itself, it is not cryptographically secure, although it is sometimes used as a component of cryptographic systems.

REALM

CRC-32 is in the ASYNC realm.

PARTICIPANTS

CRC-32 passes participants to the lower protocols without manipulating them.

CONTROL OPERATIONS

CRC-32 recognizes the following control operations; all others are passed unchanged to the lower protocol or session.

GETMAXPACKET and GETOPTPACKET: The packet size returned by the lower protocol/session is diminished by 16 bytes and passed on.

IP_PSEUDOHDR: This control operation turns on the IP pseudoheader length-fixup flag, either for a session or the entire protocol. The control operation is also passed to the lower session or protocol. See IP (page 156) for an explanation of this kludge.

The PREFIX control operations apply only to sessions, not the protocol.

HASH_SETLOCALPREFIX sets the implicit prefix used in computing the checksum for each outgoing packet. The prefix may be reset as desired, or turned off by setting a prefix of length 0.

HASH_SETREMOTEPREFIX sets the implicit prefix for incoming packets.

HASH_SETPREFIX sets the implicit prefix for both directions.

CONFIGURATION

CRC-32 expects to be configured on top of a transport protocol that preserves packet boundaries (i.e. CRC-32 will not work on top of TCP).

Example of a graph.comp file:

```
--------------------------------
@;
name=simeth/0;
```

```
name=eth protocols=simeth/0;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet;
name=crc protocols=ip;
name=udp protocols=crc;
name=udptest protocols=udp;
@;
prottbl = ../../../etc/prottbl.nonstd;
---------------------------------
```

AUTHOR

David Schwartz

NAME

CRYPT CRYPT3 and TCRYPT (DES Encryption Protocols)

DISTRIBUTION NOTE

This protocol will not be distributed via anonymous ftp. If interested, contact xkernel-bugs@cs.arizona.edu about obtaining this protocol.

SPECIFICATION

CRYPT encrypts or decrypts all bytes of all packets sent using the DES encryption algorithm. CRYPT uses the key manager protocol (KM, page 165) to map from the source address passed in at open time to a DES key. Each message is encrypted independently, unless the IV_STATE flag is used. CRYPT is designed to be composed over any datagram protocol, and it accepts arbitrary address types at open time. Except when the FRAGMENT flags are selected, CRYPT lengthens each packet by 1 to 8 bytes, to the next larger multiple of 8 bytes.

CRYPT3 is the same as CRYPT, except the triple DES algorithm in cipher-block-chaining mode is used. In this mode, the block chaining is done from the third stage back to the first; in other respects each stage operates in ECB mode. CRYPT3 encrypts all messages sent using a 16 byte key (instead of a 8 byte key) from which the first 8 bytes represent the first key and the second 8 bytes the second key. The 3 stages of triple DES are: encrypt with the first key, decrypt with second key, encrypt with the first key.

CRYPT3IN is the same as CRYPT3, except that there is no feedback between stages, and each stage operates in CBC mode. CRYPT3IN requires 24 bytes of key.

TCRYPT is also the same as CRYPT, except that packet lengths are usually unchanged. However, TCRYPT alters packets in the size range 1 to 7 bytes. All of the information below for CRYPT also applies to TCRYPT, except for packet sizes. Several of the CRYPT options change the packet size; these won't work properly with TCRYPT.

SYNOPSIS

When a CRYPT session is opened, it opens the protocol configured below it with the addresses passed to it during open. It then performs a GETPARTICPANTS operation on the newly opened session to acquire the fully resolved addresses of the participants. The set of participants is then used as the argument to open a key manager (KM) session.

When a message is pushed to a CRYPT or CRYPT3 session a KM_RESOLVE operation is done on the opened key manger session to get the current local key. Because DES and triple DES operates on blocks of 8 bytes, the message is zero-padded to the next larger multiple of 8 bytes; the last byte is set to the number of pad bytes added. Total padding is at least 1 and at most 8 bytes. The receiver uses KM_RRESOLVE to ask his key manager for the remote key. He decrypts or encrypts the message and removes the padding. If the pad bytes are not 0, or the last byte has an out-of-range value for the amount of padding, the message is dropped. Assuming the incoming length is a multiple of 8 bytes, the probability that a message decrypted with a bad key will pass the length and pad checks is about 1/255. The FRAGMENT flags may be used to select an alternative to padding. The usual operation mode of encipherment is called cipher-block-chaining (CBC), explained with the CBC flag below. There are several options for setting and using the initial vector (IV) for cipher-block-chaining, mostly controlled by flags in the session and protocol state.

The default padding method for crypt is compatible with the DES CBC transform for IPSEC (RFC1825) and the Photuris key exchange protocol (an internet draft).

TCRYPT preserves packet lengths, except those in range of 1-7 bytes. For this reason, TCRYPT is not expected to be used below protocols that produce very short messages. The first part of the packet encryption algorithm is the same as for CRYPT: as much of the packet as possible is encrypted in blocks of eight bytes. If there are leftover bytes at the end of the packet, enough (already encrypted) adjacent bytes from the packet are used to make up a complete block of eight bytes, which is encrypted. Decryption is straightforward. Because the map is length preserving, there is no possibility of decryption failing (although it might produce garbage). CRYPT and TCRYPT have each been tested on both big-endian and little-endian machines, and between machines of different endianness.

CRYPT discards the attributes attached to any outgoing message. This is done to minimize out-of-band information transmittal. CRYPT does not touch any attributes attached to incoming messages, and forwards them along with the decrypted data up to the next protocol.

On outgoing messages, if the key manager should return a key of all 0s, indicating it does not have the real key, the message is dropped. For incoming messages, a key of all 0s is allowed.

REALM

CRYPT is in the ASYNC realm.

PARTICIPANTS

CRYPT passes participants to the lower protocols without manipulating them. It uses the participants to lookup the keys for encryption and decryption.

CONTROL OPERATIONS

CRYPT supports 31 control operations. 8 of these operations are passed to the incoming key manager and 8 are passed to the outgoing key manager. Every KM control operation has two corresponding CRYPT control operations. CRYPT_IN_SET sets the key on the incoming key manager by performing a KM_SET operation on it. Likewise CRYPT_OUT_SET peforms a KM_SET on the outgoing key manager.

A CRYPT session maintains several state variables, which may be read and written with control operations. The variables are a (long) flags word; the initial value to use for cipher-block-chaining, 8 bytes for each of the up and down directions; a 16 byte state variable for the IV random number generator; and (up to) 8 bytes for remembering a fragment of a packet in each of the up and down directions.

The CRYPT control operations are listed below:

```
CRYPT_IN_RESOLVE          CRYPT_IVEC_FROM_KEY
CRYPT_IN_SET              CRYPT_ISPRIVATE
CRYPT_IN_INVALIDATE       CRYPT_READ_FLAGS
CRYPT_IN_ISEMPTY          CRYPT_SET_FLAGS
CRYPT_IN_ISVALID          CRYPT_CLEAR_FLAGS
CRYPT_IN_ISINVALID        CRYPT_READ_IV_DOWN
CRYPT_IN_KEYSIZE          CRYPT_SET_IV_DOWN
CRYPT_IN_HOSTSIZE         CRYPT_READ_IV_UP
CRYPT_OUT_RESOLVE         CRYPT_SET_IV_UP
```

```
CRYPT_OUT_SET                  CRYPT_READ_IV_RANDOM
CRYPT_OUT_INVALIDATE           CRYPT_SET_IV_RANDOM
CRYPT_OUT_ISEMPTY              CRYPT_READ_FRAG_DOWN
CRYPT_OUT_ISVALID              CRYPT_SET_FRAG_DOWN
CRYPT_OUT_ISINVALID            CRYPT_READ_FRAG_UP
CRYPT_OUT_KEYSIZE              CRYPT_SET_FRAG_UP
CRYPT_OUT_HOSTSIZE
```

The CRYPT_IVEC_FROM_KEY control operation sets the CRYPT_IV_FROM_KEY flag; see below for the effect of this flag. The CRYPT_ISPRIVATE control operation returns a value of 1 if a CRYPT session sees it. This is a way to see if a session has a CRYPT protocol somewhere beneath it.

CRYPT diminishes the results of GETOPTPACKET and GETMAXPACKET to the next lower size congruent to 7 (mod 8); this will reduce the size by at least 1 and at most 8 bytes. This is done independently of how the flags are set. (This is technically a bug, since corrections should be made for the IV length.)

TCRYPT does not change the packet sizes, except that values less than 8 are set to 0.

CRYPT forwards all other control operations to the transport protocol configured below it. Due to security concerns CRYPT zero's out the buffer of any control it passes down the graph.

Two non-CRYPT control operations get special handling. ETH_REGISTER_ARP is an exception to the rule about zeroing out the buffer on the way down: It must be allowed to pass its data through the CRYPT *protocol* (but not through CRYPT sessions). The IP_PSEUDOHDR control operation turns on the CRYPT_IP_PSEUDOHDR_HACK flag, and is passed down to any lower protocols or sessions to notify them that an IP pseudo-header will be attached to upgoing messages, and they should do a length fixup if appropriate. Use this control operation rather than setting the flag in CRYPT, since it works on the entire chain of protocols or sessions down to IP. There's no provision for turning the flag off - you could clear it in CRYPT with CRYPT_CLEAR_FLAGS, but the clearing wouldn't be propagated to lower protocols.

FLAGS

Much of the detailed behavior of CRYPT is controlled from the flags. The same flags exist for both sessions and protocols. The session and protocol flags can be read, set, and cleared with control operations. The flag setting operation does an Or into the flags, and the clearing operation does And-with-Complement. This leads to reasonably natural code.

The default flags setting has CBC on, and everything else off. The ROM options change the setting of several of the flags.

Usually the flags are setup at initialization time, either for the protocol or the session. The user is free to manipulate the flags at any time, but it seldom makes sense to do so. (Timing races are possible if the code diddles with a flag that affects processing of incoming messages.) Most of the IV-related manipulations are in effect if either the protocol or the session flag is set. This allows coarse control by setting the flag in the protocol state, or fine control by leaving the protocol flags in the default state and selectively setting the session flags on a per-session basis. The flags in the left column are effective if set for either the protocol or the session; the flags in the right column are protocol-only. The flags are each a single bit. The symbolic names are:

```
CRYPT_IP_PSEUDOHDR_HACK        CRYPT_ALL_PARTS
CRYPT_IV_FROM_KEY              CRYPT_CBC
CRYPT_SEND_IV                  CRYPT_DECRYPT
```

```
CRYPT_SEND_IV_ONCE              CRYPT_USE_IPSEC
CRYPT_RCV_IV_ONCE               CRYPT_NO_PADDING_CHECK
CRYPT_CRYPT_IV                  CRYPT_RFC_1829
CRYPT_IV_STATE
CRYPT_RANDOM_IV
CRYPT_IV_FROM_MSG
CRYPT_IV_ON_UPMSG
CRYPT_ROTATE_BYTE
CRYPT_FRAGMENTS_DOWN
CRYPT_FRAGMENTS_UP
```

Typical code to set flags for a session looks like

```
long f = CRYPT_SEND_IV | CRYPT_RANDOM_IV;
xControl(session,CRYPT_SET_FLAGS,(char *)&f,sizeof(long));
```

What the flags mean: CRYPT_ALL_PARTS
This flag is set if the ROM option *numparts* exceeds 1. See the explanantion of the Rom option.

CRYPT_CBC
This flag tells CRYPT to use Cipher Block Chaining (CBC) encryption mode. In CBC mode, each 8-byte block of ciphertext is Xored into the next plaintext block before encryption. (This causes successive identical 8-byte blocks, such as a long stream of zeros, to encrypt to different ciphertext.) This is the only flag whose default value is ON. In CBC mode, a dummy zeroth block of ciphertext is required to get started. This block is called the Initial Value (IV); it is Xored into the first plaintext block before encrypting. CRYPT has several different ways of specifying the IV, controlled from the flags. If no other IV-related flag is set, the IV is taken from the session state, either the IV_UP or the IV_DOWN field. These are initialized to 0, but may be set with a control operation. If the CBC flag is OFF, encryption is done in Electronic Code Book (ECB) mode. In this mode, each plaintext block of 8 bytes is encrypted independently. No IV is needed. This flag is cleared by the ROM options *EECB* and *DECB*.

CRYPT_DECRYPT
This flag swaps the normal encryption and decryption operations. Usually, messages are encrypted on the way down, and decrypted on the way up. This flag causes downward messages to be decrypted, and upward messages to be encrypted. Only the inner encryption box is affected, as if a different cipher were substituted. Padding is unaffected; CBC operates in the usual way: On an outbound (downgoing) message, each decrypted block is xored into the next plaintext block, and the result becomes the next block to be decrypted. This flag is set by the ROM options *DCBC* and *DECB*.

CRYPT_USE_IPSEC
This flag is set by the ROM option *useipsec*. See the explanation of the ROM option.

CRYPT_NO_PADDING_CHECK
The contents of the padding of incomming messages is not checked against zero.

CRYPT_RFC_1829
This flag is set by the ROM option *rfc1829*. See the explanation of the ROM option.

CRYPT_IP_PSEUDOHDR_HACK
This flag tells CRYPT to look for an IP pseudo-header on upgoing messages (as a message attribute), and adjust its length field to reflect any message length changes caused by CRYPT. This flag should be set with the IP_PSEUDOHDR control operation.

For all IV operations, note that their use with CRYPT3 and CRYPT3IN is may be unsupported or fragile, respectively. CRYPT3IN will use the 8 byte IV as input to the first stage; the second and third stages use a zero IV.

CRYPT_IV_FROM_KEY

This flag tells CRYPT to use the encryption key as the IV. Presumably the remote host also knows the key, and will use it for decryption. As an alternative to the usual flag setting code, this flag can be set with the control operation CRYPT_IVEC_FROM_KEY, used either on a CRYPT protocol or session. CRYPT notices the flag combination IV_FROM_KEY and either of the SEND_IV flags; unless the CRYPT_IV flag is also set (which encrypts the outgoing IV), cryptPush will refuse to send messages. (This is to avoid sending out the encryption key in a message.)

CRYPT_SEND_IV

This puts the IV at the front of outgoing messages, and expects the IV at the front of incoming messages. The IV is usually sent "in the clear" (unencrypted). This flag is required when CRYPT is used with IPSEC, the RFC1825 protocol and its DES CBC transform. Don't use this flag in TCRYPT between IP and UDP/TCP, since TCRYPT won't fixup the IP pseudo-header length for incoming messages. The *Photuris* ROM option sets this flag in the protocol.

CRYPT_SEND_IV_ONCE

The IV is sent once only, at the front of the first message that is pushed to the session. This is appropriate for an encrypted reliable stream, where packets might be repackaged, losing the position of the IV at the front of the packets. At session-creation, the session value of the flag is initialized from the value in the CRYPT protocol. Once an IV has been sent at the front of a message, the session copy of the flag is turned OFF. Perhaps a user might turn it back ON as part of resynchronizing a connection.

CRYPT_RCV_IV_ONCE

This is the receiving flag that corresponds to CRYPT_SEND_IV_ONCE. The session value of the flag is initialized from the protocol value. After a message has been received and the IV removed from the front, the session flag is turned OFF. A user might turn it back ON as part of resynchronizing, but there's a timing race to be aware of. Don't use this flag in TCRYPT between IP and UDP/TCP, since TCRYPT won't fixup the IP pseudo-header length for incoming messages.

CRYPT_CRYPT_IV

Some proposed protocols require that the IV be encrypted. This flag causes the first eight bytes of outgoing messages to be encrypted (just before being handed to the next lower protocol), and the first eight bytes of incoming messages to be decrypted (before any other processing). If the other flags are set appropriately, these bytes will contain the IV. This encryption is unaffected by the CBC or DECRYPT flags, or by the triple-encryption in CRYPT3. It will interact badly with SEND/RECEIVE_IV_ONCE.

CRYPT_IV_STATE

This flag causes the final IV (after a block is encrypted or decrypted) to be stored back into the IV_UP or IV_DOWN state. This is appropriate for stream encryption, or for chaining between packets. It will become badly confused if the messages can be reordered on the net. It is an exception to the rule that packets are encrypted independently.

CRYPT_RANDOM_IV

This supplies a pseudo-random IV for each outgoing packet. The generator has a sixteen byte internal state, presently compiled in. The encryption key is used to scramble the bits before each message is sent; the scrambled bits are Xored into the IV_DOWN. This costs 4 DES encryptions per packet, so it will slow things down a bit. This flag has no effect on the upgoing messages - the only way the receiver can learn the IV is to get it from the incoming message.

CRYPT_IV_FROM_MSG
This option is for IPSEC, the RFC1825 protocol, to support its "transform" for DES CBC. Each downgoing message passed to CRYPT is assumed to have an eight byte prefix which is popped off and used as the IV. The IV can be reattached to the message with the SEND_IV flag. There is no effect on upgoing messages. The *Photuris* ROM option sets this flag in the protocol.

CRYPT_IV_ON_UPMSG
This flag causes the IV to be prefixed onto *upgoing* messages. It is needed for Photuris. There is no effect on downgoing messages. The *Photuris* ROM option sets this flag in the protocol.

CRYPT_ROTATE_BYTE
Another required IPSEC (RFC1825) option, the payload type byte. After any prefixed IV is removed, and any message padding has been appended, downgoing messages are rotated by one byte: the first byte is removed, and placed at the end of the message. Then the message is encrypted. For upgoing messages, the reverse byte rotation takes place.

CRYPT_FRAGMENTS_DOWN
CRYPT_FRAGMENTS_UP
These flags specify data repacking options that allow DES to be used with a reliable stream transport such as TCP.

If FRAGMENTS_DOWN is ON, downgoing packets are repackaged into blocks that are a multiple of eight bytes. Any leftover fragment is retained as a prefix for the next message. For example, suppose the application was sending messages of 13 bytes. The first message would cause 8 bytes to be encrypted and shipped out; a 5 byte fragment would be retained. When the application sent a second 13 byte message, the five byte fragment would be prefixed, producing an 18 byte message. 16 bytes would be encrypted and shipped, while 2 bytes would be retained. The outgoing pattern would be a mix of 8 byte and 16 byte messages, averaging 13 bytes per message.

If the messages are not regrouped on the wire, then any of the IV options can be used, and be decrypted at the receiving end. If the wire can refragment, and the receiving CRYPT will see a different groupoing than was transmitted, then only ECB modes, or the stream IV modes, will work. Stream transmission mode is achieved by setting FRAGMENTS_DOWN and IV_STATE, and might also use SEND_IV_ONCE.

Note that the receiver cannot recover the orginal byte packing; CRYPT doesn't send any information about the fragmentation. There is no padding added to the messages in this mode (CRYPT_FRAGMENTS_DOWN).

The FRAGMENTS_UP flag is similar, but repacks the messages arriving from the wire into chunks that are a multiple of eight bytes. If FRAGMENTS_DOWN was used to send a message, FRAGMENTS_UP should be used on the receiving side, so that the receiver doesn't expect to receive padded messages.

Both modes have the property that, if they receive very short messages (less than 8 bytes), they may completely sequester the data, and not pass on a message until another arrives to complete the eight byte block. In the extreme case where an application is sending 1 byte messages, only 1/8 of its messages will cause actual network traffic, as its data is grouped into 8 byte blocks.

When the session is closed, no effort is made to transmit any final fragment - the application must either send 7 bytes of padding to guarantee the last real data has been flushed, or it must inspect the fragment and compute and send the correct padding.

Control operations have been provided to read and set the down and up fragments, in case this is needed to resynch a connection. An 8 byte buffer should be provided as an argument to the control operation; the value of the control operation is the actual number of bytes in the fragment.

CONFIGURATION

CRYPT/CRYPT3/TCRYPT can be used in two configurations: transport and "leaf mode". In transport mode, there are two lower protocols, a transport protocol and a key manager. In leaf mode, there is only a key manager. The number of lower protocols implicitly determines the mode.

Transport Mode . There are two kinds of transport protocol that are appropriate. In the most common use, the transport protocol must be a datagram protocol, one that preserves packet boundaries (e.g., TCP does not do this). Because it changes packet lengths, CRYPT and CRYPT3 will only work between IP and (UDP or TCP) if the IP_PSEUDOHDR control operation is used by the application program to instruct CRYPT or CRYPT3 to "correct" the IP pseudohdr message attribute. TCRYPT can be used without the control operation.

NB: Any protocol configured above TCRYPT should not generate packets of length between 1 and 7 inclusive, because the length will not be preserved.

A second mode of operation allows CRYPT/CRYPT3/TCRYPT to use a reliable stream transport protocol, such as TCP. The CRYPT_FRAGMENTS_DOWN and CRYPT_FRAGMENTS_UP flags together impose state retention on the encryption session management, so that any part of the message can be encrypted or decrypted. See the discussion under control operations.

Leaf Mode . In leaf mode, the caller utilizes DES much like a local library operation, i.e., no network activity ensues. The caller selects encryption or decryption by selecting xPush or xPop; upon return, the message is transformed by the algorithm. The open and close operations work normally, and the key manager is used as usual. The caller should take care to set the "up" pointer of the session to ERR_XOBJ in order to inhibit this protocol from attempting a demux operation to a non-existent upper level protocol.

Example of a graph.comp file for transport mode:

```
---------------------------------
@;
name=simeth/0;
name=eth protocols=simeth/0;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet;
name=km trace=TR_ERRORS;
name=crypt protocols=ip,km trace=TR_ERRORS;
name=udp protocols=crypt;
name=udpcrypttest protocols=udp;
@;
prottbl = DEFAULT;
---------------------------------
```

ROM file

The ROM file is used to configure the mode and direction of DES. The following options are available:

ECBC encrypts messages received from higher level protocols and decrypts messages from lower level protocols using the CBC (cipher-block-chaining) mode. This is the default direction and mode of the CRYPT and TCRYPT protocol (CRYPT and TCRYPT only).

DCBC decrypts messages received from higher level protocols and encrypts messages from lower level protocols using the CBC mode (CRYPT and TCRYPT only). Sets the DECRYPT flag.

EECB encrypts messages received from higher level protocols and decrypts messages from lower level protocols using the ECB (electronic-code-book) mode (TCRYPT only). Clears the CBC flag.

DECB decrypts messages received from higher level protocols and encrypts messages from lower level protocols using the ECB mode (TCRYPT only). Sets the DECRYPT flag, and clears the CBC flag.

The ROM option *useipsec* retrieves the secret from a key manager using the local participant (the 2nd participant) and the SPI (the 3rd participant). Crypt will use two key manager sessions, one for incoming packets and one for outgoing, and these directions may have distinct keys. Sets the USE_IPSEC flag.

The *numparts* option sets the number of participants to be used for looking up the key. Its default value is 1, and this results in using two key manager sessions, one for each direction of communication, with the possibility of a different key for each direction. The lookup key is the remote participant for outgoing messages, and the local participant for incoming messages. This scheme is insufficient for many purposes, and thus, setting the numparts value to anything greater than 1 results in the concatenation of all participants being used for the key lookup. In this case, pairwise keys between hosts can be uniquely defined. Sets the ALL_PARTS flag.

The *Photuris* option selects the IV processing needed for Photuris. It sets the protocol flags SEND_IV, CRYPT_IV_FROM_MSG, CRYPT_IV_ON_UPMSG, NO_PADDING_CHECK and RFC_1829.

The *rfc1829* option changes the behavioure of crypt as described in RFC 1829.

The following graph.comp file may be used to implement a variant of triple DES using crypt and tcrypt:

```
---------------------------------
@;
name=simeth/0;
name=eth protocols=simeth/0;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet;
name=km/one;
name=km/two;
name=tcrypt/one  protocols=ip,km/one;
name=tcrypt/two  protocols=tcrypt/one,km/two;
name=crypt/three protocols=tcrypt/two,km/one;
name=udp protocols=crypt/three;
name=udpcrypttest protocols=udp;
@;
prottbl = DEFAULT;
romopt tcrypt/one EECB;
romopt tcrypt/two DECB;
romopt crypt/three ECBC;
---------------------------------
```

NB: this illustrates a version of triple DES that is not compatible with the CRYPT3 implementation, because the cipher-block-chaining is applied in every encryption and decryption step. CRYPT3 does the cipher block chaining operation after the (encryption, decryption, encryption) combination.

SHORTCOMINGS, THINGS TO BE DONE

The GETMAXPACKET and GETOPTPACKET control operations don't make appropriate corrections for any IV sent with the packet.

This version of CRYPT/CRYPT3/TCRYPT is experimental, and some software engineering and cryptographic changes have yet to be implemented. The protocol does not zero out message plaintext after encryption, or ciphertext after decryption. Our current system of one key per host is suitable for testing, but would not be appropriate for a real application.

The information passed during opening of a session includes arbitrary participants; this provides an outbound information channel for Trojan Horse client programs. The lengths of outgoing packets are another channel. The presence or absence of communication is a third channel. A Trojan Horse client can also originate known plaintext to help out a remote attacker.

The source and destination IP addresses appear in clear for every packet on the net, so traffic analysis is trivial for a properly located eavesdropper. Message length is also available to the eavesdropper. Many protocols have standard length messages they send (acknowledgements, routing updates, etc.), often with fixed, known, or easily guessable contents. These are also available to an opponent.

AUTHORS

Richard Schroeppel, David C. Schwartz, Oliver Spatscheck, Hilarie Orman, and Sean O'Malley.

NAME

CRYPT3 (A triple DES Encryption Protocol)
See the description of CRYPT, page 112.

DISTRIBUTION NOTE

This protocol will not be distributed via anonymous ftp. If interested, contact xkernel-bugs@cs.arizona.edu about obtaining this protocol.

NAME

CRYPT3IN (A triple DES Encryption Protocol)

Crypt3in does three rounds of DES in CBC mode without feedback between stages.

An IV of 8 bytes can be specified; if so, it is used as the IV for the first round and an IV of 0 is used for the second two rounds.

This algorithm requires a 24-byte key.

See the description of CRYPT, page 112.

DISTRIBUTION NOTE

This protocol will not be distributed via anonymous ftp. If interested, contact xkernel-bugs@cs.arizona.edu about obtaining this protocol.

NAME

DHECKX (Diffie-Hellman Key Exchange Protocol using Elliptic Curves)

DISTRIBUTION NOTE

This protocol will not be distributed via anonymous ftp. If interested, contact xkernel-bugs@cs.arizona.edu about obtaining this protocol.

SPECIFICATION

THIS IS EXPERIMENTAL CODE; see "Restrictions" below. DHECKX negotiates a 56-bit DES key with its peer on a remote system. The key is entered into a key manager protocol (KM, page 165), which will then provide the key to encryption modules (CRYPT, pagẽpagerefCRYPT). After negotiating the key, DHECKX becomes transparent (passing messages unchanged in either direction), so that it may be composed inside a chain of protocols.

SYNOPSIS

When a DHECKX session is opened, it opens the protocol below, and performs a GETPARTICIPANTS operation to learn the fully resolved addresses of the participants. The participants are used to open sending and receiving key manager (KM) sessions. The DHECKX protocol then sends an OFFER message to the remote system. If an appropriate reply is received within ten seconds, a key is computed and entered into both key manager sessions. The protocol then enters the transparent state. If CRYPT and KM protocols are configured in the protocol graph, they will use the negotiated key to encrypt and decrypt subsequent messages on the channel.

When the DHECKX protocol on the remote machine receives the offer message, it opens two key manager sessions and the protocol above itself. Then DHECKX computes and sends a reply, and computes the session key and enters it into the sending and receiving key manager sessions. Finally, DHECKX becomes transparent.

The advantage of the Diffie-Hellman method is that the negotiated key never appears explicitly on the channel, so an eavesdropper will have difficulty determining the negotiated key.

REALM

DHECKX is in the ASYNC realm.

PARTICIPANTS

DHECKX passes participants to the lower protocols without manipulating them. It uses the participants to open key manager sessions to enter the keys for encryption and decryption.

CONTROL OPERATIONS

DHECKX does not have any of its own control operations. All control operations are passed unchanged to the lower protocol or session.

CONFIGURATION

DHECKX expects to be configured on top of a transport protocol and a key manager. The transport protocol must preserve packet boundaries (i.e. DHECKX will not work on top of TCP). DHECKX requires a minimum packet size of 48 bytes to negotiate a key. The most convenient way to use the DHECKX protocol is to put the encryption protocol somewhere above DHECKX in the protocol graph. If the encryption protocol is below DHECKX, then it must not interfere with the DHECKX offer and reply messages. DHECKX and the encryption protocol must be configured above the same instance of KM. The key magager must have an initial key for each communicating host, obtained from a keys file. These keys are never used, but are overwritten by the DHECKX protocol.

Example of a graph.comp file:

```
--------------------------------
@;
name=simeth/0;
name=eth protocols=simeth/0;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet;
name=km/dheckxtest;
name=dheckx protocols=ip,km/dheckxtest;
name=crypt protocols=dheckx,km/dheckxtest;
name=udp protocols=crypt;
name=udpcrypttest protocols=udp;
@;
prottbl = ../../../etc/prottbl.nonstd;
--------------------------------
```

Example of a keys file, keysdheckxtest:

```
--------------------------------
4 0-8 10
0xc00c4582  0807060504030201
0xc00c4587  080706050403d213
0xc00c456e  0103040506070802
0xc00c4574  3141592653589793
0xc00c4554  2718281828459045
0xc00c4589  9876543210fedcba
--------------------------------
```

RESTRICTIONS

THIS VERSION OF DHECKX IS EXPERIMENTAL. The Diffie-Hellman method requires a source of random numbers for generating the offer and reply messages. Ideally, a hardware random number generator should be used. We have experimented with a subroutine, xkrandom, to provide the random numbers, but the code is presently turned off. In its place, fixed, compiled-in “random” values are used by the sender and receiver. THE SAME KEY WILL BE NEGOTIATED EVERY TIME THE PROTOCOL IS USED.

This behavior masks another restriction due to the present key manager, which uses key-per-host rather than key-per-host-pair. If two DHECKX sessions are opened to differing destinations, the sending key for the second session will overwrite the sending key for the first session.

This version of DHECKX provides no protection against the “man in the middle” attack, where there is an interposer in the communication path.

AUTHOR

Richard Schroeppel, Oliver Spatscheck

NAME

DHECKXLEAF

DISTRIBUTION NOTE

This protocol will not be distributed via anonymous ftp. If interested, contact xkernel-bugs@cs.arizona.edu about obtaining this protocol.

SPECIFICATION

The DHECKXLEAF protocol is part of a suite that implements a network security architecture advocated by the IETF IPSEC working group. The documents of that group describe the architecture, terminology, and algorithms.

The protocol is used to calculate the shared secret of a Diffie Hellman Key Exchange using Elliptic Curves in accordens to group 1 in PHOTURIS draft 8.

It only supports xOpen xClose and xControll calls. On an xOpen call a pseudo random number is generated and the public value of the Diffie Hellman key exchange is calculated. The controll operations are used to retrieve the public value and to calculate the shared secret. xClose will destroy the session as usual.

REMARK: The secret exponent never leaves DHECKXLEAF.

REALM

The DHECKXLEAF is in the ASYNC realm.

CONTROL OPERATIONS

The DHECKXLEAF supports the following control operations:

`DHECKXLEAF_PUBLIC_VALUE` Retrieves the public value in network byte order (session only).

Input: none

Output: Public value of current session.

Return Value: XK_SUCCESS if sucessfull.

`DHECKXLEAF_CALCULATE_SECRET` Calculates the shared secret in network byte order (session only).

Input: Public value of the remote host in network byte order,

Output: Shared secret between the current session and the remote host in network byte order.

Return Value: XK_SUCCESS if sucessfull.

`DHECKXLEAF_VALUE_SIZE` Retrieves the size of the public value in bits (session only).

Input: none

Output: Size of the public value in bits.

Return Value: XK_SUCCESS if sucessfull.

CONFIGURATION

The protocol can only be configured as leaf protocol. No protocols below it are allowed.

Shortcommings

The format of the public value and shared secret doesn't comply with PHOTURIS draft 8.
Random numbers are generated using rand().

AUTHOR

Oliver Spatscheck

NAME

DHECKXSPI (Diffie-Hellman Key Exchange Protocol using Elliptic Curves)

DISTRIBUTION NOTE

This protocol will not be distributed via anonymous ftp. If interested, contact xkernel-bugs@cs.arizona.edu about obtaining this protocol.

SPECIFICATION

THIS IS EXPERIMENTAL CODE; see “Restrictions” below. DHECKXSPI negotiates a 128-bit key with its peer on a remote system. The key is entered into a key manager protocol (KM, page 165), which will then provide the key to the SAIDMGR (SAIDMGR, page 189).

SYNOPSIS

When a DHECKXSPI session is opened, it opens the protocol below, and performs a GETPARTICIPANTS operation to learn the fully resolved addresses of the participants. The SPI and algorithm identifier are passed as local participants on an open call. The SPI is used to open a key manager (KM) sessions.

The DHECKXSPI protocol then sends an OFFER message containing the half key, the local SPI and the requested algorithm to the remote system. If an appropriate reply is received within ten seconds, the computed key and the remote SPI which is passed in the reply message and the algorithm id are entered into the key manager sessions. If the request is not accepted by the remote machine an ERR_XOBJ is returned.

When the DHECKXSPI protocol on the remote machine receives the offer message, it checks its passive map and it calls xOpenDone on the higher level protocol to retrieve a local SPI. Then DHECKXSPI computes a reply, computes the session key and enters the key, the remote SPI and the algorithm ID into the key manager sessions. xDemux of the higher level protocol is called with the local SPI as message before the reply message is send.

The advantage of the Diffie-Hellman method is that the negotiated key never appears explicitly on the channel, so an eavesdropper will have difficulty determining the negotiated key.

REALM

DHECKXSPI is in the ASYNC realm.

MAPS

The passive map is based on the algorithm identifier. Therefore the higher level protocol can set the algorithms which it will support using xOpenEnable calls.

The active map only contains values during the key negotiation.

PARTICIPANTS

DHECKXSPI expects the remote IP address as remote participant and the local SPI, the algorithm id and the local IP address on the local participant stack. The local IP address is optional.

CONTROL OPERATIONS

DHECKXSPI does not have any of its own control operations. All control operations are passed unchanged to the lower protocol or session.

CONFIGURATION

DHECKXSPI expects to be configured on top of a transport protocol and a key manager. The transport protocol must preserve packet boundaries (i.e. DHECKXSPI will not work on top of TCP).

DHECKXSPI has to be used below a protocol which returns a SPI as result value on a xOpenDone call.

Example of a graph.comp file:

```
---------------------------------
@;
name=simeth/0;
name=eth protocols=simeth/0;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet;

name=km/a;
name=km/b;
name=km/haval;
name=km/md5;
name=km/crypt;
name=km/md5crypt_md5;
name=km/md5crypt_crypt;

name=dheckxspi protocols=ip,km/b;
name=saidmgr protocols=dheckxspi,
km/a,km/b,km/haval,km/md5,km/crypt,km/md5crypt_md5,km/md5crypt_crypt;
....

@;
prottbl = ../../../etc/prottbl.nonstd;
---------------------------------
```

RESTRICTIONS

THIS VERSION OF DHECKXSPI IS EXPERIMENTAL. The Diffie-Hellman method requires a source of random numbers for generating the offer and reply messages. Ideally, a hardware random number generator should be used. We have experimented with a subroutine, xkrandom, to provide the random numbers, but the code is presently turned off. In its place, fixed, compiled-in “random” values are used by the sender and receiver.

This version of DHECKXSPI provides no protection against the “man in the middle” attack, where there is an interposer in the communication path.

AUTHOR

Oliver Spatscheck, Richard Schroeppel

NAME

DHKX (Diffie-Hellman Key Exchange Protocol)

DISTRIBUTION NOTE

This protocol will not be distributed via anonymous ftp. If interested, contact xkernel-bugs@cs.arizona.edu about obtaining this protocol.

SPECIFICATION

THIS IS EXPERIMENTAL CODE; see "Restrictions" below. DHKX negotiates a 56-bit DES key with its peer on a remote system. The key is entered into a key manager protocol (KM, page 165), which will then provide the key to encryption modules (CRYPT, pagẽpagerefCRYPT). After negotiating the key, DHKX becomes transparent (passing messages unchanged in either direction), so that it may be composed inside a chain of protocols.

SYNOPSIS

When a DHKX session is opened, it opens the protocol below, and performs a GETPARTICIPANTS operation to learn the fully resolved addresses of the participants. The participants are used to open sending and receiving key manager (KM) sessions. The DHKX protocol then sends an OFFER message to the remote system. If an appropriate reply is received within ten seconds, a key is computed and entered into both key manager sessions. The protocol then enters the transparent state. If CRYPT and KM protocols are configured in the protocol graph, they will use the negotiated key to encrypt and decrypt subsequent messages on the channel.

When the DHKX protocol on the remote machine receives the offer message, it opens two key manager sessions and the protocol above itself. Then DHKX computes and sends a reply, and computes the session key and enters it into the sending and receiving key manager sessions. Finally, DHKX becomes transparent.

The advantage of the Diffie-Hellman method is that the negotiated key never appears explicitly on the channel, so an eavesdropper will have difficulty determining the negotiated key.

REALM

DHKX is in the ASYNC realm.

PARTICIPANTS

DHKX passes participants to the lower protocols without manipulating them. It uses the participants to open key manager sessions to enter the keys for encryption and decryption.

CONTROL OPERATIONS

DHKX does not have any of its own control operations. All control operations are passed unchanged to the lower protocol or session.

CONFIGURATION

DHKX expects to be configured on top of a transport protocol and a key manager. The transport protocol must preserve packet boundaries (i.e. DHKX will not work on top of TCP). DHKX requires a minimum packet size of 136 bytes to negotiate a key. The most convenient way to use the DHKX protocol is to put the encryption protocol somewhere above DHKX in the protocol graph. If the encryption protocol is below DHKX, then it must not interfere with the DHKX offer and reply messages. DHKX and the encryption protocol must be configured above the same instance of KM. The key magager must have an initial key for each communicating host, obtained from a keys file. These keys are never used, but are overwritten by the DHKX protocol.

Example of a graph.comp file:

```
---------------------------------
@;
name=simeth/0;
name=eth protocols=simeth/0;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet;
name=km/dhkxtest;
name=dhkx protocols=ip,km/dhkxtest;
name=crypt protocols=dhkx,km/dhkxtest;
name=udp protocols=crypt;
name=udpcrypttest protocols=udp;
@;
prottbl = ../../../etc/prottbl.nonstd;
---------------------------------
```

Example of a keys file, keysdhkxtest:

```
---------------------------------
4 0-8 10
0xc00c4582  0807060504030201
0xc00c4587  080706050403d213
0xc00c456e  0103040506070802
0xc00c4574  3141592653589793
0xc00c4554  2718281828459045
0xc00c4589  9876543210fedcba
---------------------------------
```

RESTRICTIONS

THIS VERSION OF DHKX IS EXPERIMENTAL. The Diffie-Hellman method requires a source of random numbers for generating the offer and reply messages. Ideally, a hardware random number generator should be used. We have experimented with a subroutine, xkrandom, to provide the random numbers, but the code is presently turned off. In its place, fixed, compiled-in “random” values are used by the sender and receiver. THE SAME KEY WILL BE NEGOTIATED EVERY TIME THE PROTOCOL IS USED. This behavior masks another restriction due to the present key manager, which uses key-per-host rather than key-per-host-pair. If two DHKX sessions are opened to differing destinations, the sending key for the second session will overwrite the sending key for the first session.

This version of DHKX provides no protection against the “man in the middle” attack, where there is an interposer in the communication path.

AUTHOR

Richard Schroeppel

NAME

DHKXLEAF

DISTRIBUTION NOTE

This protocol will not be distributed via anonymous ftp. If interested, contact xkernel-bugs@cs.arizona.edu about obtaining this protocol.

SPECIFICATION

The DHKXLEAF protocol is part of a protocol suite that implements a network security architecture advocated by the IETF IPSEC working group, the RFC1825 protocols AH and ESP (see RFC1825 page 175). The documents of that group describe the architecture, terminology, and algorithms.

The protocol is used to calculate the shared secret of a Diffie-Hellman Key Exchange using modular exponentiation in accordance with the group identity 2 in PHOTURIS (see page 207) draft (working group draft number 8, winter of 1995).

It only supports the xOpen, xClose, and xControl calls. On an xOpen call a pseudo random number is generated and the public value of the Diffie-Hellman key exchange is calculated. The control operations are used to retrieve the public value and to calculate the shared secret. xClose will destroy the session as usual.

REMARK: The secret exponent never leaves DHKXLEAF.

REALM

The DHKXLEAF is in the ASYNC realm.

CONTROL OPERATIONS

The DHKXLEAF supports the following control operations:

`DHECKXLEAF_PUBLIC_VALUE` Retrieves the public value in network byte order (session only).

> **Input:** none
>
> **Output:** Public value of current session.
>
> **Return Value:** XK_SUCCESS if sucessfull.

`DHECKXLEAF_CALCULATE_SECRET` Calculates the shared secret in network byte order (session only).

> **Input:** Public value of the remote host in network byte order,
>
> **Output:** Shared secret between the current session and the remote host in network byte order.
>
> **Return Value:** XK_SUCCESS if sucessfull.

`DHECKXLEAF_VALUE_SIZE` Retrieves the size of the public value in bits (session only).

> **Input:** none
>
> **Output:** Size of the public value in bits.
>
> **Return Value:** XK_SUCCESS if sucessful.

CONFIGURATION

The protocol can only be configured as leaf protocol. No protocols below it are allowed.

SHORTCOMINGS

Random numbers are generated using rand().

AUTHOR

Oliver Spatscheck

NAME

DSS (NIST Digital Signature Standard)

SPECIFICATION

The DSS protocol pushes a 40 byte cryptographic signature onto the front of a message. On the incoming side, it pops a 40 byte signature, and checks if it's consistent with the SHA checksum computed for the rest of the message. The code for DSS is based on the NIST specification.

DSS is designed to be composed over any datagram protocol.

This version of DSS is experimental. It hasn't been connected to the key manager; it uses one compiled-in identity for both signing and checking signatures. See the RESTRICTIONS section for further information.

SYNOPSIS

When a message is pushed to a DSS session, an SHA checksum is computed for the message. The checksum is "signed", giving a 40 byte signature. The signature is pushed onto the front of the message. When an incoming message is popped to a DSS session, the 40 byte signature is popped, and an SHA checksum is computed for the rest of the message. The signature is checked for consistency with the checksum. If tracing is enabled, a message is printed reporting whether or not the signature is valid. No special action is taken on messages with an invalid signature; they are passed on as usual to the next higher protocol.

REALM

DSS is in the ASYNC realm.

PARTICIPANTS

DSS passes participants to the lower protocols without manipulating them.

CONTROL OPERATIONS

DSS recognizes the following control operations; all others are passed unchanged to the lower protocol or session.

GETMAXPACKET and GETOPTPACKET: The packet size returned by the lower protocol/session is diminished by 40, and passed upward.

IP_PSEUDOHDR: This control operation turns on the IP pseudoheader length-fixup flag, either for a session or the entire protocol. The control operation is also passed to the lower session or protocol. See IP (page 156) for an explanation of this kludge.

CONFIGURATION

DSS expects to be configured on top of a transport protocol. The transport protocol must preserve packet boundaries (i.e. DSS will not work on top of TCP).

Example of a graph.comp file:

```
---------------------------------
@;
name=simeth/0;
name=eth protocols=simeth/0;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet;
name=dss protocols=ip;
name=udp protocols=dss;
name=udptest protocols=udp;
@;
prottbl = ../../../etc/prottbl.nonstd;
---------------------------------
```

RESTRICTIONS

The security of the signature scheme depends on discrete logs being hard for 512 bit primes, and on the security of the SHA hash method. A cryptographic random number is required for each message signed. Since we don't have a hardware random number generator handy, I've used the GNU random number function as a stub. If an adversary is able to determine the random number that was used for signing a single message, he can forge messages. DSS uses the GNU bignum routines, and no special care has been taken to erase intermediate values before returning memory to the general pool.
This version of DSS is experimental. It hasn't been connected to the key manager; it uses one compiled-in identity for both signing and checking signatures.
DSS has not been tested on little-endian machines, or between machines of differing endianness.

ACKNOWLEDGMENT

DSS uses the GNU multiprecision package to do its bignum arithmetic.

AUTHOR

Richard Schroeppel

NAME

Elliptic Curve Groups

SPECIFICATION

The algorithm for elliptic curve group operations is described here. This supports the Diffie-Hellman key exchange protocol, DHECKX, but the algorithm implementation is interesting enough in its own right to merit separate documentation. The source file can be compiled as a standalong program, or as part of an xkernel protocol. This document describes the standalone use.

The file `util-noforn/ell.c` implements operations with elliptic curves over a finite field. These operations can be useful in cryptography, usually for exchanging keys.

I recommend Al Menezes' book "Elliptic Curve Public Key Cryptosystems" for an introduction to elliptic curves. Any good algebra book will explain finite fields. I also recommend our paper from Crypto '95 (in Springer LNCS #963), or our tech-report (`ftp://ftp.cs.arizona.edu/reports/1995/TR95-03.ps`).

This documentation uses the phrases "field element" and "polynomial" interchangeably.

BUILDING

The compile instructions are at the front of the source file. The file `ell.examples` contains sample compilations. The MAIN switch is used for compiling a standalone program. The ALPHA switch selects 64bit code. There is one assembly language routine for the Alpha. The Sparc compile takes several minutes.

Example of compiling for the Sparc:

```
-------------------------------------------------------
leibniz% gcc2 -O9 -o ell-960117-gcc2O9b -DMAIN ell.c
-------------------------------------------------------
```

The name of the binary program, ell-960117-gcc2O9, includes the date, compiler, and optimization level.

Running the program. An example:

```
-------------------------------------------------------
leibniz% time ell-960117-gcc2O9 -m55 100000
GF155 here.
Multiply test M55, 155*155 bits, 100000 times:
07531fca 53579ace ff003742 98765432 abcdef12
05371fac abdecf12 35795ace f003f742 98754632
001a94d3 ce915148 99c5bf6f 64daaef0 0f620c12 d6a32b04 d25a6f7a
   aeb1a2e5 c59b9857 71e42144
11.2u 0.0s 0:12 91% 0+336k 0+1io 0pf+0w
-------------------------------------------------------
```

This test is for timing the multiplication of two field elements. The "-m55" switch selects the test; the number 100000 says to run the test 100000 times. (This number is required.) The compiled-in inputs are 155 bits each; the product is 309 bits. From this test, we learn that the time per multiplication is 11.2sec/100000 = 112 microseconds.

The file ell.examples contains more examples of running the program. There are about 100 routines that can be called from top level. See the code in "main" for a complete list, and the -names that call them. Most of the tests will run without arguments (other than the repetition count). To determine if a test allows other arguments, it's necessary to examine the code in "main" that calls the test. If you run the program without arguments, or forget to include the count, it runs a few basic tests.

Timing note: The "main" program includes a fair amount of setup time (10-100 msec). I always run my timing tests for several seconds to minimize the (relative) effect of the setup.

Programming Matters

This is a research tool, and it contains much obsolete code. This code is useful in explaining the thread of ideas that leads to the more complicated "optimized" routines. It may also be helpful when creating systems for other machines, with different time/memory/cache/register constraints. The older routines tend to be shorter and simpler. When several low-level routines with the same functionality exist, a #define is used to select which routine is used by higher level code.

To make a production system, strip out all the inessential code. The top-level routine could be either "testecv" or "testdhtm". The Diffie-Hellman key exchange protocol requires the program to choose random numbers to use as the point multipliers. I have instead used fixed, compiled-in values. A production version will need to include a suitable source of random numbers.

I've used global variables and arrays in many places, so the code isn't reentrant.

This program was written to run on two particular machines, the Sun Sparc model IPC, with 32bit words, running at 25MHz, under Sunos 4.1.1; and the DEC Alpha series, with 64bit words, under DEC Unix (OSF/1 V3.0); our timings were run on a 175 MHz machine. A long time (and many versions) ago, the program was compiled and run on a 66 MHz 486 machine under Mach+Unix, and on a DEC Mips 25 MHz workstation running Mach+Unix. The times for the basic elliptic curve operations were within 10Sparc times.

The Sparc has a 64KB instruction cache, more than enough for our purposes. The Alpha has an 8KB on-chip primary instruction cache, and getting the right code into that cache was a headache. [The Alpha has two 32bit instructions in each word, so the multiply and inversion have to fit in 2048 instructions. Recent versions of the Alpha chip have a 16KB Icache, ameliorating this problem.] We tried both the native compiler and gcc2 on the Alpha: The native compiler produced slightly faster code, but insisted on arranging the routines to its own taste. The low-level code ran faster, but we could find no way to get the correct code for doing a full ecurve calculation (a point multiplication) into the cache without overlap, so the time for a real unit of work was worse than with GCC2. GCC2 was dumber about rearranging the routines, so we were able to get the critical code into the cache unoverlaped simply by removing all extraneous code. The #define ECV switch eliminates everything that isn't needed to perform an elliptic curve point multiplication, and uses a stripped-down top-level. The MACRO64 switch causes some routines to be inlined; it's currently turned on within the code (under an #ifdef ALPHA). Use ECV and MACRO64 only on the Alpha.

Porting: The C code is pretty much vanilla. Watch for problems with unsigned compares, and shifts of unsigned quantities. I've used the capability of running together strings in the print statements; older C compilers don't like this. The example file will be helpful in testing a port. The program will break badly if the data type (unsigned long) is other than 32 or 64 bits. Start with the lowest level routines and work up. To get the fastest results for your machine, it may be necessary to change the macros which select among the low-level routines. The principal code selection macros are MULT, SQUARE, REDUCE155, MODINV, Edouble, and Epow. There are some secondary selections that only matter when particular primary selecions are made.

Data Structures

Most arrays and variables that represent field elements are (unsigned long). Numerical variables, counters, etc. are generally (long).

The polynomial representation (for elements of $GF[2^{155}]$) is five consecutive 32bit words of memory (three 64bit words for the Alpha). The lower degree powers of u go into the lower-order ("rightmost") bits. $u^0 = 1$ goes into the low-order bit with numerical value 1. The high- order bit represents u^{31}. Polynomials are usually represented by small arrays of five words (Alpha: three), with array[0] holding u^{31} ... u^0, and array[4] holding $u^{154}...u^{128}$. The unused high-order 5 (Alpha: 37) bits of the last array word are 0, unless the code says otherwise. Sometimes, a polynomial is represented by a group of named variables, such as A0...A4. In this case, A0 holds the low degree terms $u^{31}...u^0$, and A4 holds the high degree terms $u^{154}...u^{128}$. Some computations require double-sized arrays to hold polynomials of twice the normal degree. In this case, the polynomial terms are placed consecutively: the bit representing u^{155} is right next to the bit for u^{154}, and the "unused 5 bits" in array[4] are occupied by $u^{159}...u^{155}$. Of course there will be 10 unused high-order bits in array[9], and they will be 0. On the Alpha, array[2] will hold bits for $u^{191}...u^{128}$, while array[4] has 10 unused high order bits. A double-length array will usually have a wasted word array[5]. A double-length array is usually named with a "d" at the end, or with an upper-case letter at the end. Some routines need double length arrays as arguments; the comments tell all. The program has a fair number of compiled in test values (field elements), and the FIVE macro is used to create the array elements. The polynomial powers are given with the lowest degree part first: In FIVE(A,B,C,D,E), A represents $u^{31}...u^0$, and E represents $u^{155}...u^{128}$.

Program Organization

The routines are generally in the order they were written, with low- level primitives at the front of the file, and the top-level "main" routine at the end.

Many routines are directly followed by a test routine. The test routine will call the target routine several times with either canned arguments or arguments supplied by the user from the command line. The test routine is called by "main". Sometimes the results are checked and/or printed.

The last section of the program, before the top-level main routine, is a klunky implementation of (most of) the Schoof algorithm, which determines the number of points on a particular curve. The Schoof algorithm is explained in Menezes' book. My implementation isn't an exact match to his explanation. With a separate Lisp program to fill in some of the missing details, I've determined sizes for a couple of groups.

The top-level of the program is a collection of timing tests, which time and/or test most of the routines. The interface to the program is a simple command line with arguments. The command line says which test to run, how many times, and with what arguments. The program runs the test, may print an answer, and exits. The user is responsible for wrapping the command in a Unix "time" command to determine the actual execution time for a routine.

The most important routines, from the viewpoint of execution time, are those for multiplying two polynomials, and the inversion routine. I wrote many versions of these routines, and selected the fastest based on timing tests. (Which is fastest depends on the processor - if you are working on a new processor, you probably want to rerun the timing tests, and perhaps change the macros that select which routines are used in "production" code.)

Most of the program works with field elements from $GF[2^{155}]$, defined by the irreducible polynomial $u^{155} + u^{62} + 1$. The elliptic curve used is defined by the equation $Y^2 + XY = X^3 + A6$, where A6 is a field element. Generally the curve defaults to a compiled-in value if no other is specified. The routines Epowknown, testdhtm, elgamalsetup, and elgamalEnc also implicitly assume a particular curve, because the table of point values at ectabxy is built in.

The program is committed to the field $GF[2^{155}]$, and to the polynomial $u^{155}+u^{62}+1$ as the magic trinomial $T(u)$. It is only slightly committed to the choice of elliptic curve and generating point.

The particular elliptic curve is usually specified by giving a point on the curve, which determines the curve parameter A6. A6 must not be 0. Most of the paths from startup to the curve code include a setup for A6 (the setup routine is "a6calc"). You may use other means of setting up A6, so long as you also set up A60. For efficiency, the present code assumes that A6 is only 20 bits long, which restricts the starting points. If you limit the starting coordinates X,Y so that $X < 128$ and $Y < 1024$ (decimal), then A6 will be at most 20 bits. There are lots of larger points which will also evaluate to a small A6, but it's trickier to find them. If you need to use an A6 larger than 20 bits, modify the routine selected by MULTKRED (either mult15c20xred or mult13c20xred), or rewrite the point doubling routine. The point addition routine doesn't use A6, except when it calls the doubling code.

Elliptic Curves for the Disinclined

(Read this even if you don't want to know anything about elliptic curves.)

What does this program do? - Very basic.

The program mostly operates on 155 bit quantities. These are called field elements or polynomials. They are represented in 5 computer words on 32bit machines, and 3 words on 64bit machines. The field elements are not ordinary numbers, but they can be used in some of the same ways. There are operations that add and multiply field elements, and take reciprocals. Multiplication of two field elements gives a 309bit result, which is given to a "reduce" routine to turn it back into a 155bit quantity. Negation of a field element is a nop, and subtraction is the same as addition.

The second data type is called an "elliptic curve point" and is a pair of field elements, just like an ordinary point in the plane is made up of a pair of real numbers. Not just any old pair of field elements will do - in order to qualify as a "point on the curve", the two elements have to satisfy a special equation (using the special addition and multiplication of field elements). Different equations give different curves (i.e., different sets of points). Each curve has roughly 2^{155} points, but getting the exact count is a challenge.

Elliptic curve points have their own kind of addition defined. This addition has nothing to do with ordinary arithmetic addition, or with addition of field elements, or addition of vectors. The only excuse for calling the operation "addition" is that it satisfies some of the same algebraic laws as ordinary addition. Once this special addition is defined, we can talk about a "multiple of a point P". If P is a point, then $2P$ is just $P+P$; and $3P$ means $P+P+P$ (which is the same as $P+2P$); and $4P$ means $P+P+P+P$. Larger multiples can be calculated in several different ways: $4P$ can be done as $P+3P$, or $2P+2P$. The program usually uses the straightforward double-and-increment scheme. It's easy to calculate $-P$ from P, so sometimes points are subtracted. (This would be a good way to compute $127P$.)

The key exchange starts with a particular elliptic curve and point P on that curve. (There are default choices compiled in; you can change them by giving arguments to the program when you run it.) We assume all parties to the key exchange know the particular curve in use, and the starting point P. The program should choose a random 155bit number K (an ordinary integer!); it then calculates KP. This point is another pair of (155bit) field elements. This pair is supposed to be sent over the network to another host, which picks its own 155bit random number L, computes LP, and sends it back to us. We can then compute the Kth multiple of the point LP, which we denote by KLP. The remote host computes LKP. These two points are equal, which gives us 310 bits of "shared secret" to work with.

In the actual program, the numbers K and L are compiled in, and the program pretends it is handling both ends of the transaction. To actually turn this program into a key-exchange scheme, you must add a source of random numbers, and network communications software.

Elliptic Curves for the Less Disinclined

(Read this if you want to muck around with the code.)

What does this program do? - More details.

The basic "numbers" that the program works with are 155bit quantities. These quantities represent polynomials of degree up to 154, with coefficients either 0 or 1 (only). The basic operations on the polynomials are addition, and multiplication, with all coefficients reduced modulo 2. This means that, for example, $X + X = 2X = 0$, and that $(X + 1)^2 = X^2 + 2X + 1 = X^2 + 1$. In this system, addition of the polynomials is done by Xoring the bit representations. Multiplication of two polynomials is done by the usual shift-and-conditionally-add scheme for multiplying binary numbers, except that the addition step is changed to an xor step.

Because we interpret the coefficients mod 2, addition is the same as subtraction; every polynomial is its own negative.

Complication: Since the multiplication step can create polynomials of degree up to 308, we always reduce the result of a multiplication modulo the "magic trinomial" $T(u) = u^{155} + u^{62} + 1$. This forces the degree back down to 154 or less. For example, u^{200} is reduced to $u^{107} + u^{45}$. (The rule is that u^{155+N} is replaced with $u^{(62+N)} + u^N$, which can be done with shift and xor manipulations. Two iterations are needed when $N > 92$.)

It happens, because the magic trinomial T is irreducible mod 2, that every non-zero polynomial P of degree $\leq$ 154 has a unique reciprocal P' polynomial, which satisfies the equation $P * P' = 1(modT)$. For example, the reciprocal of u^3 is $u^{152} + u^{59}$. If these two polynomials are multiplied the result is $u^{155} + u^{62}$. After the modular reduction, the result is 1.

In consequence, the set of polynomials of degree $\leq$ 154 forms a mathematical FIELD, in which the operations of addition, subtraction, multiplication, and division follow most of the usual rules of algebra. This field has exactly 2^{155} elements - there are exactly 2^{155} polynomials; and they can be represented as quantities of 155 bits.

The first part of the program is concerned with implementing the operations for the field efficiently, especially multiplication and reciprocals. We've done very well in that regard.

The next concept required is trickier. Consider the equation in two varibles X and Y: $Y^2 + XY = X^3 + A$. If A were a particular real number, this equation would be a curve in the $X - Y$ plane. Particular combinations of X,Y that satisfied the equation would be points on the curve. We make a mathematical leap, and extend the notions of curve and point to the field we described above. We say that X and Y will be elements of the field (polynomials of degree up to 154, with coefficients only 0 or 1) and that A is a particular field element, perhaps $u^{19} + u^7 + 1$. The set of X,Y pairs which satisfy the equation are called "points" on the "curve". This is quite a stretch – we've replaced ordinary numbers with these weird polynomials, we are using mod 2 addition and multiplication for the coefficients, and the equation is counted as true as long as long as it's true when the two sides are both reduced modulo the magic trinomial T. The real number curve has an infinite number of points, while in the field case we could in theory write down a complete list of the solutions. Moreover, the real number case is continuous: you can move along the curve, usually changing X and Y by a little bit. The notion of points being "nearby" is pretty much discarded in the finite field case. If X is changed by 1 bit, Y will be completely different.

So what's the point of all this? There's a piece of magic, discovered centuries ago, whose implications were fully realized only in the last decade. Mathematicians were interested in finding solutions to equations like ours where X and Y were both rational numbers. Long ago, mathematicians discovered that you could use two solutions of the elliptic curve equation to generate a third solution. It turns out if you draw a line through two points on the curve, it usually intersects the curve in exactly one more point. When analytic geometry was invented, it turned out that that third solution could be specified by an algebraic formula. If you take the third point and reflect it through the curve's axis, you get a fourth point. This fourth point usually isn't on the same line as the first three points, so you can do the line trick again to get more solutions. In favorable cases, this scheme lets you use two starting solutions to create an infinite set of other solutions. If you've only got one solution, another trick comes to the rescue: you can draw a tangent to the curve (as if you had two equal solutions!) and there will still be a "third" point of intersection of the tangent with the curve. This third point can be reflected through the curve axis as before, and away you go.

Using analytic geometry, the operation of taking two points, finding the third intersection, and reflecting it through the axis to get a fourth point on the curve, can be specified entirely in terms of algebraic formulas. This has two interesting consequences: The first is that we can carry out the steps by evaluating the formulas, even when the geometry makes no sense. In particular, we can do it in our field $GF[2^{155}]$. The second consequence is even more interesting: The manipulation that takes two points and produces a fourth is called "addition of elliptic curve points" because it has many of the properties of ordinary addition. [This is in spite of the fact that the operation has very little to do with ordinary vector addition of points, or addition of the coordinates. Mathematicians are pretty free when it comes to reusing names.] It's pretty obvious that swapping the two starting points won't change the answer, so the "addition" is commutative. But there's extra magic here: the addition turns out to also be associative - you can add up three points in any order, and always get the same answer. There's no easy way to explain why this works, but it's possible to carry out the algebra and check it. (It was probably discovered when people trying to generate points on a curve kept getting the same answers repeatedly, and had to be careful with the starting points in order to get new points.) There's one more operation we need to mention: reflecting a point through the curve axis gives the "negative" of the point. There's also a "zero" point, although it's kind of strange - both coordinates X and Y are taken to be "infinity". This happens when the line joining the two input points doesn't intersect the curve; then the answer is taken to be the zero point. When you are adding a point to itself ("doubling" a point), you use the formula for the tangent to the curve – even though the notion of differentiation is meaningless in a finite field. (This complicates the proofs of associativity no end, because of all the special cases for zero and doubling.)

So we have an addition operation defined for curve points, and it works for curves with field elements from $GF[2^{155}]$. What good is it?

We've finally reached the payoff. The Diffie-Hellman Key Exchange method was originally defined to use arithmetic modulo a prime P, but it can be generalized to work in any group - for example, the group of points of an elliptic curve defined over $GF[2^{155}]$ with a bizarro addition rule. The payoff is that elliptic curve systems appear to be immune to the most effective cryptanalytic attack that works on regular Diffie-Hellman. [The underlying reason for this was mentioned above: the concept "nearby" doesn't work for elliptic curves over finite fields.] In regular Diffie-Hellman, you pick a prime modulus P, and a generator G (both ordinary integers), and the parties to the key exchange calculate various powers of G (mod P). The calculations use multiplication (mod P) as the basic operation. To calculate G^X, you examine the binary representation for X, and do a series of squarings and some multiplications (all operations mod P). In the Elliptic Curve variation, you choose a curve by picking the element A, and find some curve point (X,Y) as the generator. The parties calculate some numeric multiple of the (X,Y), such as 65537 (X,Y), as the work of the key exchange. The multiples are calculated by a series of point doublings an additions. [65537 (X,Y) would be done with 16 doublings and one addition.] In either scheme, the bits of the final answer are used to create a secret key that is used for further communication.

Why is it worth doing Diffie-Hellman in this oddball group? Because the main attack that works on ordinary Diffie-Hellman doesn't work against our oddball variation, the numbers involved can be substantially shorter. We pay a price in code complexity, but at the end of the day, our key exchange runs faster because the number of operations is smaller. In the 155bit case, our security is roughly the same as ordinary Diffie-Hellman with a 1024bit modulus, and our speed is three times as fast. Compared to ordinary Diffie-Hellman with a 512bit modulus, we have similar speeds but substantially stronger security.

The program's compiled-in curve is based on the point (0x7b, 0x1c8), which has A60 = 0x7338f. The group order for this curve is

45671926166590716193865565914344635196769237316

which is 12*Prime. The point (0x7b, 0x1c8) is a cube (the triple of a generator of full order). The best possible group order with this class of curve equations is 4*Prime. (The size of the group is approx- imately fixed; the figure-of-merit is log2(largest-prime-factor)/2. 4*Prime would have a slightly better figure of merit. Our curve has f.o.m. 75.7, versus the best possible 76.5.

Two Improvements Which Could be Made

The routine for inverting a field element, modinvh2, has two improvements possible. The first is that only some of the length combinations of the variables are possible or likely, and the code could be shortened by removing impossible combinations and coalescing unlikely ones. This is of interest mainly on the Alpha, because the primary instruction cache is tight: If the inversion routine were shorter, we would have room to use Oliver's Spatscheck's best multiplication routine – presently, it won't fit in the instruction cache with the hogsized inversion routine. This might lead to a 10% time improvement.

The second routine is important for both processors: The code does single bit shifts of the internal variables representing field elements. It would be better to test for a few low-order 0s in the A variable, and shift by several bits at once. This should lead to a 10-20% improvement. If the first improvement is made, shortening the code, then the second improvement shouldn't lengthen the code excessively.

Modifying the Program to Prevent the Kocher Timing Attack

If exponents are not reused in Diffie Hellman style key exchanges, the Kocher timing attack doesn't work. If an exponent is reused in many key exchanges, the timing attack must be thwarted. One way to prevent timing attacks is called blinding: The basic computation is modified in some way that produces the correct answer, but with the details of the modification invisible to the attacker. For example, the operation used in Elliptic Curve Key Exchange is to compute a multiple K of a curve point P, giving KP. A simple blind is to pre-select a random curve point Q, and pre-compute KQ. When the curve protocol wants to compute KP, instead compute $K(P+Q) - KQ$. The extra time in the protocol is minimal: an extra curve addition to compute P+Q, and an extra curve subtraction of KQ. If Q is changed only occasionally, the amortized extra cost of computing KQ is small. For example, it might be appropriate to select Q at the same time as K.

To compute a random curve point, choose a random X coordinate and solve for Y. This will work 50% of the time – half the Xs have no Ys. (The other Xs have two Ys, giving a curve point and its negative.) Check that X,Y satisfies the curve equation, and try another X if not. The routine "ysolve" solves for Y from X. Be sure that your random source is really not visible to an opponent – random numbers based on the time of day usually aren't a safe choice.

Acknowledgement

Oliver Spatscheck wrote some of the assembly code for the DEC Alpha, and contributed an important idea in optimizing the Alpha Xor Multiply.

AUTHOR

Richard Schroeppel

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

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

NAME

GROUP MEMBERSHIP PROTOCOL: grpmem - phase 1

SYNOPSIS

Grpmem implements a simple membership service based on a single (possibly multi-homed) owner and a set of IP addresses of members. The view of the groups is kept in a key manager that can be accessed by other protocols that may need group information. Internally to the membership protocol, each group is handled as an independent session, that can be used by an upper protocol for control purposes.
The owner notifies the members of each change to the group and also provides periodic notification about the state of each group it owns. This is done to avoid problems with lost messages (sits on top of UDP).
Clients keep timers for updates from the owner. If no updates are received for some interval they may send query messages to server to try to get a more recent description of the group. As an option, the protocol can be configured to delete a group if no answer is received for a number of consecutive queries.
The protocol can be configured (at compile time) to operate with point-to-point or broadcast messages, what may provide slightly different behaviors. When operating with point-to-point messages members can be in any network having a path to the owner, and only the members of a group know about the existence and elements of the group. When operating with broadcast messages, groups are limited to the physical networks to which the owner is directly connected. In this case, all hosts in those networks that run the membership protocol keep track of the members of all groups in that network, even it they are not members of the group.
A group is created at initialization time based on information in the rom file as described in the CONFIGURATION section, or when a session is opened with a group id not known to the local host.
The protocol takes the top of the local part of the participant list as a group identifier (an integer of type IDTYPE, currently long). If the group is not known to the protocol on that host, it assumes a new group must be created with that identifier and having the local host (identified by all its interfaces) as the owner. In any case, the corresponding session is returned and can be used in control operations, for example to add or remove members.

PARTICIPANTS

As just described, grpmem interprets the top of the participant stack as the identifier of a group to be created or just open for queries.

CONFIGURATION

Currently, the protocol recognizes three options in the rom file:

- UDP port to be used by the protocol in all hosts. The following line shows that could be set to 2233 (it should be the same on all hosts).

  ```
  grpmem port 2233
  ```

- Interval at which updates must be sent to all members of the groups owned by a host. It is represented in seconds (the default is 30 seconds). The following example sets the interval to one minute:

```
grpmem update 60
```

- Groups that are owned by the host. This information must be present only in the owner's rom file. The pre-defined update scheme is used to notify other hosts of the existence of the group. It must list the group id (an integer) and the list of IP addresses of the members of the group.

  ```
  grpmem rule grpadd 1001 192.12.69.1 192.12.69.2 192.12.69.3
  ```

  This creates the group 1001, owned by the local host and containing the hosts identified by the three IP addresses as members.

CONTROL OPERATIONS

There are some control operations allowed on the protocol to handle entire groups, and others on sessions to perform operations on a given group.

The following operations are allowed on the protocol:

`GRPMEM_CNTGRP`: Used by the protocol above grpmem to know how many groups are currently known to the local host.

Input: none

Output: `int`

`GRPMEM_ALLGRP`: Fills in an array of IDTYPE with the identifiers for all groups known to the local host.

Input: none

Output: `int[GRPMEM_CNTGRP];`

`GRPMEM_DELGRP`: By design option a group, although created when the first xOpen on it is done, is not deleted when the session is close. This control operation can be used to delete a group.

Input: `IDTYPE`

Output: none

The following are the session operations, which affect a single group:

`GRPMEM_CNTGRP`: Used by the protocol above grpmem to know how many groups are currently known to the local host.

`GRPMEM_CNTMEM`: return the number of members currently registered in a group.

Input: none

Output: `int`

`GRPMEM_ALLMEM`: Fills in an array with the IP addresses of the member of the group represented by that session.

Input: none

Output: `IPHost[GRPMEM_CNTMEM]`

`GRPMEM_CNTOWN`: Return the number of IP addresses associated with the owner of the group.

Input: none

Output: `int`

`GRPMEM_GETOWN`: Fills in an array with the IP address(es) of the owner.

Input: none

Output: `IPHost[GRPMEM_CNTOWN]`

`GRPMEM_ADDMEM`: If the local host is the owner of the group, the list of IP addresses passed in the control operation is added to the group and the new state of the group is sent to all members. The list of addresses must be terminated with the address 0.0.0.0 . The protocol just ignores requests to add a member that is already in the group.

Input: `IPHost[]`

Output: none

`GRPMEM_DELMEM`: If the local host is the owner of the group, all IP addresses in the list are removed from the group, if currently in it. Attempts to remove members that are not in the group are just ignored. Again, the list of addresses must be terminated by 0.0.0.0 .

Input: `IPHost[]`

Output: none

AUTHORS

Dorgival Guedes and Hilarie Orman

NAME

HAVAL (One-Way Hashing Algorithm)

SPECIFICATION

For outgoing messages, the HAVAL protocol calculates a 32 byte checksum and pushes it onto the front of the message. For incoming messages, a 32 byte checksum is popped from the front of the message, and compared against the computed checksum of the rest of the message. No special action is taken when the compare fails; the shortened message is passed on as usual to the next higher protocol. If tracing is enabled, a message is printed when the compare fails. There is an option to use an implicit prefix at the front of each packet when computing the checksum; separate prefixes may be used for incoming and outgoing packets.

The checksum algorithm is the HAVAL algorithm as described in TR 93-13 of the University of Wollongong, Australia.

REALM

HAVAL is in the ASYNC realm.

PARTICIPANTS

HAVAL passes participants to the lower protocols without manipulating them.

CONTROL OPERATIONS

HAVAL recognizes the following control operations; all others are passed unchanged to the lower protocol or session.

GETMAXPACKET and GETOPTPACKET: The packet size returned by the lower protocol/session is diminished by 32 bytes and passed on.

IP_PSEUDOHDR: This control operation turns on the IP pseudoheader length-fixup flag, either for a session or the entire protocol. The control operation is also passed to the lower session or protocol. See IP (page 156) for an explanation of this kludge.

The PREFIX control operations apply only to sessions, not the protocol.

HASH_SETLOCALPREFIX sets the implicit prefix used in computing the checksum for each outgoing packet. The prefix may be reset as desired, or turned off by setting a prefix of length 0.

HASH_SETREMOTEPREFIX sets the implicit prefix for incoming packets.

HASH_SETPREFIX sets the implicit prefix for both directions.

CONFIGURATION

HAVAL expects to be configured on top of a transport protocol that preserves packet boundaries (i.e. HAVAL will not work on top of TCP).

Example of a graph.comp file:

```
--------------------------------
@;
name=simeth/0;
```

```
name=eth protocols=simeth/0;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet;
name=haval protocols=ip;
name=udp protocols=haval;
name=udptest protocols=udp;
@;
prottbl = ../../../etc/prottbl.nonstd;
---------------------------------
```

OPTIONS

The ROM option *numparts* determines how many participants will be used for lookups to the key manager for the secret initialization vector. The default is to use one participant for each direction (incoming and outgoing). A more secure combination uses two participants for a pairwise unique key. In either case, only the first stack element of the participant(s) is used. Example:

```
name=haval  numparts 2
```

The ROM option *trailers* attaches the hash value to the end of the message instead of the beginning. Example:

```
name=haval  trailers
```

The ROM option *secretappend* appends the secret initialization vector at the beginning and the end of the message. Example:

```
name=haval  secretappend
```

The ROM option *useipsec* retrieves the secret from a key manager using the SPI passed as 3rd participant.

AUTHOR

Oliver Spatscheck

NAME

ICMP (Internet Control Message Protocol)

SPECIFICATION

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

SYNOPSIS

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

REALM

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

PARTICIPANTS

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

CONTROL OPERATIONS

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

> **Input:** `int`
>
> **Output:** none

CONFIGURATION

`name=icmp protocols=ip;`

AUTHOR

Clinton Jeffery

NAME

IP (Internet Protocol)

SPECIFICATION

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

SYNOPSIS

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

REALM

IP is in the ASYNC realm.

PARTICIPANTS

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

CONTROL OPERATIONS

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

Input: none

Output: `IPhost`

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

Input: `IPhost[2]`

Output: none

`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.

The network mask or subnetwork mask can be set through the rom file. The format is:

```
netmask   netAddress   mask
```

For example,

```
netmask   192.12.69.0  255.255.255.196
```

COMPILATION OPTIONS

There are three compilation options that have been developed for experimental extensions to IP for security work.

- Arbitrary source addresses for outgoing packets. The check for a valid source address in the local participant for an OPEN call can be disabled with the ALLOW_FAKES compile flag.
- Delivery of packets to arbitrary destinations. The DELIVER_ALL flag allows all packets to be delivered to higher protocols, even if those packets are not addressed to the local host. This can be useful for routers and firewall configurations.
- When the security extensions for IP as described in RFC1825-1827 are configured into the graph (protocol rfc1825), then IP should be compiled with the IPSEC flag in order to enable the non-standard interface between the two protocols.

AUTHORS

Clinton Jeffery, David Kays and Ed Menze

NAME

IPSEC (A Simple IP Security Layer)

SPECIFICATION

IPSEC is a simple version of the emerging Internet IP security protocol. It uses a one-byte SAID which is an algorithm identifier. The algorithm-SAID correspondence is determined by the protocol graph, as described below. In conjunction with a full suite of supprting cryptographic protocols and key management, it can provides host-host security services such as privacy, authentication, and message integrity.

REALM

IPSEC is in the ASYNC realm.

PARTICIPANTS

IPSEC passes participants to the lower protocols without manipulating them.

CONTROL OPERATIONS

CONFIGURATION

IPSEC must be configured over a datagram protocol that preserves packet boundaries. However, the complete environment for constructing a sensible security service based on IPSEC is considerably more complicated. The cryptographic services for the system should be configured over IPSEC, and they should use active key management.

IPSEC can manage several cryptographic handling stacks, as long as they adhere to a special naming convention. The numeric characters of the security service instance name is used as an algorithm identifier. Machines usig IPSEC for communication must have the same naming conventions to assure proper decryption or message validation. The graph.comp shown belown uses "1" to mean the DES algorithm, "2" to mean the md5 algorithm. Also, the scheme should below negotiates only one key for each host pair. A later version of IPSEC will support an arbitrary number.

Example of a graph.comp file:

```
---------------------------------
@;
name=simeth;
name=eth protocols=simeth;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet dir=ip files=ip,ip_frag,ip_input,ip_gc,
     ip_util,ip_rom,ip_control,ip_hdr,iproute;
name=ipsec       protocols=ip dir=ipsec files=ipsec_simple;
name=km/sig     dir=km files=km;
name=km/dh;
name=dhkx       dir=dhkx files=dhkx protocols=dss,km/dh;
name=sha        dir=sha  files=sha;
name=dss        dir=dss  files=dss,dssaux protocols=ip,sha,km/sig
```

```
     trace=TR_FULL_TRACE;
name=km/said;
#
name=km/md5 protocols=dhkx,km/dh trace=TR_FULL_TRACE;
name=md5/m2 protocols=ipsec,km/md5 dir=md5 files=md5 trace=TR_FULL_TRACE;
name=crypt/c1 protocols=ipsec,km/cr dir=crypt files=crypt trace=TR_FULL_TRACE;
name=udp protocols=md5/m2 trace=TR_DETAILED;
#
name=udptest protocols=udp;
name=confounder dir=confounder trace=TR_DETAILED;
@;
prottbl=/cs/xkernel.v3.2/etc/prottbl.std;
romopt crypt/c1 numparts 2;
romopt md5/m2 numparts 2;
romopt dhkx numparts 2;
romopt dhkx delay 30;
romopt confounder zerofill 4;
@;
prottbl = ../../../etc/prottbl.nonstd;
---------------------------------
```

AUTHOR

Hilarie Orman

NAME

IPSECSPI (A Simple IP Security Layer)

DISTRIBUTION NOTE

This protocol will not be distributed via anonymous ftp. If interested, contact xkernel-bugs@cs.arizona.edu about obtaining this protocol.

SPECIFICATION

IPSECSPI is a simple version of the emerging Internet IP security protocol. It uses a SPI to identify a security algorithm. The local SPI, protocol number and local IP address are used to demultiplex incoming packages to the corresponding stack of security algorithms. The local SPI is also used to retrieve the remote SPI from the KM which is configured below IPSECSPI. The remote SPI and the higher level protocols protocol ID is added to the header of every outgoing package.

On a xOpen call the third participant has to be the local SPI. On a xOpenEnable or xOpenDisable call the third participant has to be the algorithm id which is added or removed to the passive map.

REALM

IPSECSPI is in the ASYNC realm.

PARTICIPANTS

IPSECSPI passes the first two participants to the lower protocols without manipulating them. Every additional participant is removed.

CONTROL OPERATIONS

IPSECSPI supports the following control operations:

`SAIDMGR_GETLSAID`: Retrieves the local SPI of a given session (session only).

> **Input:** none
>
> **Return Value:** local SPI of the session.

`IPSEC_GETPEERHOST`: Returns the same as GETPEERHOST (session only).

`IPSEC_GETMYHOST`: Returns the same as GETMYHOST (session only).

IPSECSPI also provides the control options GETMAXPACKET, GETOPTPACKET, GETPARTICIPANTS, GETPEERHOST, GETMYHOST, IP_GETPSEUDOHDR and IP_PSEUDOHDR which are described elsewhere.

CONFIGURATION

IPSECSPI must be configured over a datagram protocol that preserves packet boundaries. However, the complete environment for constructing a sensible security service based on IPSECSPI is considerably more complicated. The cryptographic services for the system should be configured over IPSECSPI.

IPSECSPI has to be configured above a datagram protocol and a key manager (in that order).

Example of a graph.comp file:

```
--------------------------------
@;

name=simeth/0;
name=eth protocols=simeth/0;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet ;
name=km/a;
name=km/b;
name=km/haval;
name=km/md5;
name=km/crypt;
name=km/md5crypt_md5;
name=km/md5crypt_crypt;

name=dheckxspi protocols=ip,km/b;
name=saidmgr protocols=dheckxspi,
km/a,km/b,km/haval,km/md5,km/crypt,km/md5crypt_md5,km/md5crypt_crypt;

name=ipsecspi  protocols=ip,km/a ;

name=haval  protocols=ipsecspi,km/haval;
name=md5  protocols=ipsecspi,km/md5;
name=crypt protocols=ipsecspi,km/crypt;

name=md5/md5crypt protocols=ipsecspi,km/md5crypt_md5;
name=crypt/md5crypt  protocols=md5/md5crypt,km/md5crypt_crypt;

name=vsecselspi protocols=haval,md5,crypt,crypt/md5crypt,km/a,ip;

name=tcp protocols=vsecselspi ;

name=rl protocols=tcp,saidmgr ;

@;
prottbl=./prottbl.nonstd;
romopt  tcp useipsec;
```

```
romopt  haval useipsec;
romopt  md5 useipsec;
romopt  crypt useipsec;
romopt  md5/md5crypt useipsec;
romopt  crypt/md5crypt useipsec;
romopt  km/a kmparams 4 0-100 100 ;
romopt km/b kmparams 4 0-100 100 ;
romopt km/haval kmparams 4 16-16 100 ;
romopt km/md5 kmparams 4 16-16 100 ;
romopt km/crypt kmparams 4 8-8 100 ;
romopt km/md5crypt_md5 kmparams 4 16-16 100 ;
romopt km/md5crypt_crypt kmparams 4 8-8 100 ;
romopt dheckxspi delay 60;
romopt  saidmgr allowalgorithms haval md5 crypt md5crypt;
---------------------------------
```

AUTHOR

Oliver Spatscheck, Hilarie Orman

NAME

JOIN

SPECIFICATION

SYNOPSIS

JOIN merges up to N outgoing messages pushed to N different sessions into a single outgoing message. When this message arrives it its broken up into its original configuration and demuxed to N potentially different protocols. JOIN is bidirectional and will perform the same action on the reply. N can be any number between 1 and 16. The default value for N is four and other values may be set using ROM file options.

The number of xOpen's and xOpenenable's that should be performed on JOIN and the number of xOpenDone's JOIN will perform on higher level protocols should be equal to N + 1. The first open will return, and the first open done will produce, a control session which is not used to transmit data. To transmit a message, the user of JOIN should push a single message to one or more of the remaining N sessions and then invoke the JOINDONE control operation on the control session. When JOINDONE is invoked on JOIN, the protocol gathers all of the packets pushed, joins them into a single packet, and transmits it.

When the packet arrives at the receiver it is broken up into its original configuration and demuxed to the appropriate higher level protocols. If no packet has been pushed to a specific JOIN session, no packet will be demuxed at the corresponding join session at the receiver. When this is finished JOIN will perform an upward JOINDONE control operation on the session or protocol passed to JOIN by the receiving protocol using the JOINSETCONTROL operation. This allows the user to detect if any of the incoming message fragments have been lost or never sent. NOTE: some protocol must know that it is using JOIN and must perform JOINSETCONTROL immediately after the first openDone performed by JOIN on the SESSION passed up by the openDone.

The order in which JOIN will demux the messages to the higher level protocol(s) can be set in the ROM file.

REALM

JOIN is in the ASYNC realm.

PARTICIPANTS

JOIN takes arbitrary participants. The only restriction is that JOIN must be opened N+1 times with each open having exactly the same participant list. The best way to achieve these duplicate opens is for the higher level protocol to perform one open which eventually trickles down to JOIN and then do a get participants on the resulting session. This participant list should be used to perform the remaining opens. JOIN will create one and only one set of JOIN sessions for each unique address. Attempts to open another set of JOIN sessions with the same address will fail. Note there is a potential race condition if the higher level protocol which knows about join is in a different scheduling domain from JOIN.

CONTROL OPERATIONS

JOINGETNUMSEG: Returns N.

JOINGETORDER: Returns an array of 16 integers which gives the order in which JOIN will demux the messages fragments. While all 16 ints are returned, only the first N of them should be considered valid. If N=4, then “0 1 2 3 4” is a valid order and “4 3 2 1” is also a valid order.

JOINDONE: When invoked on the JOIN control session it causes JOIN to collect all fragments currently available and send a composite message. JOIN invokes JOINDONE on the higher level protocol or session passed to it by JOINSETCONTROL when it is finished demultiplexing message fragments. JOINDONE takes no arguments.

JOINSETCONTROL: Takes a protocol or session as argument. Some higher level protocol must invoke this operation on the JOIN control session before any packets are delivered. The protocol implementor must implement a JOINDONE control operation in the control procedure.

CONFIGURATION

The composability of JOIN is rather complex and not particularly well understood. There must be some higher level protocol that knows it is being composed over JOIN. This protocol will be configured over N+1 protocols. The first protocol must be JOIN. Between this protocol JOIN any number of other protocols may be composed on each of the remaining N paths. For example the following is a valid join protocol graph:

```
name=auth protocols= join, join,join,crypt/one,join,crypt/two;
name=crypt/one protocols=join;
name=crypt/two protocols=join;
name=join ....
```

Note that the participant lists that propagate to JOIN through each of these paths must be identical. Therefore if one path to JOIN modifies the participant list, all other paths must also perform identical modifications. For situations where the incoming and outgoing paths differ, it may be possible to use standard asymmetric protocol graph techniques to allow the incoming and outgoing message to pass through different protocol graphs. Note that any protocol composed between AUTH and JOIN must xPush no more than one message down for each xPush done to it. Protocols which fragment messages should not be composed between AUTH and JOIN.

ROM File

The ROM file is used to set N and the order in which the message parts should be demuxed to higher level protocols. The following is an example ROM file:

```
join segments 4
join order 3 2 1 0
```

After the user has set the number of segments to be joined, he should set the order at which he would like the message segments to be demuxed. In this case JOIN will demux the messages in reverse order. Note the default N is 4 and the default order is (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15). However the user should always set N and the order.

AUTHOR

Sean W. O’Malley

NAME

KM (An Encryption Key Manager)

DISTRIBUTION NOTE

This protocol will not be distributed via anonymous ftp. If interested, contact xkernel-bugs@cs.arizona.edu about obtaining this protocol.

SPECIFICATION

KM provides a key that matches a participant list.

SYNOPSIS

KM maps addresses to keys. Both keys and addresses are uninterpreted byte strings to KM. KM reads a key file at start up. The first line of the key file contains the length of the addresses in bytes, the length of the keys in bytes and the size to pass when creating the key map. The remaining lines of the key file contain address key pairs.

When a protocol such as CRYPT needs to look up a key, it opens a session to KM. The entire participant list is converted into a byte string and used to lookup a session in the key map. Note if you wish to associate a key with a single participant in a participant list you must create a new participant list with that participant as its only member. (Not an easy thing to do with the current participant structure.) The control operation KM_RESOLVE is used to lookup the actual keys. The control operation KM_SET is used to change the value of a key. The control operation KM_INVALIDATE is used to invalidate a key. The state of the KM session can be determined by calling the control operations KM_ISEMPTY, KM_ISVALID, KM_ISINVALID. If the state is empty no key value has ever been set for this address. If the state is valid there is a valid key for this address if the state is invalid an invalidate operation has been performed on this session. If a higher level protocol is no longer interested in a specific key it should perform a close operation on the KM session. No messages are pushed or popped to KM.

The keys file name is formed by concatenating the string “keys” with the instance name of the protocol (which may be null).

REALM

KM is in the ASYNC realm.

PARTICIPANTS

KM accepts participants in order to do a table lookup to find the key. The entire participant list is converted into a byte string which is then used to lookup the key.

CONTROL OPERATIONS

KM_RESOLVE returns a copy of the key stored in that session. This key is the key which is associated with the participant list passed at open time. KM_RESOLVE can fail if no key has been set or that key has been invalidated.

KM_SET sets the key for that session to the value passed.

KM_INVALIDATE marks the key as invalid and removes the KM session from the map. The session is not destroyed until all users of that session have closed it.

KM_ISEMPTY succeeds if no key for this session has been defined.

KM_ISVALID succeeds if there is a valid key for this session.

KM_ISINVALID succeeds if a KM_INVALIDATE operation has been performed in this session.

CONFIGURATION

KM is configured with cryptographic protocols such as CRYPT. It can be configured as a terminal node of the graph, or with two lower protocols. The first should be a transport protocol capable of key retrieval (such as DHKX A), and the second should be the key manager used by that protocol.

OPTIONS

There are three options: *kmparams*, *kmkeyfile*, and *kmkey*. The first can be used to specify the same information as the first line of a keys file (see below); if this option is used without a name following, no keys file will be used by the protocol. The second option is the name of the keys file. The third option can specify key information to be merged with the information in the keys file (if any).

CRYPTOGRAPHIC SHORTCOMINGS

The operating system file protections must be used to protect the keys file from unauthorized snooping.

EXAMPLE KEY FILES

Here are four examples of key files.

The first example shows a key file which maps 4 byte IP addresses to 8 byte DES keys. The 57 is the number passed to mapCreate as the size of the key map. The IP address is followed by a DES key, of 16 hex digits. Hex letters may be in upper or lower case. A trailing comment is optional, but it helps to keep track of the machine name. This keying method is appropriate if the CRYPT module is configured above IP. Note that for historical reasons the id field begins with 0x while the key field does not.

```
---------------------------------
4 8 57
0xc00c4582 0807060504030201
0xc00c4587 080706050403d213
0xc00c456e 0103040506070802
0xc00c4589 abbadabbad0add15   Berlioz  192.12.69.137
0xc00c4518 141421356abcdefa   Britten  192.12.69.24
0xc00c4554 2718281828459045   Cottonwood 192.12.69.84
0xc00c4574 3141592653589793   Leibniz  192.12.69.116
0xc00c4562 98cba765fed43210   Tchaikovsky 192.12.69.98
0xc00c4536 010304050ad53812
0xc00c4563 7766554433221100
0xc00c4561 0103040706070a02
---------------------------------
```

The second example shows keying based on Ethernet address, which is 6 bytes long. This keying is appropriate when the CRYPT module is placed above ETH.

```
---------------------------------
6 8 57
0xc00c456e0ca2 0807060504030201
0xc00c45432694 0103040506070802
---------------------------------
```

The third example shows keying based on the combination of UDP port number and IP address. The lookup data is 8 bytes long: 4 bytes of UDP port number and 4 bytes of IP address. This keying is appropriate when the CRYPT module is configured above UDP.

```
---------------------------------
8 8 57
0x000007d1c00c456e 0807060504030201
0x00000100c00c4582 0103040506070802
0x000007d1c00c4587 010304a50607080f
---------------------------------
```

The fourth example shows some additional features of the key file parser. Addresses and keys may be character strings instead of hex byte strings; the strings are demarcated with double-quotes. For long keys, the backslash character may be used to hide line breaks, to include octal-coded characters, and to quote backslash and doublequote characters. A comment may be placed after backslash-space. Following a hidden newline, initial whitespace is discarded, but another backslash turns off hiding.

Variable length keys are permitted. To use this option, the first line of the file should have a range of lengths instead of a single number.

The 0x in front of the address is optional for addresses, prohibited for keys. Individual hex bytes may be separated with one space or punctuation character. A hex byte string is terminated with tab, newline, double space, double punctuation, or the double-quote character. Single space is allowed as a terminator when the byte string is maximum permitted length, and no internal spaces have been used in the string. Single hex digits are ok, but odd length blocks of hex digits are not. In AA-B-CCC-DDDD, the B is ok but the CCC isn't.

Comment lines begin with #. Also, any text following a key is treated as a comment line, even without a #.

```
---------------------------------
6-10  0-2000 100
"Alabama"  99 11 33 66  a four byte key
"Arkansas"      "An example of a very long \
                 key, that shows some of the \
                 uses of backslash, such as \
                 for breaking long lines, for \
                 including \000 oddball characters, \
                 for quoting \\ and \", \   freebie comment!
                 \  and for resuming scanning \
                 after a hidden line, \
```

```
                 and even hiding multiple blank lines!"
"Alaska"  ""  null key is not a good idea, but permitted
001122334455  "an address or key may be expressed either \
               as a character string, or as hex bytes" if you want
0X C0 C 45 74 fF Ff   3141-59:2653;58,97  punctuation run riot
#and finally, an ordinary comment
---------------------------------
```

Limited support for additional address styles is available. The following are only useful with 32-bit host addresses and can be specified only through the input file (i.e., not *KMSET*). These styles impinge on the generality of the quoted string facility.
CAVEAT: these might need byte-swapping on some architectures. If so, this should be considered a bug in the key manager code that needs fixing.

The 4-byte IP address style is supported when enclosed in quotes:

```
4 4 16
"192.12.69.113" 000000003
```

Wildcard bytes with the 4-byte IP address style are supported. The "*" character can be used to specify a byte that is ignored in matches:

```
4 4 16
"192.12.69.*" 000000003
```

The wildcard matches are processed in the *open* and *openenable* routines after attempting a specific match. The wildcards are processed in the order they occur in the keys file.

A general 32-bit mask can be specified using "x" to specify "don't care" bits.

```
4 4 16
"m 110xxxxxx11111111111111111111111" 000000003
```

In the example above, the first two characters ("m ") indicate that a bit mask follows. The string will match any bitstring with ones and zeroes in the indicated positions. Bits in the "x" positions do not matter.

AUTHORS

David C. Schwartz, Sean W. O'Malley, and Rich Schroeppel.

NAME

MachNetIPC (a network implementation of Mach IPC)

SPECIFICATION

Mach3 Kernel Principles, with appropriate extensions.

SYNOPSIS

MachNetIPC generalizes Mach IPC to work between machines connected by networking services.

REALM

MachNetIPC is in the anchor protocol realm; it is specific to the Mach3 platform.

PARTICIPANTS

The SSR protocol communicates trivially with MachNetIPC using two elements in the participant stack: the Mach reply port for the client request, and the IP address of the remote host. To accomplish this, SSR and MachNetIPC must reside in the same Mach task.

SSR is only used to bootstrap initial Mach ports. All other addressing is done implicitly through Mach message passing that involves MachNetIPC as an intermediary.

CONTROL OPERATIONS

There are no control operations.

CONFIGURATION

MachNetIPC uses CHAN as its transport protocol, and BIDCTL for detecting host reboots. There is an unused option for two transport protocols, and that is why *udp* occurs in the configuration line:

```
name=machripc protocols=chan,udp,bidctl;
```

In addition, the *srx* and *rrx* protocols must be included in the configuration. There is a tight interface between these and MachNetIPC that is achieved via subroutine calls, not *x*-kernel operations.

CAVEATS and DISCLAIMERS

The network port identifiers are fragile for the time being. They are based on the lower 24 bits of the IP address and an 8 bit sequence number. This is inadequate and will be changed in the near future.

Behavior under heavy load or with machine reboots during complex port transfers have not thoroughly been tested.

AUTHOR

Hilarie Orman

NAME

MD5 (Message Digest 5 Algorithm)

SPECIFICATION

For outgoing messages, the MD5 protocol calculates a 16 byte checksum and pushes it onto the front of the message. For incoming messages, a 16 byte checksum is popped from the front of the message, and compared against the computed checksum of the rest of the message. No special action is taken when the compare fails; the shortened message is passed on as usual to the next higher protocol. If tracing is enabled, a message is printed when the compare fails. There is an option to use an implicit prefix at the front of each packet when computing the checksum; separate prefixes may be used for incoming and outgoing packets.

The checksum algorithm is Ron Rivest's MD5 algorithm. It is believed to have various good cryptographic qualities.

REALM

MD5 is in the ASYNC realm.

PARTICIPANTS

MD5 passes participants to the lower protocols without manipulating them.

CONTROL OPERATIONS

MD5 recognizes the following control operations; all others are passed unchanged to the lower protocol or session.

GETMAXPACKET and GETOPTPACKET: The packet size returned by the lower protocol/session is diminished by 16 bytes and passed on.

IP_PSEUDOHDR: This control operation turns on the IP pseudoheader length-fixup flag, either for a session or the entire protocol. The control operation is also passed to the lower session or protocol. See IP (page 156) for an explanation of this kludge.

The PREFIX control operations apply only to sessions, not the protocol.

HASH_SETLOCALPREFIX sets the implicit prefix used in computing the checksum for each outgoing packet. The prefix may be reset as desired, or turned off by setting a prefix of length 0.

HASH_SETREMOTEPREFIX sets the implicit prefix for incoming packets.

HASH_SETPREFIX sets the implicit prefix for both directions.

CONFIGURATION

MD5 expects to be configured on top of a transport protocol that preserves packet boundaries (i.e. MD5 will not work on top of TCP).

Example of a graph.comp file:

```
--------------------------------
@;
name=simeth/0;
```

```
name=eth protocols=simeth/0;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet;
name=md5 protocols=ip;
name=udp protocols=md5;
name=udptest protocols=udp;
@;
prottbl = ../../../etc/prottbl.nonstd;
---------------------------------
```

OPTIONS

The ROM option *numparts* determines how many participants will be used for lookups to the key manager for the secret initialization vector. The default is to use one participant for each direction (incoming and outgoing). A more secure combination uses two participants for a pairwise unique key. In either case, only the first stack element of the participant(s) is used. Example:

```
name=md5  numparts 2
```

The ROM option *leafmode* invokes Leaf Mode. Instead of the normal processing, where the hash+packet combination is passed to the next lower protocol, the updated packet is returned to the caller. There is no transport protocol; it is the caller's responsibility to handle the revised packet.

The ROM option *trailers* attaches the hash value to the end of the message instead of the beginning. Proposed standards for IP security use this format (as of early 1995). Example:

```
name=md5  trailers
```

The ROM option *secretappend* logically appends the secret initialization vector at the beginning and the end of the message for hashing. Example:

```
name=md5  secretappend
```

The ROM option *useipsec* retrieves the secret from a key manager using the SPI passed as 3rd participant.

The ROM option *rfc1828* invokes the special hashing method described in RFC1828. For the hash computation, the message is prefixed by a 64-byte block, made of the secret, a hex 80 byte, zero-padding, and a length field as defined for MD5. The secret is also appended to the message. The sandwich is hashed with MD5. This ROM option subsumes secretappend.

AUTHOR

Richard Schroeppel, Hilarie Orman

NAME

NNS (Network Name Service)

SYNOPSIS

NNS provides an entry point to MachNetIPC using the netname interface. A registry mapping service can be accessed using the MIG generated netname interface routines: netname_check_in, netname_check_out and netname_look_up. Documentation for the netname interface can be found either in /usr/mach/man or in the xkernel distribution in mach3/api/netipc/netname_*.3 manpages. NNS will not interfere with a previously executing name server. To guarantee that NNS is being used during checkins, lookups and checkouts, do a lookup on the service "NetworkNameServer" and use the associated port as the default name server port. That way requests will not mistakenly be directed to another name server instead. See mach3/user/netipc/test/nnstest.c for more details.

REALM

NNS is in the anchor protocol realm; it is specific to the Mach3 platform.

CONFIGURATION

Broadcast name lookup requests are handled using the UDP protocol. NNS depends on the BIDCTL protocol to detect reboots. Additionally the SSR protocol must be configured below NNS. See notes below for details of their interface.

```
name=nns protocols=udp, bidctl;
```

NOTES

NNS utilizes the SSR protocol for bootstrapping remote host ports into MachNetIPC. NNS and SSR communicate using a globally defined Mach port which requires that they be configured into the same Mach task.

CAVEATS and DISCLAIMERS

To specify a host in the remote lookup, use the IP address of the host and not the host name. Host name resolutions are currently not supported.

Only a single instance of NNS may run at any time.

AUTHOR

Hasnain Karampurwala

NAME

RAT (Record Accessed TCP)

SPECIFICATION

RAT uses TCP to ship very large packets to remote hosts.

SYNOPSIS

RAT is a protocol uses TCP to send very large packets to remote hosts. The current implementation of RAT manages a single TCP connection between each host pair. Messages pushed to RAT are in turn pushed to TCP. Unlike TCP RAT preserves packet boundaries. One the receiving end RAT reconstructs the original packet from an arbitrary list of buffers handed to it by TCP. RAT should be able to accept packets as large as the message tool implementation supports.

Note that RAT is totally responsible for the management of the TCP connection. In the current implementation this connection is not opened until a message is pushed to RAT. In the future more than one TCP connections between any two hosts could be supported without affecting the protocols which use RAT.

Currently a compiled constant in RAT determines the TCP send and receive buffer sizes.

RAT will almost always perform worse than BLAST on packets that are small enough for BLAST to deliver.

RAT uses any message attribute attached to the outgoing message to communicate the fact that it is reliable to some higher level protocol (in this case CHAN).

REALM

RAT is in the ASYNC realm.

PARTICIPANTS

The interface to RAT is intended to look as much like IP, VNET, and BLAST as possible. RAT assumes that the first participant passed to it at open time contains the IP address of the remote host. RAT openEnable takes participant which is ignored. OpenEables are based upon the protocol ID of the higher level protocol.

CONTROL OPERATIONS

RAT passes all control operations to IP the second protocol configured.

CONFIGURATION

RAT is not freely composable. It must be configured as follows:

name=rat protocols=tcp, ip

RAT has been designed to work under VSIZE with either BLAST, VNET, or IP as the short message protocol.

RAT recognizes the following ROM options:

`rat/xxx sourcePort N`: Instantiation *xxx* of RAT will use port N as its TCP source port. If N is equal to zero TCP will chose a port at random.

`rat/xxx destPort N`: Instantiation *xxx* of RAT will use port N as its TCP destination port. Each instance of RAT must have its own destination port.

AUTHORS

David C. Schwartz and Sean W. O'Malley

NAME

RFC1825 (IP Security Protocol)

SPECIFICATION

RFC1825 is a virtual protocol which can be used to ensure authenticity, integrity, and privacy for IP messages. The overall archictecture is defined by RFC 1825, but RFC's 1825 through 1829 together define the required functionality. The protocol is known informally as "IPSEC", and this documentation uses the informal name freely.

SYNOPSIS

This protocol co-exists with IP the SAIDMGR and is used to apply the Authentication Header and Encapsulation Security Protocol enhancements to messages. The protocol must be used with a suite of cryptographic protocols and either manual or automated key exchange. RFC's 1825-1829 explain the formats and algorithms that are required for supporting IP security.

REALM

RFC1825 is in the ASYNC realm.

PARTICIPANTS

RFC1825 is used when 3 participants are present. The third one, the SPI_PART, has a list of SPI's, and these SPI's identify the algorithms and keys that are used for enhancing an IP datagram.

The SPI participant can be set automatically by the SPIINT protocol, or it can be set by the application layer. The SAIDMGR has an interface to allow applications to lookup SPI's for specified destinations. A variant of the UDPTEST protocol that uses this interface is available locally.

The SPI participant is a list, and it is possible to use more than one SPI in that list; the message enhancements indicated by the corresponding security associations are applied in the reverse order of the SPI participant list. Thus, it is possible to use both AH and ESP for a session. There is an experimental udptest program in the `protocols/test/udptest-said` directory that illustrates this use.

The first element in the SPI participant list must be the special value -PART_TYPE_SECURE (which is -4) defined in the `ipsec-extras.h` file.

CONTROL OPERATIONS

`GETPARTIPANTS:` IP will determine the first two participants, and IPSEC will add the third.

The GETPARTICIPANTS and GETOPTPACKET operations are diverted by IP to IPSEC.

Two generic security control operations, CRYPT_ISPRIVATE and IPSEC_GETPEERHOST are supported on a per session basis.

The IP_PSEUDOHDR control operation is diverted to IPSEC so that the pseudo header length field can be modified for use by upper level protocols without making them aware of IPSEC headers, etc.

Two control operations specific to the IPSEC-to-IP interface are also supported: IPSEC_GET_MODE and IPSEC_GET_HDRLEN. The first returns the primary mode of an IPSEC session, and the second determines how many bytes will be added to a packet by IPSEC processing.

CONFIGURATION

IPSEC has a complicated configuration which involves using a key manager for the security associations and a protocol for each cryptographic algorithm supported by the local configuration. IPSEC and the SAIDMGR must coordinate their use of these algorithms and their associated protocol numbers.

In addition, configuring IPSEC involves recompiling the IP protocol with IPSEC support. This can be done in the build directory via the following steps:

- `mkdir ip`
- cp $XRT/protocols/ip/*.[ch] ip
- cp $XRT/protocols/rfc1825/ipsec-combo-ah3.h ip
- cp $XRT/protocols/rfc1825/ipsec-combo.h ip
- cp $XRT/protocols/rfc1825/ipsec-extras.h ip
- cp $XRT/protocols/rfc1825/ipsec_auth_hdr.h ip
- Modify the graph.comp file to include the following line:
  `name=ip dir=ip files=ip,ip_gc,ip_input,iproute,ip_control,`
  `ip_hdr,ip_rom,ip_frag,ip_util protocols=eth,ipsec;`
- Modify the Makefile to include the following line:
  `TMP_CFLAGS = -DIPSEC -DWRD=0`

A possible configuration line in graph.comp for RFC1825 is the following:
`name=ipsec`
`protocols=km/sauser,km/sauser,md5,haval,crypt,crypt3,tcrypt,md5,crypt3in;`
The first downward protocol is the security association lookup file; it must be the first protocol under the SAIDMGR, also.

The second downward protocol is not used. Any protocol can be used as a place holder.

The third and subsequent protocols are the cryptographic algorithms. Each algorithm will be used in "leaf mode", meaning that each will have only a key manager under it — no transport protocol is used. The order of the algorithms must match the numbering used by the SAIDMGR in setting the "algorithm" field of the security association.

1. MD5 is the first downward protocol, see pagẽpagerefMD5.
2. HAVAL, see pagẽpagerefHAVAL.
3. The DES algorithm (named "crypt" for historical reasons); see pagẽpagerefCRYPT.
4. DES3, a 3 stage DES algorithm with cipher feedback from the third to first stage, pagẽpagerefCRYPT.
5. TCRYPT

6. MD5 with DES; this is not currently supported. The sample configuration line merely repeats MD5 for this protocol.

7. DES3IN, a 3 stage DES algorithm with CBC within each stage, pagep̃agerefCRYPT.

The order of the protocols must match the numbering in the algorithms database used by the saidmgr. This database, in the third protocol under the saidmgr, maps a relative protocol number to an encoded triple of integers: the IPSEC algorithm number, the IV size in bytes, the hash size in bytes. The order of the protocols under IPSEC must match the order indicated by the IPSEC algorithm number in this database. This is the file that corresponds to the order shown above.

```
1 3 10
05    010010  # md5
10    030000  # des
11    030400  # des4t
12    030800  # des8t
14    070000  # des3in
15    070400  # des3in4t
16    070800  # des3in8t
28    040000  # des3
29    040400  # des34t
2a    040800  # des38t
```

See also the Security Association Management protocol, pagep̃agerefSAIDMGR.

The configuration of a cryptographic protocol into "leaf mode" varies by protocol; some require a rom option, some are automatically in "leaf mode" if there is no downward protocol or only one (which must be the key manager). An example, for DES, 3DES, and MD5 follows (not done yet):

For MD5, the following ROM options are necessary:

```
romopt  md5 useipsec;
romopt  md5 leafmode;
romopt  md5 rfc1828;
```

For DES (the "crypt" protocols), the following ROM option is necessary:

```
romopt  crypt useipsec;
```

It is important to note that the SAIDMGR must have access to the keys files for the cryptographic protocols used by IPSEC. This is because the SAIDMGR has the information necessary for creating the initial security associations, including any predetermined keys. The information described in the RFC1825 document describes the contents of a security association; we have split the key from the security association data structure so that the non-secret information can be accessed by local policy managers, the RFC1825 protocol, etc.

The following ROM options define the characteristics of the security association database and the cryptographic key database shared by the saidmgr and the dynamic key exchange protocol:

```
 romopt
km/sauser kmparams 8 0-100 100 ;
romopt km/b kmparams 4 0-100 100 ;
```

CAVEATS

This implmentation uses the convention that the SPI implies which protocol, AH or ESP, is to be used. Thus, one cannot use the same SPI value for two different security associations, one for AH, one for ESP. **This behavior might not be consistent with the intent or revisions of RFC1825.**

When ESP mode is used with tunneling, this protocol uses the same header for the inner and outer IP headers. When an incoming tunneled packet is processed, the source addresses for the inner and outer packets must be identical, as must the destination addresses.

DEBUGGING

- No security enhancements are applied (this might be evident from looking at the output of the "vtap" protocol, if it were configured beneath IP, for example). If no protocol in the stack opens a connection using three participants, then the RFC1825 protocol will not be used. Usually either the test program or the "spiint" protocol does this. In the latter case, there must be already be an appropriate SPI defined, either in a preset_keys file or via a key exchange protocol. If not, "spiint" will open an ordinary connection; using tracing with "spiint" will reveal this situation.

  If the preset_keys file has errors in the IP addresses, then this situation will occur. The tracing output from the "saidmgr" protocol will summarize the preset keys, and this may reveal a configuration error.

- Non-existent SPI. The tracing messages for rfc1825 may show this:
  `ipsec sa lookup:  cannot resolve SPI data structure 0x806021ab`
  . This means that the receiver of an AH or ESP message cannot find a security association for the indicated SPI. If you are using a preset_keys file with the SAIDMGR, make sure that the client and server machines have appropriately configured lines. The two lines should be "mirror images" of each other: what is local for one (first half of the line) should be remote for the other (second half of the line).

- Zero length IV. Use of a zero length IV with DES is not supported. If the preset keys file indicates an IV of zero, no error messages will be generated during the processing of the file, but the decryption will fail.

AUTHOR

Hilarie Orman

NAME

RIP (The Routing Information Protocol)

SPECIFICATION

Hedrick, [RFC 1058] Routing Information Protocol June 1988.

D. Comer and D. Stevens. Internetworking with TCP/IP Volume II 2nd Ed. 1994.

SYNOPSIS

An x-kernel implementation of the standard Internet Routing Information Protocol(RIP). RIP allows routers (hosts with more than one connected network) to periodically exchange routing information. End hosts (hosts with only one connected network) can use RIP to listen to the RIP packets exchanged by routers to keep there routing table up to date.

This RIP implementation is very standard. It differs from Comer's only in that this RIP maintains its own routing table separate from the IP routing table. Whenever a change is made to the RIP routing table RIP does a control operating on IP to change its routing table.

Warning: RIP may take several seconds to start up after rip_init is finished. All test cases that use rip should delay for at least 30 seconds before starting.

REALM

RIP has no top interface. Its bottom interface is ASYNC.

PARTICIPANTS

RIP does not support open or openenable.

CONTROL OPERATIONS

RIP does not support control operations.

CONFIGURATION

RIP can be configured only on top of UDP in a standard Internet protocol graph. RIP depends upon VNET control operations to determine how many networks the current host is connected to and to tell which network a given packet arrived on. RIP will not work without VNET. Based on the number of networks configured below VNET RIP will configure itself as an router or end-host RIP instance.

AUTHORS

Sean O'Malley (based on Comer's source)

NAME

Secure rlogin programs

SPECIFICATION

The rlogin protocol is used to initiate access to a Unix system. Under some circumstances, a password can be omitted. This version for the xkernel has some extensions that allow it to work with IPSEC protocol suite and avoid some authentication and password vulnerabilities. *This documentation is under construction*

```
- proxyrl
    - spoofeth driver for spoofing
    - sun mode allows GETMYHOST

The goal of this project was to demonstrate an rlogin environment that was
resistant to network spoofing.

TODO: change login port back to 513
TODO: client: ../xkernel -sun -script SCRIPT SCRIPT.out
TODO: server: ../xkernel -s -noipsec

TODO: proxyrl: better error message when script args not given in pairs


- vulnerabilities
    - network spoofing
        - .rhosts uses hostnames; vulnerable to DNS attacks
- no way to authenticate remote hosts
- testbed
    - rlogin server in xkernel
        - rlogin server
    - provides shell services to remote clients
    - follows rcmd protocol
- ipsec
    - rlogin client
        - proxyrl
    - connects with remote server
    - can take interactive input from keyboard or script from file
- rloginc
    - implements rcmd semantics for rlogin connection

- details
    - rlogin server
        - listens on well known port for connection requests
    - accepts {locuser, remuser, termtype} tuple from client
    - fires off /bin/login with {hostname} passed as argument
```

```
    - /bin/login calls rcmd with {locuser, remuser, hostname}
     which is correlated with information in the user's .rhosts
file
    - determines whether to prompt for passwd
- for testing, use -shell /bin/sh
- NOTE:  for testing, sit on a nonreserved port so we don't have
to have exclusive network priorities on Mach
- arguments
    -v enables vtap
    -n noopenmode: open .rhosts file in local directory, not home
     directory.
    -noipsec
allows rlogin sessions to exist without ipsec layer
allows passing of hostnames to /bin/login instead of
IP numbers.
    -shell shellname
       fork off specified shell instead of /bin/login
    -debug enables debugging messages


    - new /bin/login
     - arguments
    -n  uses .rhosts file in current directory.  I use
      this on Mach to prevent NFS filesystem access.
    -ipsec tells login that it's running with IPSEC in the
     protocol stack.

- TODO: turn ip address into hostname
- TODO: make sure ruserok() works with ip numbers

    - proxy-rlogin client
       - proxyrl handles input/output presentation
    - allows interactive or script driven rlogin sessions

- rloginc handles communications with remote server
    - interfaces with remote rlogin server

- arguments
    -spoof ip.address
     turns on IP spoofing with the given address
XXX: no argument for now...

    - auxiliary programs
     - unprivileged rlogin program
    - uses alternative port
```

```
    - changes rcmd.c to return non-privileged port in rcmd protocol

- demonstration
    - spoofing rlogin without ipsec


Sample graph.comp entries for server:

name=tcp protocols=ip;
name=rlogin protocols=tcp;

Sample graph.comp entries for client:

name=tcp protocols=ip;
name=rloginc protocols=tcp;
name=proxyrl protocols=rloginc;

The security subgraph is below the transport protocol (rlogin must use tcp).

The server side must have access to a specially modified login
program at runtime; see PLATFORM/user/login.
```

Additional notes

The pathnames in these notes are relative to the directory /cs/xkernel.v3.2/merge/build.

```
Running rlogin:

SunOS:
    SERVER
cd gkim-rl
cd server
../xkernel -s -noipsec -shell /bin/sh

on Solaris, I get
    :
    :
    :
    locuser=gkim, remuser=gkim, termtype=xterm
      RLOGIN server_shell_init: starting
    xkernel abort: warm_pool_handler: read_ether failed
    xkernel abort: warm_pool_handler: read_ether failed

*** I think the fork() is screwing up the threads pkg?
```

```
it works on SunOS and Mach fine, though...


    CLIENT
cd gkim-proxyrl
cd client
../xkernel -c192.12.69.1 -sun -v

    LOGIN
cd gkim-rl
cd newlogin

        to test, try:
    ./login -r 192.12.69.35 {-auth} {-priv} {-ipsec}
    gkim^@gkim^@xterm
        ( ^@ is ctrl-spacebar)
(locuser, remuser, termtype)
this will exercise the authentication routines
```

TODO

Better error message when prottbl doesn't have the defined protocol number

AUTHOR

Gene Kim

NAME

RRX (transfer Mach receive right).

SYNOPSIS

RRX is a receive-right transfer protocol. Transferring a receive right involves coordination among the original holder of the receive right, the new holder of the receive right, and all of the holders of send rights to the port.

EXTERNAL INTERFACE

RRX uses subroutine calls to accept transfer requests, and to keep *MachNetIPC* apprised of the movement of receive rights.

CONFIGURATION

RRX depends on a synchronous lower protocol and BIDCTL to detect reboots.

```
name=rrx protocols=chan, bidctl;
```

AUTHOR

Ed Menze

NAME

RSA (Rivest-Shamir-Adleman Encryption Algorithm)

DISTRIBUTION NOTE

This protocol will not be distributed via anonymous ftp. If interested, contact xkernel-bugs@cs.arizona.edu about obtaining this protocol.

SPECIFICATION

The RSA protocol encrypts all bytes of all packets sent using the RSA encryption algorithm in ECB (electronic code book) mode. RSA uses the key manager protocol (KM, page 165) to map from the destination address passed in at open time to an RSA key. Each message is encrypted independently. RSA is designed to be composed over any datagram protocol, and it accepts arbitrary address types at open time. RSA lengthens each packet by as much as a few hundred bytes. When packets are received, decryption reverses the encryption, and the original packet lengths are restored.

The important cryptographic feature of RSA is that the encryption and decryption keys contain different information, so that the encryption key may be published, without compromising the decryption key.

SYNOPSIS

When an RSA session is opened, it opens the protocol configured below it with the addresses passed to it during open. It then performs a GETPARTICPANTS operation on the newly opened session to acquire the fully resolved addresses of the participants. The first two participants are then used as arguments to open two key manager (KM) sessions.

When a message is pushed to an RSA session a KM_RESOLVE operation is done on the destination key manager session to get the current remote encryption key. This will be a series of two or three large numbers, expressed in decimal as a string, several hundred characters long. A special parsing routine converts the numbers to the binary values required for the RSA algorithm. The character string key, and converted binary key are cached; on subsequent messages, the key returned from the key manager is compared to the cached value, and if it hasn't changed, the conversion step is skipped.

The message length is prepended to the message as an unsigned four byte number.

The message is divided into blocks of size (binary-key-size - 1) bytes. The last block is padded with 0s if needed.

Each block is encrypted with the RSA algorithm, producing an output block one byte larger than the input block. These blocks are concatenated to make the output message. A typical RSA key size is 64 bytes: In this case, input messages of size 0 - 59 bytes will expand upon encryption to one block of 64 bytes; a message of size 1000 bytes will expand to 1024 bytes.

Decryption is straightforward, using essentially the same algorithm. (The KM_RESOLVE is done with the local host address.) Some partial checking is done that the decrypted input makes sense. Of course, received packets shrink some when they are decrypted.

RSA has not been tested on little-endian machines, or between machines of differing endianness.

RSA discards the attributes attached to any outgoing message. This is done to minimize out-of-band information transmittal. RSA does not touch any attributes attached to incoming messages, and forwards them along with the decrypted data up to the next protocol. Exception: If the IP psuedoheader length fixup flag is turned on, the length field is adjusted in upgoing pseudoheaders.

For packets going in either direction, if the KM_RESOLVE fails, the message is dropped. The present code assumes that the decimal-to-binary conversion operation on the key will succeed; bad format keys will probably cause a memory fault.

REALM

RSA is in the ASYNC realm.

PARTICIPANTS

RSA passes participants to the lower protocols without manipulating them. It uses the first two participants to lookup the keys for encryption and decryption.

CONTROL OPERATIONS

RSA recognizes the following control operations; all others are passed unchanged to the lower protocol or session. To prevent an outbound channel, the data buffer is zeroed in the downward direction.

GETMAXPACKET and GETOPTPACKET: The packet size returned by the lower protocol/session is passed on unchanged. This isn't exactly the right thing to do.

IP_PSEUDOHDR: This control operation turns on the IP pseudoheader length-fixup flag, either for a session or the entire protocol. The control operation is also passed to the lower session or protocol. See IP (page 156) for an explanation of this kludge.

ETH_REGISTER_ARP is recognized by rsaControlProtl, and is passed down without zeroing the data buffer.

RSA does not support the CRYPT series of control operations for communicating with the key manager.

CONFIGURATION

RSA expects to be configured on top of a transport protocol and a key manager. The transport protocol must preserve packet boundaries (i.e. RSA will not work on top of TCP).

Example of a graph.comp file:

```
--------------------------------
@;
name=simeth/0;
name=eth protocols=simeth/0;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet;
name=km;
name=rsa protocols=ip,km;
name=udp protocols=rsa;
name=udpcrypttest protocols=udp;
@;
```

```
prottbl = ../../../etc/prottbl.nonstd;
--------------------------------
```

KEYS

An RSA key is a set of three numbers, separated by any nonnumeric characters. The three numbers are the Modulus, the Encryption exponent, and the Decryption exponent, in decimal. The decryption exponent may be 0 if absent. A bad format key can cause a core dump in the key parser. No consistency checking is done on the numbers. No cryptographic care is taken by the key parser to erase intermediate results. The modulus M is the product of two large primes P and Q, each typically around 75 digits. P and Q don't appear explicitly in the key. They must be kept secret. There is various magic for selecting "good" primes, but random primes of the right size are usually ok. In the example below, I have standardized on the value $2^{511} + 3$ as the encryption exponent E. Any large number about the size of the modulus may be used for E, but it must be relatively prime to (P-1)*(Q-1). The modulus and the encryption exponent are published. The decryption exponent D is the solution to D*E = 1 (mod (P-1)*(Q-1)). It must be kept secret.

Example of a keys file:

```
--------------------------------
4 10-1000 10
0xc00c4574  #leibniz
            # P = 314159*10^70 + 143; Q = largest prime < 2^512/P
            # Mod = P*Q;  EncExp = 2^511+3
            "134078079299425970995740249982058461274793658205923933 7\
             77235614437217640300732179994995185511919396148903695 25\
             0588234977653711187693076711003702762655843 33
             67039039649712985497870124991029230637396829102961966 88\
             86178072186088201503677348840093714908345171384501592 90\
             9324302542687694140597328497321682450304 2051
             22107192903880453469053393557874340754687433974717032 59\
             90020571004006987352815757818719728038701171156035347 18\
             2522011239461783157977576044155444568182 5271"
0xc00c4554  #cottonwood
            # P = 271828*10^70 + 183; Q = largest prime < 2^512/P
            # Mod = P*Q;  EncExp = 2^511+3
            "134078079299425970995740249982058461274793658205923933 7\
             77235614437217640300735298002972323929919090655383808 29\
             5424768401961563886440733641129989616000 82697
             67039039649712985497870124991029230637396829102961966 88\
             86178072186088201503677348840093714908345171384501592 90\
             9324302542687694140597328497321682450304 2051
             59071699597645543529629196404023285898437578384944472 94\
             11567287797487016678130484262221184319210061087236538 029\
             5904423761567948962334399813939936575990 4515"
--------------------------------
```

For testing, I have all my machines sharing one keys file, and the decryption exponent is included with each key. In normal use, each host will have its own keys file, and the third number in each key will be 0 except for the host's own key.

RESTRICTIONS

Most of the restrictions mentioned in CRYPT also apply to RSA. One additional weakness of RSA is that it operates in ECB (electronic code book) mode, so, in a message that contains a repeated block of plaintext, the plaintext will be enciphered to the same value each time. A block of 0s enciphers to a block of 0s, and a block with all zeros except the low bit will encipher to the same thing. RSA is quite slow.

ACKNOWLEDGMENT

RSA uses the GNU multiprecision package to do its bignum arithmetic.

AUTHOR

Richard Schroeppel

NAME

SAIDMGR

SPECIFICATION

The SAIDMGR protocol is part of a suite that implements a network security architecture advocated by the IETF IPSEC working group, the RFC1825 protocols AH and ESP (see RFC1825, page 175). The documents of that working group describe the architecture, terminology, and algorithms.

The SAIDMGR manages and retrieves the data associated with a data structure known as a Security Association (SA); the identifier of the SA is its index, and it is known as the Security Parameter Index or SPI. The SAIDMGR negotiates the SA parameters, especially the shared key information, using the PHOTURIS protocol (see page 207); the key is placed into the key manager used by the security algorithms. The SAIDMGR provides the RFC1825 protocols with the information for mapping the SPI to an algorithm identifier; this is used for demultiplexing to the correct security algorithm.

Manually distributed security associations, including keys, can be supported via the preset_keys file. There is a utility program "sm" for generating this file (the source is in util/sm). See the `doc/local/sm` for more information on using this program.

SYNOPSIS

The SAIDMGR only supports xOpen, xOpenDone, xClose, xControl and xDemux calls.

On a xOpen call the higher level protocol must pass the remote address in the remote participant, the local address in the local participant, and the user identity, mode and algorithm in the third participant. The local address is required. The SAIDMGR generates a locally unique SPI adds it to the remote participant stack, and then opens the key exchange protocol (which is the first lower protocol; the only compliant protocol at this time is PHOTURIS). After this call returns the SAIDMGR retrieves three items (the key, remote SPI, algorithm) from the key manager used by PHOTURIS. The SAIDMGR stores the key in the key managers used by the security algorithm, and it stores the algorithm id and remote SPI in the key manager used by RFC1825. The SPI and IP address is used as participant to open the key manager session. The information stored in the key manager used by PHOTURIS is removed. The xOpen call returns a session object.

xOpenDone returns a unique SPI.

xClose removes all information in all KMs associated with a given session.

xDemux expects a local SPI as contents of the message and uses this SPI to distribute the information as described in the xOpen case.

Shortcoming: On a passive open no session is passed to any higher level protocol.

On initialization the SAIDMGR calls xOpenEnable on the lower level protocol for all security algorithms specified in the romoption `allowalgorithms`.

REALM

The SAIDMGR is in the ASYNC realm.

CONTROL OPERATIONS

The SAIDMGR supports the following control operations:

`SAIDMGR_GETLSAID`: Retrieves the local SPI of a given session. (session only)

Input: none

Output: none

Return Value: local SPI of the session.

`SAIDMGR_RESOLVE_NAME`: Retrieves the relative protocol number of the security algorithm given as character string. The relative protocol number is used as algorithm id. (session or protocol)

Input: Null terminated string.

Output: none

Return Value: algorithm id.

`SAIDMGR_RESCAN_PRESET_KEY_FILE`: Rescans the preset keys file. (protocol or session)

Input: none

Output: none

Return Value: none

`SAIDMGR_GET_SPIS_FOR_HOST`: Returns pair of SPIs first the local then the remote SPI. The ret value is the number of pairs returned. If there are more SPI pairs than will fit into the buffer, the buffer is filled with arbitary SPI sets for the requested host. (This is a protocol only control operation).

Input: Null terminated string.

Output: SPI set list;

Return Value: number of SPI sets in the list, -1 if no SPI set exist or the buffer is to small.

CONFIGURATION

The SAIDMGR must be configured over the following protocols, in the order given:

1. a key negotiation protocol (PHOTURIS),
2. VNET
3. a key manager which holds the security association information (see the KMA_MAPPING data structure),
4. a key manager used between the SAIDMGR and the key negotiation protocol (this holds the keys),
5. a key manager that has the algorithm attributes,
6. and the key manager protocols used by the security algorithms.

For the key managers used by the security algorithms, the instance name of the KM protocol is used to retrieve a relative protocol number which associates this KM with the corresponding security algorithm. Thus, the instance name must be a protocol name, e.g. `km/md5`. If the instance name contains a _ character (underscore), then the substring of the instance name preceding that character is used for the association, e.g. `md5crypt_md5` uses the protocol `md5crypt`. If a security algorithm is associated with more than one key manager, the key is entered into each KM associated with the security algorithm (see `md5crypt` below). The key size is determined by the key size of the KM.

The key managere storing the algorithm attributes is expeceted to have the following contents:

```
---------------------------------------------------------
05    010010  MD5 with relProtNo=1 has hash size 16 byte.
10    030000  CRYPT with relProtNo=3 has iv of 0 byte.
11    030400  CRYPT with relProtNo=3 has iv of 4 byte.
12    030800  CRYPT with relProtNo=3 has iv of 8 byte.
14    070000  CRYPT3in with relProtNo=7 has iv of 0 byte.
15    070400  CRYPT3in with relProtNo=7 has iv of 4 byte.
16    070800  CRYPT3in with relProtNo=7 has iv of 8 byte.
28    040000  CRYPT3 with relProtNo=4 has iv of 0 byte.
29    040400  CRYPT3 with relProtNo=4 has iv of 4 byte.
2a    040800  CRYPT3 with relProtNo=4 has iv of 8 byte.
---------------------------------------------------------
```

The first entrie respresents the transform number as defined in the PHOTURIS draft. The second entry contains the relative protocol number (with respect to the SAIDMGR and RFC1825) of the algorithm implementing this transform, followed by the IV length and the hash size.

Example of a graph.comp file: The following example shows a PHOTURIS, RFC1825 configuration for Mach OS machine running icmptest.

```
---------------------------------
@;

name=ethdrv/el0  ;
name=eth protocols=ethdrv/el0 ;
name=arp protocols=eth ;
name=vnet protocols=eth,arp ;

name=ip protocols=vnet,ipsec;

name=ipsec protocols=km/sauser,km/sauser,md5,md5,crypt,md5,tcrypt ;

name=spiint  protocols=ip,saidmgr ;

name=icmp protocols=spiint ;

name=km/sauser;
name=km/b;
name=km/md5;
name=km/alg;
name=km/crypt;
name=km/tcrypt;
```

```
name=km/md5crypt_md5;
name=km/md5crypt_crypt;
name=md5  protocols=km/md5 ;
name=tcrypt  protocols=km/tcrypt ;
name=crypt   protocols=km/crypt;
name=dheckxleaf/g1;
name=dhkxleaf/g2;
name=crypt/a18 protocols= km/photuris_crypt;
name=crypt/a17 protocols= km/photuris_crypt ;
name=md5/a5;
name=km/photuris_crypt;
name=km/id;

name=saidmgr  protocols=photuris,vnet,
        km/sauser,km/b,km/alg,km/md5,km/crypt,km/md5crypt_md5,km/md5crypt_crypt;

name=cookie_photuris protocols=udp;

name=photuris dir=photuris
        protocols=cookie_photuris,km/b,km/id,dheckxleaf/g1,dhkxleaf/g2,crypt/a18
,crypt/a17,md5/a5;

name=udp protocols=ip ;

name=icmptest dir=icmptest protocols=icmp ;
@;
prottbl=./prottbl.std;
prottbl=./prottbl.crypto;
romopt  udp useipsec;
romopt  tcp useipsec;
romopt  haval useipsec;
romopt  md5 useipsec;
romopt  md5 leafmode;
romopt  md5 rfc1828;
romopt  crypt useipsec;
romopt  tcrypt useipsec;
romopt  md5/md5crypt useipsec;
romopt  crypt/md5crypt useipsec;
romopt  trypt/md5crypt useipsec;
romopt  km/sauser kmparams 8 0-100 100 ;
romopt  km/b kmparams 4 0-100 100 ;
romopt  km/haval kmparams 4 16-16 100 ;
romopt  km/md5 kmparams 8 16 16 ;
romopt  km/crypt kmparams 8 8 10 ;
romopt  km/tcrypt kmparams 8 8 10;
```

```
romopt  km/md5crypt_md5 kmparams 4 16-16 100 ;
romopt  km/md5crypt_crypt kmparams 4 8-8 100 ;
romopt  crypt/a17 Photuris;
romopt  crypt/a17 useipsec;
romopt  crypt/a18 Photuris;
romopt  crypt/a18 useipsec;
romopt  km/photuris_crypt kmparams 8 8 100 ;
romopt  md5/a5 leafmode;
romopt  saidmgr allowalgorithms md5t des0t des4t des8t des30t des34t des38t
         des3in0t des3in4t des3in8t;

---------------------------------
```

It is also possible to establish keys at system startup time using the preset_keys file. Every line in the preset keys file has the following contents: local_spi;local_IP_address;local_transform_name;local_transform_value; local_mode_number;local_key=remote_spi;
remote_IP_address;remote_transform_name;remote_transform_value;remote_mode_number;remote_key

The following transform names are currently defined (followed by there transform number):

```
md5t          5
des0t        16
des4t        17
des8t        18
des3in0t     20
des3in4t     21
des3in8t     22
des30t       40
des34t       41
des38t       42
```

The transform names must use the transform number as defined in the Photuris draft 8 as relative protocol numbers to the SAIDMGR. This is done with the protocol tables, and the distributed protocol tables have done this.

The transform value should always be set to 0.

The mode defines the mode the transform is used in as defined in RFC1825.

The key has to be 128 bit long. If a smaller key is to be set (e.g., a DES key), zero padding on the right side is required.

Shortcommings

The lack of a configuration interface for the security policy of a host using PHOTURIS and the SAIDMGR.

A local and remote SPI are bound together into one session; the SPI's cannot be used independently.

AUTHOR

Oliver Spatscheck

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

NAME

SHA (NIST Secure Hash Algorithm)

SPECIFICATION

For outgoing messages, the SHA protocol calculates a 20 byte checksum and pushes it onto the front of the message. For incoming messages, a 20 byte checksum is popped from the front of the message, and compared against the computed checksum of the rest of the message. No special action is taken when the compare fails; the shortened message is passed on as usual to the next higher protocol. If tracing is enabled, a message is printed when the compare fails. There is an option to use an implicit prefix at the front of each packet when computing the checksum; separate prefixes may be used for incoming and outgoing packets.

The checksum algorithm is NIST's Secure Hash Algorithm, intended for use with the NIST Digital Signature Standard. It is believed to have various good cryptographic qualities.

REALM

SHA is in the ASYNC realm.

PARTICIPANTS

SHA passes participants to the lower protocols without manipulating them.

CONTROL OPERATIONS

SHA recognizes the following control operations; all others are passed unchanged to the lower protocol or session.

GETMAXPACKET and GETOPTPACKET: The packet size returned by the lower protocol/session is diminished by 20 bytes and passed on.

IP_PSEUDOHDR: This control operation turns on the IP pseudoheader length-fixup flag, either for a session or the entire protocol. The control operation is also passed to the lower session or protocol. See IP (page 156) for an explanation of this kludge.

The PREFIX control operations apply only to sessions, not the protocol.

HASH_SETLOCALPREFIX sets the implicit prefix used in computing the checksum for each outgoing packet. The prefix may be reset as desired, or turned off by setting a prefix of length 0.

HASH_SETREMOTEPREFIX sets the implicit prefix for incoming packets.

HASH_SETPREFIX sets the implicit prefix for both directions.

CONFIGURATION

SHA expects to be configured on top of a transport protocol that preserves packet boundaries (i.e. SHA will not work on top of TCP).

Example of a graph.comp file:

```
--------------------------------
@;
name=simeth/0;
```

```
name=eth protocols=simeth/0;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet;
name=sha protocols=ip;
name=udp protocols=sha;
name=udptest protocols=udp;
@;
prottbl = ../../../etc/prottbl.nonstd;
---------------------------------
```

AUTHOR

Richard Schroeppel

NAME

SIMSIMETH (Simulated Simulated Ethernet Driver Protocol)

SPECIFICATION

SIMSIMETH simulates an *x*-kernel ethernet driver by sending and receiving messages using any x-kernel protocol that accepts UDP addresses. SIMSIMETH is platform independent. SIMSIMETH interoperates with SIMETH (and itself). SIMSIMETH's primary purpose is to support the testing of protocols between x-kernel simulators such as the SunOS simulator and native mode x-kernel implementations such as the Mach 3.0 implementation.

SYNOPSIS

Each instantiation of SIMSIMETH is associates a *simulated* Ethernet address with a specific UDP address and simulates an Ethernet driver for a single interface. SIMSIMETH reads the UDP port from the ROM file and gets its IP address by performing a GETMYHOST control operation on the protocol configured below it.

The mapping between Unix UDP ports and SIMSIMETH Ethernet addresses is very simple. The six bytes of SIMSIMETH 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 SIMSIMETH instantiation. Note that this IP host must be valid and defined by the protocol graph below SIMSIMETH. Since one generally runs two copies of ARP when SIMSIMETH is configured keeping your IP addresses straight is sometimes difficult. See the CONFIGURATION section below.

When a message is sent using SIMSIMETH a map is checked to see if a lower level session exists for that destination address. If no lower level session exists one is created by performing an open on the protocol configured below SIMSIMETH (most probably) UDP. Once SIMSIMETH has opened a lower level session it is never closed. SIMSIMETH then pushes an Ethernet header on the message and pushes the message using the lower level session. When a packet arrives at SIMSIMETH the Ethernet header is removed and SIMSIMETH presents the packet received as an incoming Ethernet packet.

Note that an *x*-kernel may be configured with multiple instantiations of SIMSIMETH, each with its own UDP port, to simulate a multihomed host. SIMSIMETH can awkwardly simulate Ethernet broadcast messages. When an outgoing broadcast message is sent to SIMSIMETH, SIMSIMETH asks its corresponding ARP protocol for a dump of all hosts in its table. SIMSIMETH then sends the message to each of these hosts in a point-to-point fashion. Note that for a reasonable simulation of Ethernet broadcast, all *x*-kernels in communication should have the same ARP table (see the ARP appendix.)

The primary purpose of SIMSIMETH is to allow simulated x-kernels and native x-kernels to interoperate. This is possible because SIMSIMETH interoperates with the SIMETH protocol. Interoperability is achieved as follows. The graph.comp file for the simulated x-kernel is unchanged and is rooted at SIMETH the simulated Ethernet driver. The graph.comp for the machine running a native x-kernel must contain the same graph in the simulated version accept that SIMSIMETH is configured instead of SIMETH and SIMSIMETH is configured on top of the standard Internet protocol graph. If the following was the graph.comp on the SunOS simulator:

```
#SunOS Graph.comp
@;
```

```
name=simeth;
name=eth protocols=simeth;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet;
name=udp protocols=ip;
name=udptest protocols=udp;
@;
prottbl = ../prottbl.simsimeth;
```

The corresponding graph for the native mode x-kernel would be:

```
#Mach 3.0 Graph.comp
@;
name=ethdrv/SE0;
name=eth/lower protocols=ethdrv/SE0;
name=arp/lower protocols=eth/lower;
name=vnet/lower protocols=eth/lower,arp/lower;
name=ip/lower protocols=vnet/lower;
name=icmp/lower protocols=ip/lower;
name=udp/lower protocols=ip/lower;
name=simsimeth protocols=udp/lower;
name=eth/upper protocols=simsimeth;
name=arp/upper protocols=eth/upper;
name=vnet/upper protocols=eth/upper,arp/upper;
name=ip/upper protocols=vnet/upper;
name=udp/upper protocols=ip/upper;
name=udptest protocols=udp/upper;
@;
prottbl = ../prottbl.simsimeth;
```

The /lower protocols in the above graph.comp correspond to the Internet protocol suite implemented in the SunOS kernel. The /upper protocols correspond the protocols running on the simulated x-kernel. Note that a user defined protocol table is needed. This is done because the protocol table must define the correct (800) relative ETH protocol id for IP for this technique to work. The default protocol table defined by the x-kernel defines the IP protocol number as 3900 to avoid interference when testing new protocols. Note also that if you use SIMSIMETH please make sure that your versions of UDP, IP, ARP, VNET, and ETH are at least as new as SIMSIMETH.

There must be a similar correspondence between the ROM files on the SunOS side and the native Mach 3.0 side. For example the following is an example ROM file for the SunOS platform:

```
simeth 9875
eth mtu 1400
arp 192.12.69.1 192.12.69.67 9875
arp 192.12.69.2 192.12.69.99 1234
```

While the following is an example ROM file for the Mach 3 platform:

```
arp/lower     192.12.69.99    08:00:2b:23:6d:ec # mozart
simsimeth 1234
eth/upper mtu 1400
arp/upper 192.12.69.1 c0:0c:45:43:26:93 # translation:  192.12.69.67 9875
arp/upper 192.12.69.2 c0:0c:45:63:04:d2 # translation:  192.12.69.99 1234
```

The SunOS ROM file is identical to the standard ROM file except for the addition of a line to set the Ethernet MTU to 1400 bytes. If the default MTU of 1500 bytes were to be used SunOS IP would fragment the outgoing simulated Ethernet packets into two real Ethernet packets of 1500 bytes and 64 bytes respectively. This pattern of packets can result in a serious increase in the number of dropped packets. The MTU of the simulated Ethernet driver on the Mach platform (eth/upper) is also set to 1400 bytes to avoid the same problem. Note you should always manually set the MTU's of the simulated Ethernet drivers to the same number!

The ROM for Mach 3.0 must be changed to set the appropriate IP to Ethernet address correspondence for the real ARP (ARP/lower). While the simulated ARP (ARP/upper) must be configured with the same information as given in the SunOS ROM file. This is made more complex because ARP on Mach 3.0 platforms expects real Ethernet addresses while the ARP on the SunOS platform expects to find simulated Ethernet address. Therefore the user must manually convert a port IP address pair into an Ethernet address using the algorithm given above. For example the "Ethernet address": c0:0c:45:43:26:93 is simply the hex translation of 192.12.69.67 9875.

Note that when the client is started it should be passed the simulated IP address of the server (192.12.69.1or2) not the real IP address of the server.

Working graph.comp, ROM, and protocol table files can be found in the Template directory.

REALM

SIMSIMETH is in the ASYNC realm, supporting the Ethernet driver interface described in the ETH appendix.

PARTICIPANTS

SIMSIMETH 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 SIMSIMETH driver. This is used to simulate Ethernet broadcasts as described above. If no ARP protocol registers with a SIMSIMETH instantiation, broadcasts on that instantiation will not be possible.

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

Output: none

EXTERNAL INTERFACE

SIMSIMETH supports the Ethernet driver interface described in the ETH appendix.

CONFIGURATION

SIMSIMETH should be configured on top of any protocol that takes UDP addresses and below any protocol which supports the ETH lower level driver interface. It can be configured in either the driver section or the protocol section of graph.comp.

SIMSIMETH recognizes the following ROM options:

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

AUTHOR

Sean O'Malley

NAME

SRX (transfer Mach send right).

SYNOPSIS

SRX is a send-right transfer protocol, used to transfer a Mach send right when there are more than two parties involved (i.e., when the original holder of the send right, the new holder of the send right, and the holder of the receive right are on different hosts.)

EXTERNAL INTERFACE

SRX uses subroutine calls to accept transfer requests, and to notify *MachNetIPC* of the arrival of new rights.

CONFIGURATION

SRX depends on a synchronous lower protocol and BIDCTL to detect reboots.

```
name=srx protocols=chan, bidctl;
```

AUTHOR

Ed Menze

NAME

SPIINT (Security Association Interface Protocol)

SPECIFICATION

SPIINT is a virtual protocol which can be used in conjunction with the RFC1824 IP security protocol and the SAIDMGR protocol to use security enhancements to hosts that support them.

SYNOPSIS

The purpose of SPIINT is to add a security participant to the participants for each "open", if an appropriate security association is available.

SPIINT can sit almost anywhere in the protocol graph where datagrams are processed. Placing it over IP is ideal. It needs access to a key manager database of security association information; this is the SA database used by the SAIDMGR.

If PHOTURIS is configered below the SAIDMGR, SPIINT cannot be configered below UDP since otherwise a cycle in the protocol graph would be present.

In DEBUG mode, SPIINT will note whether or not security enhancements are in effort for each new connection.

REALM

SPIINT is in the ASYNC realm.

PARTICIPANTS

SPIINT dereferences a pointer to an IPhost from the top of the stack of the remote participant; if there are two elements on the stack it uses the second one as the pointer. If there is no local participant it uses GETMYHOST to find one. If it finds a security association between the two hosts, it will use a SPI as the third participant. If there is no appropriate SPI, it will pass the participants through unchanged.

ROM file

The ROM option *usephoturis* followed by an identity enables SPIINT to initiate a PHOTURIS key exchange if no appropriate security association is available. The resulting security association will use MD5 and AH for the local SPI. The transforms used for the remote SPI are determined by the remote machine. The identity is passed to PHOTURIS and is used to look up the signature key.

CONFIGURATION

```
name=spiint protocols=ip,saidmgr;
```

AUTHOR

Hilarie Orman, Oliver Spatscheck

NAME

SSR (a simple name server protocol for use with MachNetIPC).

SPECIFICATION

See the file "ssr.h". This contains the request identifier list and the data structure defining an SSR message. This is not MiG generated.

SYNOPSIS

SSR is used for bootstrapping ports into MachNetIPC. It can register a server port and forward requests to its peers on remote hosts. SSR listens for messages on its main port. This port accepts Mach messages containing a service identifier, a request identifier, an IP address, and a Mach reply port. The requests are sent to the remote peer, and the Mach reply port is translated into a MachNetIPC port. The request is forwarded to the server registered on the remote host (if any). There is no reply to SSR.

REALM

SSR is in the anchor protocol realm; it is specific to the Mach3 platform. SSR communicates with the NNS protocol using a globally defined Mach port; this requires that the two protocols exist in the same Mach task.

CONFIGURATION

SSR communicates with MachNetIPC as its only lower protocol.

```
name=ssr protocols=machripc;
```

AUTHOR

Hilarie Orman

NAME

SUNRPC (Sun Remote Procedure Call Protocol)

SPECIFICATION

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

SYNOPSIS

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

REALM

SUNRPC is in the RPC realm.

PARTICIPANTS

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

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

CONTROL OPERATIONS

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

> **Input:** none
>
> **Output:** `struct rpc_err`

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

> **Input:** `struct timeval`
>
> **Output:** none

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

> **Input:** none
>
> **Output:** `struct timeval`

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

Input: `struct timeval`

Output: none

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

Input: none

Output: `struct timeval`

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

Input: none

Output: `int`

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

Input: none

Output: `char *`

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

Input: `int`

output: none

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

Input: `char *`

Output: none

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

Input: none

Output: `int`

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

Input: none

Output: `char *`

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

Input: `int`

Output: none

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

Input: `char *`

Output: none

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

Input: `char *`

Output: none

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

Input: `char *`

Output: none

CONFIGURATION

SUNRPC is a realm boundary protocol which assumes its lower protocol is symmetric (in the ASYNC realm.)
`name=sunrpc protocols=udp;`

AUTHORS

Sean O'Malley and Richard Schroeppel

NAME

PHOTURIS-Key Exchange Protocol

SPECIFICATION

The PHOTURIS protocol is part of a suite that implements a network security architecture advocated by the IETF IPSEC working group: the RFC1825 protocols AH and ESP (see RFC1825, page 175). The documents of that working group describe the architecture, terminology, and algorithms.

The current PHOTURIS implementation complies with working group draft 8 (winter 1995).

It is possible to use PHOTURIS without the SAIDMGR (see SAIDMGR, page 189); however in the current state of development it is not encouraged.

On each xOpen call PHOTURIS initiates a key exchange. The SPI_PART has to contain the local SPI, the transform requested, the mode (AH or ESP) and the identity which should be used for the signature. After the key exchange is completed Photuris returns from the xOpen call. The SA is transfered to the SAIDMGR as descibed in the SAIDMGR section on page 189.

REALM

The PHOTURIS protocol is in the ASYNC realm.

CONFIGURATION

PHOTURIS expects to be configured above (in this order) COOKIE_PHOTURIS, the key manager to the SAIDMGR (the fourth downward protocol), a key manager containing identity to secret key mappings for MD5 signatures, one or more group choices with instance name gx (where x is the group number as defined in Photuris draft 8), one or more transforms with instance name ax, where x is the transform number as defined in Photuris draft 8.

The contents of the key manager containing identity to secret key mappings for MD5 signatures has the following format:

```
4-32 8-32 50
"spatsch@cs.arizona.edu"    "Key for spatsch@cs.arizona.edu"
"spatsch_server"         "Key for spatsch as server"
```

Example of a graph.comp file:

```
--------------------------------
@;
name=simeth/0;
name=eth protocols=simeth/0;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet ;

name=km/a dir=keym ;
name=km/b ;
```

```
name=km/algo_db;
name=km/md5;
name=km/crypt;
name=km/crypt3in;
name=km/crypt3;
name=km/photuris_crypt1;
name=km/photuris_crypt2;
name=km/id;

name=udp protocols=ip;

name=dheckxleaf/g1;
name=dhkxleaf/g2;
name=crypt/a18 protocols= km/photuris_crypt1 ;
name=crypt/a17 protocols= km/photuris_crypt2 ;
name=md5/a5;

name=cookie_photuris protocols=udp;

name=photuris dir=photuris  protocols=cookie_photuris,km/b,km/id,dheckxleaf/g1,dhkxleaf/g2,
crypt/a18,crypt/a17,md5/a5 ;

name=saidmgr dir=saidmgr protocols=photuris,vnet,
km/a,km/b,km/algo_db,km/md5,km/crypt,km/crypt3,km/crypt3in ;

name=udptest dir=ph_test files=udptest.c protocols=saidmgr,vnet;

@;
prottbl=./prottbl.nonstd;
romopt  crypt/a18 Photuris;
romopt  crypt/a18 useipsec;
romopt  km/a kmparams 8 0-100 100 ;
romopt km/b kmparams 4 0-100 100 ;
romopt km/md5 kmparams 8 16-16 100 ;
romopt km/crypt kmparams 8 8-8 100 ;
romopt km/crypt3 kmparams 8 16-16 100 ;
romopt km/crypt3in kmparams 8 16-16 100 ;
romopt km/photuris_crypt1 kmparams 8 8 100 ;
romopt km/photuris_crypt2 kmparams 8 8 100 ;
romopt vdrop interval 2 ;
romopt  md5/a5 leafmode;

---------------------------------
```

ROM file

The romoption *servername* followed by an identity is REQUIRED. The identity is used in the signature phase if this host has the role of a responder.

The romoption *grouptimeout* followed by an integer sets the time interval between the precalculation of new public values for the Diffie-Hellman Key exchange. An integer value of 1 is equal to 10 seconds.

The romoption *exchangetimeout* followed by an integer sets the lifetime of one key exchange. An integer value of 1 is equal to 10 seconds. REMARK: This is NOT equal to the lifetime of a security association negotiated during this exchange.

Shortcommings

The main shortcoming of PHOTURIS is the lack of a configuration interface for the security policy of a host using PHOTURIS and the SAIDMGR.

PHOTURIS has the following limitations:

- PHOTURIS only passes the first transform in the attribute choice to the SAIDMGR.
- If PHOTURIS is the responder of a key exchange it will always chose the same transforms as the initiator.
- PHOTURIS only supports groups 1 and 2 and MD5 as signature transform.
- PHOTURIS offers the transforms which are configured below it and will reject every transform which was not openenabled by the SAIDMGR. It was only tested with transforms 5,17,18.
- The change message is not implemented. Every change message will result in an error message.

  Unresolved xControl calls are not passed to the lower level protocol.

AUTHOR

Oliver Spatscheck

NAME

PMAP Sun Portmapper

SPECIFICATION

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

SYNOPSIS

The portmapper supports four of the five of the procedures defined by Sun. Indirect Call, Function 5, is not implemented. This implementation is compatible with Sun's native implementation. The *x*-kernel portmapper is treated as a separate protocol that sits on top of the *x*-kernel implementation of Sun RPC. This implementation only supports UDP as a lower level protocol. Client and server protocols running above the sunrpc protocol make remote procedures calls to the portmapper, using its well-known UDP port number 111.

Our implementation of the Sun Portmapper protocol uses some of the original SUN_RPC include files. You must have access to Sun include files to compile this protocol.

REALM

The portmapper is in the ASYNC realm.

CONFIGURATION

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

AUTHORS

Sean O'Malley and Richard Schroeppel

NAME

TCP (Transmission Control Protocol)

SPECIFICATION

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

SYNOPSIS

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

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

NAME

TCRYPT (A transparent DES Encryption Protocol)

See the description of CRYPT, page 112.

DISTRIBUTION NOTE

This protocol will not be distributed via anonymous ftp. If interested, contact xkernel-bugs@cs.arizona.edu about obtaining this protocol.

NAME

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

SPECIFICATION

The test protocol runs a “ping-pong” test of the protocol below it for various message lengths and numbers of iterations.

SYNOPSIS

Transport test protocols run in one of two roles, either as “client” or as “server.” 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
```

If you will be running several tests between the same hosts you may find it convenient to copy the test protocol to your build directory and edit the declaration of ServerAddr and ClientAddr in your local copy to name these hosts directly. This will eliminate the need for client and server flags. Note that most test protocols include other files from the protocols/test directory, so you will either have to copy those files as well or add $(XRT)/protocols/test to the TMP_INCLUDES variable in your Makefile.

If a test protocol is configured with an instance name of 'client' or 'server', it will come up in the appropriate role. This can be used to run both a client and server in the same kernel for loopback testing as in this graph.comp excerpt:

```
...
name=udp protocols=ip;
name=udptest/server protocols=udp;
name=udptest/client protocols=udp;
```

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

```
xkernel -testip -testudp
```

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

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

```
xkernel -trips=10000
```

If you use "dns" lines (see 9.7) in your rom file to map host names to IP addresses, then you can use the name in place of the IP address when starting the client, i.e.,

Rom file entry:

```
dns mars 192.12.69.54
```

Sample command lines for starting client:

```
xkernel -c mars
xkernel -cmars
```

Note that the name on the command line must be an exact match (not a substring) of the rom file entry.

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.

NAME

TWO4ONE (Changing or multiplying sessions)

SPECIFICATION

TWO4ONE is used to simulate multicasting to a small number of hosts, or to open and use a different session for outgoing messages.

SYNOPSIS

TWO4ONE can be configured to multicast to a group or to use a second session for replying to broadcasts. The default mode is the latter; a ROM option can select the former behavior.

The second session creation for broadcasts is useful for protocols that receive IPv4 broadcasts and need to use a separate path through the protocol graph for replies. For example, if broadcasts are signed with a public key, but replies are encrypted with symmetric keys.

REALM

In multicast mode ASYNC or SYNC (note that reply messages are discarded in SYNC mode, as there is no standard way to identify and aggregate the multiple replies).

In broadcast mode, TWO4ONE is a virtual protocol inheriting its type from that of its upper and lower protocols.

This is a virtual protocol that is active during open, openenable, push or call in multicast mode. The demux operation should bypass this protocol.

PARTICIPANTS

On an xOpen, TWO4ONE in its multicast mode passes the remote participant to the key manager to get a list of multicast participants.

On an xOpenDone, TWO4ONE in broadcast (or "switch") mode opens a second session for replies. This session has different participants than the incoming broadcast session. The recognition of broadcast addresses is limited to IPv4-type addresses.

CONFIGURATION

In multicast mode, the first lower protocol should be an ASYNC or SYNC transport protocol and the second protocol should be a key manager.

The key manager must be configured to map an address to a list of addresses. This list is of limited length (see the source code for TWO4ONE). The key manager's input file's first line should indicate the length of the address (in bytes), the maximum length of the "external" form of the group list (see partExternalize), and the size of the map table (usually 16). A group management protocol will usually sit above the key manager.

ROM OPTIONS

```
two4one numparts 2
```

The *overip* rom option is available to indicate that IP addresses are being used for the determination of the security service. This allows broadcast sessions to be a separate security class. This option is not yet coordinated with the *numparts* option. The default number of participants elements is 1.

```
two4one overip
```

The *multiply* rom option causes TWO4ONE to expect a key manager and to push all messages to several lower sessions, as indicated by the participants obtained from the key manager.

```
two4one multiply
```

AUTHOR

Hilarie Orman

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

NAME

VCACHE (Virtual Caching Protocol)

SPECIFICATION

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

SYNOPSIS

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

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

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

REALM

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

PARTICIPANTS

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

CONFIGURATION

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

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

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

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

AUTHOR

Ed Menze

NAME

VCHAN (Channel Virtual-Protocol)

SPECIFICATION

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

NAME

VCONF (Virtual Confounder Protocol)

DISTRIBUTION NOTE

This protocol will not be distributed via anonymous ftp. If interested, contact xkernel-bugs@cs.arizona.edu about obtaining this protocol.

SPECIFICATION

VCONF is a virtual protocol; it does not directly manipulate messages. At open, VCONF chooses a hash prefix for messages, based on the destination. The prefix is requested from a key manager (KM, page 165), and passed to a lower hashing protocol that accepts a prefix, either MD5 or SHA. VCONF doesn't create a session; the lower session is linked directly to the protocol above.

SYNOPSIS

When VCONF is opened, it opens the protocol configured below it with the addresses passed to it during open. The first participant is then used as an argument to open a key manager (KM) session. A key is looked up, and passed down via a control operation to a lower hash protocol, becoming the hash prefix for that session. The key manager session is closed. VCONF doesn't create a session, but arranges for the lower session to be linked directly to the protocol above.

When an opendone happens from a lower protocol, VCONF does a GETPARTICIPANTS to determine the destination. The key lookup, etc. happens as if the open were from above.

REALM

VCONF is in the ASYNC realm.

PARTICIPANTS

VCONF passes participants to the lower protocols without manipulating them. When VCONF is opened from above, it uses the first participant to lookup the hash prefix to pass to the lower hash protocol. When opened from below, it uses the second participant for the same purpose.

CONTROL OPERATIONS

VCONF passes all protocol control operations unchanged to the lower protocol. There are no VCONF sessions, hence no session control operations.

CONFIGURATION

VCONF expects to be configured somewhere above a hash protocol (MD5 or SHA) and directly above a key manager. It accepts arbitrary address types at open time.

Example of a graph.comp file:

```
---------------------------------
@;
name=simeth/0;
name=eth protocols=simeth/0;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet;
name=md5 protocols=ip;
name=km;
name=vconf protocols=md5,km;
name=udp protocols=vconf;
name=udpcrypttest protocols=udp;
@;
prottbl = ../../../etc/prottbl.nonstd;
---------------------------------
```

KEYS

There are no special restrictions on keys for VCONF; they may be any length. See the description for the key manager for example keys.

AUTHOR

Sean O'Malley

NAME

VDISORDER (Virtual Disorder Protocol)

SPECIFICATION

Shuffles the order of incoming packets. Used to exercise the recovery mechanisms of other protocols.

SYNOPSIS

VDISORDER sessions deliver packets in a somewhat different order than they receive them. The characteristics of the reordering are controlled by per-session parameters which can be set by control operations. VDISORDER will eventually deliver all packets it receives.

VDISORDER has no effect on outgoing packets.

REALM

VDISORDER is in the ASYNC realm.

PARTICIPANTS

VDISORDER passes participants to the lower protocols without manipulating them.

CONTROL OPERATIONS

`VDISORDER_SET_ARRIVAL_INTERVAL:` Sets the arrival interval for this session. This is the number of packets that are delivered in-order before the session goes into a disordering state. (session only)

Input: `int interval`

Output: none

`VDISORDER_GET_ARRIVAL_INTERVAL:` Returns the arrival interval for this session. (session only)

Input: none

Output: `int interval`

`VDISORDER_SET_DISORDER_INTERVAL:` Sets the disorder interval for this session. This is the number of packets that are held while a session is in a disordering state before the set is delivered in a random order to the upper protocol (and the session leaves the disordering state.) This parameter controls how far out-of-order a packet might arrive. (session only)

Input: `int interval`

Output: none

`VDISORDER_GET_DISORDER_INTERVAL:` Returns the disorder interval for this session. (session only)

Input: none

Output: `int interval`

`VDISORDER_SET_TIMEOUT_INTERVAL:` Sets the timeout interval for this session. When a session enters a disordering state, it starts a timeout event for this many milliseconds. If this event fires before the session has received sufficient packets to satisfy the disorder interval, it disorders the packets and delivers them to the upper protocol anyway. (session only)

Input: `int interval`

Output: none

`VDISORDER_GET_TIMEOUT_INTERVAL:` Returns the timeout interval for this session. (session only)

Input: none

Output: `int interval`

AUTHOR

Erich Nahum

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

NAME

VMUX (Virtual Muliplexing Protocol)

SPECIFICATION

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

SYNOPSIS

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

REALM

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

PARTICIPANTS

VMUX passes its participants to the lower protocols unmodified.

CONTROL OPERATIONS

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

CONFIGURATION

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

AUTHOR

Ed Menze

NAME

VNET (Virtual Network Protocol)

SPECIFICATION

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

SYNOPSIS

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

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

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

REALM

VNET is in the ASYNC realm.

PARTICIPANTS

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

CONTROL OPERATIONS

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

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

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

Output: VnetClassBuf

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

Input: none

Output: int

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

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

Input: IPhost

Output: none

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

Input: none

Output: VOID *

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

Input: VOID *

Output: none

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

Input: VOID *

Output: none

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

Input: IPhost

Output: none

CONFIGURATION

VNET expects its lower protocols to be configured in network/ARP pairs:
`name=vnet protocols=eth/1,arp/1,eth/2,arp/2;`

AUTHOR

Ed Menze

NAME

VSECSEL (Virtual Security Selection Protocol)

DISTRIBUTION NOTE

This protocol will not be distributed via anonymous ftp. If interested, contact xkernel-bugs@cs.arizona.edu about obtaining this protocol.

SPECIFICATION

VSECSEL sits above several lower protocols, selecting them with security enhancements appropriate to the source and destination addresses of a session. In a protocol graph, VSECSEL should sit over a a key manager instance that can associate an address component with a security service.

SYNOPSIS

VSECSEL is only active during connection establishment. When openenabled, VSECSEL openenables one of its lower protocols on behalf of the upper protocol. When opened, VSECSEL opens a lower protocol, making its choice based on interactions with the key manager. The successfully opened lower session is then returned.

The default service type is the checksum service.

REALM

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

PARTICIPANTS

On an xOpen, VSECSEL passes the first address component of the remote participant to the key manager to select the lower security service. VSECSEL passes its participants to that lower protocol unmodified. If that service was an encryption service, then the xOpen of that service completes, VSECSEL will set the key for both the local and remote participants to be the same key.

On an xOpenEnable, VSECSEL passes the first address component of the local participant to the key manager to select the lower security service. VSECSEL passes its participants to that lower protocol unmodified.

On an xOpenDone, VSECSEL checks that the first address component of the local participant is compatible with the lower protocol. For a lower protocol that is an encryption service, VSECSEL will set the keys for the local and remote participants to be the same key. VSECSEL then passes the unmodified participants list to the upper protocol in xOpenDone.

If the *selectbypart* ROM option is indicated (as described below), then the first item on the stack for the remote participant is used as the security selector index. The key manager is not invoked. The selector is of type *int*.

CONTROL OPERATIONS

VSECSEL determines the type of control op and forwards it to the appropriate lower protocol. For example, CRYPT control ops will be forwarded to the encryption service. Other control ops will be forwarded to all lower services, but the value returned will be from the first lower protocol.

CONFIGURATION

The following entry shows how to configure vsecsel over UDP. The order of the protocols is crucial and must be correlated with the key manager file.

```
name=vsecsel dir=vsecsel_km files=vsecsel_km protocols=udp,md5,crypt,km/vsec;
```

A key manager file example is shown below. It associates UDP port 2001 (decimal) with lower protocol 1, the checksum service; UDP port 2002 with lower protocol 0, the no-security service; UDP port 2003 with lower protocol 2, the encryption service; UDP port 2004 with the checksum service. By default, the encryption service must be the third lower protocol, because the control operations to set the keys for the local and remote participants depend on it.

```
4 4 32
000007d1        00000001
000007d2        00000000
000007d3        00000002
000007d4        00000001
```

ROM OPTIONS

The keys file serves to switch sessions to the desired service. If for some reason the services are not ordered under vsecsel in the usual way, the correlation between the functionality of the service and the keys file can be set via ROM options. For example, to indicate that the first lower protocol is an encryption service:

```
vsecsel private 0
```

The keywords *sig* for “signature”, *clear* for no enhancements, and *safe* for message integrity are also available. NB that the key manager must *always* be the last protocol under *vsecsel*.

NB the following option is not fully implemented; the first component of the local participant is used, and the author intends to fix this omission at a later time. The *numparts* rom option is available to indicate how many participants (usually two, local and remote) are to be used in the key lookup, i.e.,

```
vsecsel numparts 2
```

The *overip* rom option is available to indicate that IP addresses are being used for the determination of the security service. This allows broadcast sessions to be a separate security class. This option is not yet coordinated with the *numparts* option.

```
vsecsel overip
```

The *selectbypart* rom option is available to indicate that higher level protocols are security cognizant and have included a participant stack member that is the index of the lower security protocol to use for open's and openenable's. In this case, the participant is popped off the stack, and other participant stack components are not examined. NB: this option affects the operation of the entire protocol; all protocols above this one must be aware that the extra participant is needed. The participant is of type *int*.

```
vsecsel selectbypart
```

AUTHOR

Hilarie Orman

NAME

VSECSELSPI (Virtual Security Selection Protocol)

DISTRIBUTION NOTE

This protocol will not be distributed via anonymous ftp. If interested, contact xkernel-bugs@cs.arizona.edu about obtaining this protocol.

SPECIFICATION

VSECSELSPI sits above several lower protocols, selecting them based on a SPI. In a protocol graph, VSECSELSPI should sit over a key manager instance that can associate a SPI with an algorithm id.

SYNOPSIS

VSECSELSPI is only active during connection establishment. When openenabled, VSECSELSPI openenables the protocol specified in the SPI which is passed as third participant. If no third participant is present every security algorithm and IP (if configured below VSECSELSPI) is openeabled. The special SPIs VSECSEL_ANY_ALGORITHM and VSECSEL_SAID_IP passed as third participant openenable all security algorithms incl. IP if present and IP respectively.

xOpenDisable uses the same nameing schema.

When opened, VSECSELSPI opens a lower protocol, making its choice based on the algorithm id corresponding to the SPI which is passed as third participant. The algorithm id is retrieved from the key manager. If no third participant or VSECSEL_SAID_IP is passed IP is opened if present. The successfully opened lower session is then returned.

REALM

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

PARTICIPANTS

The remote address is passed as remote participant, the local address is passed as local participant and a SPI may be passed as third participant.

CONTROL OPERATIONS

Protocol control ops will be forwarded to all lower level protocols, but the value returned will be from the last lower level protocol or IP if IP is present.

CONFIGURATION

VSECSELSPI expect one KM as lower level protocol which provides VSECSELSPI with the local SPI to algorithm id mapping. All other protocols should be security protocols or IP. The mapping security protocol to algorithm id is established by retrieving the relative protocol number of the security protocol. If an instant name is present the instant name is used to retrieve the algorithm id instead of the protocol name.

Example of a graph.comp file:

```
---------------------------------
@;

name=simeth/0;
name=eth protocols=simeth/0;
name=arp protocols=eth;
name=vnet protocols=eth,arp;
name=ip protocols=vnet ;
name=km/a;
name=km/b;
name=km/haval;
name=km/md5;
name=km/crypt;
name=km/md5crypt_md5;
name=km/md5crypt_crypt;

name=dheckxspi protocols=ip,km/b;
name=saidmgr protocols=dheckxspi,
km/a,km/b,km/haval,km/md5,km/crypt,km/md5crypt_md5,km/md5crypt_crypt;

name=ipsecspi  protocols=ip,km/a ;

name=haval  protocols=ipsecspi,km/haval;
name=md5  protocols=ipsecspi,km/md5;
name=crypt protocols=ipsecspi,km/crypt;

name=md5/md5crypt protocols=ipsecspi,km/md5crypt_md5;
name=crypt/md5crypt  protocols=md5/md5crypt,km/md5crypt_crypt;

name=vsecselspi protocols=haval,md5,crypt,crypt/md5crypt,km/a,ip;

name=tcp protocols=vsecselspi ;

name=rl protocols=tcp,saidmgr ;

@;
prottbl=./prottbl.nonstd;
romopt  tcp useipsec;
romopt  haval useipsec;
romopt  md5 useipsec;
romopt  crypt useipsec;
romopt  md5/md5crypt useipsec;
```

```
romopt  crypt/md5crypt useipsec;
romopt  km/a kmparams 4 0-100 100 ;
romopt km/b kmparams 4 0-100 100 ;
romopt km/haval kmparams 4 16-16 100 ;
romopt km/md5 kmparams 4 16-16 100 ;
romopt km/crypt kmparams 4 8-8 100 ;
romopt km/md5crypt_md5 kmparams 4 16-16 100 ;
romopt km/md5crypt_crypt kmparams 4 8-8 100 ;
romopt dheckxspi delay 60;
romopt  saidmgr allowalgorithms haval md5 crypt md5crypt;
---------------------------------
```

AUTHOR

Oliver Spatscheck, Hilarie Orman

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

NAME

VTAP (Virtual Wiretap Protocol)

SPECIFICATION

VTAP simulates a wiretap by intercepting all messages as they are sent or received.

SYNOPSIS

VTAP is a virtual protocol that intercepts messages. It will print the entire message unless the tap has been disabled.

REALM

VTAP is in the ASYNC realm.

PARTICIPANTS

VTAP passes participants to the lower protocols without manipulating them.

CONTROL OPERATIONS

`VTAP_ENABLETAP:` Cause the session to print the intercepted message. (session only)

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

`VTAP_DISABLETAP:` Cause the session to simply pass along the intercepted message without printing it. (session only)

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

`VTAP_PRINTHEX:` If the tap is enabled, causes the session to print the hex representation of each byte of the message. (session only)

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

`VTAP_PRINTCHARS:` If the tap is enabled, causes the session to print the character representation of each byte of the message. (session only)

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

CONFIGURATION

VTAP expects to be configured between two ASYNC protocols.

AUTHOR

David C. Schwartz

NAME

XKSOCKET (the *x*-kernel socket interface)

SPECIFICATION

BSD Socket operations as implemented by the Mach3 Unix Single-Server.

SYNOPSIS

Sockets are an application interface to UDP and TCP networking services.

REALM

Sockets are in the anchor protocol realm. This version is specific to the Mach3 platform.

PARTICIPANTS

Sockets use IP addresses. The addresses are passed in messages generated by MiG stubs.

CONFIGURATION

```
name=xksocket protocols=udp,tcp;
```

AUTHOR

David Mosberger-Tang

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.

The device drivers are organized by *x*-kernel platform.

B.1 Multi-platform

NAME

SIMETH (Simulated Ethernet Driver Protocol (sunos 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 arbitrary peers is not possible with this driver. Communication is possible between SunOS and IRIX platforms.

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

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

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

REALM

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

PARTICIPANTS

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

CONTROL OPERATIONS

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

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

Output: none

EXTERNAL INTERFACE

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

CONFIGURATION

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

SIMETH recognizes the following ROM options:

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

AUTHORS

Larry Peterson and Norm Hutchinson (sunos platform), Erich Nahum, David Yates, and Jim Doyle (irix platform).

B.2 Mach 3

NAME

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

SPECIFICATION

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

SYNOPSIS

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

REALM

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

PARTICIPANTS

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

CONTROL OPERATIONS

`ETH_SET_PRIORITY:` Changes the priority of the packet filter to the given value. See CONFIGURATION below for an explanation of filter values.

Input: `int filterValue`

Output: none

EXTERNAL INTERFACE

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

CONFIGURATION

ETHDRV requires no lower protocol. Its instance name must be the name of the unmapped device as passed to the Mach system call `device_open`, unless the rom `dev` option is used as described below. For example, if the unmapped device is named `SE0`, the instance of ETHDRV using that device must be configured as:

```
name=ethdrv/SE0;
```

Check to make sure that `/etc/NETWORKS` is set up to use the unmapped device. E.g.,

```
192.12.69.1 on SE0
```

The device name is case-sensitive; `se0` represents the mapped device, while `SE0` represents the unmapped device. Usually the device name for a new machine can be determined by checking the network initialization commands in the /etc directory.
A device name can also be specified from the command line, e.g.,

```
xkernel -s -ethdrv el0
```

This overrides any `graph.comp` instance name, although it will not override a name specified in a rom file (see below).

ETHDRV recognizes the following ROM options:

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

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

`ethdrv/xxx dev devname`: Instantiation xxx of ETHDRV should open the device specified in the string "devname". For example,
`ethdrv/1 dev SE0;`

AUTHORS

Mats Bjorkman, Hilarie Orman, Michael Pagels, Ed Menze

NAME

NS8390 (Mach3 in-kernel ns8390 ethernet driver)

SPECIFICATION

NS8390 is a Mach in-kernel ethernet driver, presenting both a mach kernel interface and an *x*-kernel interface.

SYNOPSIS

Each instantiation of NS8390 is associated with a single ethernet device. NS8390 can be configured to either copy incoming packets to both the *x*-kernel and the standard Mach networking interface (thus both the in-kernel *x*-kernel and the Unix server will see packets) or to bypass the Mach interface and not allow the Unix server to see any packets.

If compiled with ALLOW_NON_XK_USERS defined, NS8390 will allow non-*x*-kernel users to send and receive network packets through the standard Mach interfaces.

REALM

NS8390 is in the ASYNC realm, supporting the ethernet driver interface described in the ETH appendix.

PARTICIPANTS

NS8390 supports the ethernet driver interface rather than a standard xkernel UPI interface and thus makes no use of participant stacks.

EXTERNAL INTERFACE

NS8390 supports the ethernet driver interface described in the ETH appendix.

CONFIGURATION

NS8390 requires no lower protocol. The last digit of the instance name must correspond to the unit device number for the particular instantiation. E.g.,

```
name=ns8390/el0;
```

for device 0.

AUTHOR

Hasnain Karampurwala (modifications to Mach kernel source distribution)

NAME

XKLANCE (Mach3 in-kernel lance ethernet driver)

SPECIFICATION

XKLANCE is a Mach in-kernel ethernet driver, presenting both a mach kernel interface and an *x*-kernel interface.

SYNOPSIS

Each instantiation of XKLANCE is associated with a single ethernet device. XKLANCE can be configured to either copy incoming packets to both the *x*-kernel and the standard Mach networking interface (thus both the in-kernel *x*-kernel and the Unix server will see packets) or to bypass the Mach interface and not allow the Unix server to see any packets.

If compiled with ALLOW_NON_XK_USERS defined, XKLANCE will check the value of the global variable xklanceAllUsers to determine whether to allow non-*x*-kernel users to send and receive network packets through the standard Mach interfaces.

REALM

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

PARTICIPANTS

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

EXTERNAL INTERFACE

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

CONFIGURATION

XKLANCE requires no lower protocol. The last digit of the instance name must correspond to the unit device number for the particular instantiation. E.g.,

```
name=xklance/SE0;
```

for device 0.

AUTHORS

Richard Schroeppel, Hilarie Orman (modifications to Mach kernel source distribution)

B.3 Irix

NAME

IRIXETH (Raw Ethernet Driver Protocol (irix platform))

SPECIFICATION

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

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 supports the ethernet driver interface described in the ETH appendix.

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

NAME

IRIXFDDI (Raw FDDI Driver Protocol (irix platform))

SPECIFICATION

IRIXFDDI is an 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 ASYNC realm, supporting the fddi driver interface described in the FDDI appendix.

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 supports the fddi driver interface described in the FDDI appendix.

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

NAME

SIMFDDI (Simulated FDDI Driver Protocol (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 arbitrary 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 the ARP appendix.)

REALM

SIMFDDI is in the ASYNC realm, supporting the fddi driver interface described in the FDDI appendix.

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 supports the fddi driver interface described in the FDDI appendix.

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

B.4 Scout (Alpha standalone)

NAME

LANCE (Scout in-kernel lance ethernet driver)

SYNOPSIS

LANCE is a Scout in-kernel ethernet driver. It is based on the LANCE driver for the Mach 3 kernel. Each instantiation of LANCE is associated with a single ethernet device.

REALM

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

PARTICIPANTS

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

EXTERNAL INTERFACE

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

CONFIGURATION

LANCE requires no lower protocol. The instance name should correspond to the name of the device controlled by that particular instantiation. E.g.,

```
name=lance/ln0;
```

for lance device ln0.

AUTHORS

David Mosberger-Tang, Richard Schroeppel, Hilarie Orman (modifications to Mach kernel source distribution)

C Application Programmer Interfaces

This appendix describes the installation and use of *x*-kernel application programmer interfaces. APIs in the *x*-kernel are platform-specific.

C.1 Mach 3

C.1.1 Sockets

The Mach 3 out-of-kernel platform provides a user socket library. At this time, the library can only be compiled with a DECstation 5000. The library can be used to communicate with the XSI protocol running in an out-of-kernel *x*-kernel in a separate task. The combination of the library and the XSI protocol provides Berkeley socket semantics. The mach 3 user-level build will create a socket library in the `xkernel/mach3/user/socket` directory which can be linked with any application making Berkeley socket calls. These calls will then be routed through the *x*-kernel socket implementation rather than the sockets in the BSD server.

The user-level build (see below) will also create some test programs that exercise the socket library and the XSI protocol in `xkernel/mach3/user/socket/tst`. Two versions of each test program are created, one linked against the *x*-kernel socket library and one linked against the standard Mach libraries. An *x*-kernel configured with the XSI protocol must be running for the *x*-kernel versions of the test programs to run, or a message of the form:

```
xsi_user_init:  netname_lookup():  (ipc/mig) server died
```

will be displayed. The test programs have a number of flags controlling their behavior. Invoking them with `-h` will give a list of these parameters.

C.1.2 MachNetIPC

The user-level build (see below) will create a test program, nnstest, which exercises the *x*-kernel's out-of-kernel implementation of MachNetIPC in `xkernel/mach3/user/netipc/test`.

NNS test communicates with the *x*-kernel protocols NNS which is the top layer of MachNetIPC. There must be an *x*-kernel task with MachNetIPC running on the host.

NNS test takes command line arguments. Before performing checkins, checkouts, and lookups the test program first looks up the service "NetworkNameServer" and uses the associated port as the default name server port. This is to avoid conflicts with a previously running simple name server. Following are some example runs.

To checkin a service on the local host do:

```
nnstest -i service_name & ;
```

Note that to perform remote host lookups, you have to specify the IP address of the host and not the host name. To look up a service on a remote machine with IP address 192.12.69.88 do:

```
nnstest -h192.12.69.88 -l service_name ;
```

To checkin, lookup and checkout a service on the remote machine do:

```
nnstest -h192.12.69.88 -ilo service_name ;
```

C.1.3 Building

The Mach 3 APIs can be built from a normal out-of-kernel *x*-kernel build directory. Running

```
make userDepend
```

will build the dependency files for the APIs and

```
make user
```

will build the user libraries and the test programs. Building the APIs should only be attempted *after* a successful build of the rest of the *x*-kernel system.

Index