Appendix

Appendix A

Uniform protocol interface

session = open(high_level_protocol, low_level_protocol, participants)

A high level protocol calls a low-level protocol to actively open a session with the given participant set.

open_enable(high_level_protocol, low_level_protocol, participants)

A high level protocol calls a low-level protocol to passively open(enable the opening of) a session that contains the given participant set.

demux(session, msg)

A session of some low-level protocol passes a message up to a high-level protocol.

controlprotl(protocol, opcode, buffer, length)

Perform a control operation on some protocol; used to set and get parameters.

push(session, msg)

Pass a message down to a session.

pop(session, msg)

Pass a message up a session.

close(session)

Close a session.

controlsessn(session, opcode, buffer, length)

Perform a control operation on some session; used to set and get parameters.

Appendix B

Proposed Ipv6 Address Allocation bits

Type of address	Binary prefix
Reserved	0000 0000
Unassigned	0000 0001
Reserved for NSAP allocation	0000 001
Reserved for IPX allocation	0000 010
Unassigned	0000 011
Unassigned	0000 1
Unassigned	0001
Unassigned	001
Aggregate Global Unicast Address	010
Unassigned	011
Reserved for Geographic Unicast Addresses	100
Unassigned	101
Unassigned	110
Unassigned	1110
Unassigned	1111 0
Unassigned	1111 10
Unassigned	1111 110
Unassigned	1111 1110 0
Link Local Use Addresses	1111 1110 10
Site Local Use Addresses	1111 1110 11
Multicast Addresses	1111 1111

Appendix C

ip.c program listing

```
/*
 * ip.c
 *
 * x-kernel v3.3
 *
 * Copyright (c) 1996,1993,1991,1990  Arizona Board of Regents
 *
 * $Revision: 1.6 $
 * $Date: 1996/06/19 16:30:13 $
 */

#include "xkernel.h"
#include "gc.h"
#include "ip.h"
#include "ip_i.h"
#include "arp.h"
#include "route.h"

#ifdef __STDC__

static void callRedirect( Event, VOID * );
static Sessn createLocalSessn( Protl, Protl, Protl, ActiveId *, IPhost * );
static void destroyForwardSessn( Sessn s );
static void destroyNormalSessn( Sessn s );
static void destroySessn( Sessn, Map );
static Sessn forwardSessn( Protl, ActiveId *, FwdId * );
static Sessn fwdBcastSessn( Protl, Sessn, ActiveId *, FwdId * );
static void fwdSessnInit( Sessn );
static IPhost *getHost( Part * );
static long getRelProtNum( Protl, Protl, char * );
static int  get_ident( Sessn );
/* static XkReturn ipCloseProtl( Protl ); */
static XkReturn ipCloseSessn( Sessn );
static Sessn ipCreateSessn( Protl, Protl, Protl, void (*)(), IPhost * );
static int  ipHandleRedirect( Sessn );
static Sessn ipOpen( Protl, Protl, Protl, Part * );
static XkReturn ipOpenDisable( Protl, Protl, Protl, Part * );
static XkReturn ipOpenEnable( Protl, Protl, Protl, Part * );
static XkHandle ipPush( Sessn, Msg * );
static Sessn localPassiveSessn( Protl, ActiveId *, IPhost * );
static void localSessnInit( Sessn );
static int routeChangeFilter( void *, void *, void * );
extern void scheduleIpFragCollector( PState * );

#else

static void callRedirect();
static Sessn createLocalSessn();
static void destroyForwardSessn();
static void destroyNormalSessn();
static void destroySessn();
static Sessn forwardSessn();
static Sessn fwdBcastSessn();
static void fwdSessnInit();
static IPhost *getHost();
static long getRelProtNum();
static int  get_ident();
/* static XkReturn ipCloseProtl(); */
static XkReturn ipCloseSessn();
static Sessn ipCreateSessn();
static int  ipHandleRedirect();
static Sessn ipOpen();
static XkReturn ipOpenDisable();
static XkReturn ipOpenEnable();
static XkHandle ipPush();
```

```
static Sessn localPassiveSessn();
static void localSessnInit();
static int routeChangeFilter();
extern void scheduleIpFragCollector();

#endif __STDC__

extern void ipProcessRomFile2();

int traceipp;

#ifdef XMEMTRACK

int IpTrackId = 0;

extern char *xMallocTrack(unsigned, int);
extern char *xMallocZeroTrack(unsigned, int);
extern void xFreeTrack(char *, unsigned, int);
extern int  TrackGetId(char *);

#define xMalloc(s) xMallocTrack(s, IpTrackId)
#define xMallocZero(s) xMallocZeroTrack(s, IpTrackId)

#undef X_NEW
#define X_NEW(Type) (Type *)xMallocTrack(sizeof(Type), IpTrackId)

#endif

static IPhost ipSiteGateway = {192,0,0,100} ;

#define SESSN_COLLECT_INTERVAL    20 * 1000 * 1000      /* 20 seconds */
#define IP_MAX_PROT               0xff

static long
getRelProtNum( hlp, llp, s )
    Protl     hlp, llp;
    char      *s;
{
    long      n;

    n = relProtNum(hlp, llp);
    if ( n == -1 ) {
            xTrace3(ipp, TR_SOFT_ERRORS,
                "%s: couldn't get prot num of %s relative to %s",
                s, hlp->name, llp->name);
            return -1;
    }
    if ( n < 0 || n > 0xff ) {
            xTrace4(ipp, TR_SOFT_ERRORS,
                "%s: prot num of %s relative to %s (%d) is out of range",
                s, hlp->name, llp->name, n);
            return -1;
    }
    return n;
}

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

    xTrace0(ipp, TR_MAJOR_EVENTS, "ipOpenDisableAll");
    defaultOpenDisableAll(ps->passiveMap, hlp, 0);
    defaultOpenDisableAll(ps->passiveSpecMap, hlp, 0);
    return XK_SUCCESS;
}

/*
 * ip_init: main entry point to IP
 */
void
```

```
ip_init(self)
    Protl self;
{
    PState  *ps;
    Part    part;
    IPhost  host;

#ifdef XMEMTRACK
    if (ipTrackId == 0)
      IpTrackId = TrackGetId("IP");
#endif

    xTrace0(ipp, 1, "IP init");
#ifdef IP_SIM_DELAYS
    xError("Warning: IP is simulating delayed packets");
#endif
#ifdef IP_SIM_DROPS
    xError("Warning: IP is simulating dropped packets");
#endif
    /* initialize protocol-specific state */
    ps = X_NEW(PState);
    self->state = (char *) ps;
    ps->self = self;
    ps->activeMap = mapCreate(IP_ACTIVE_MAP_SZ, sizeof(ActiveId));
    ps->fwdMap = mapCreate(IP_FORWARD_MAP_SZ, sizeof(FwdId));
    ps->passiveMap = mapCreate(IP_PASSIVE_MAP_SZ, sizeof(PassiveId));
    ps->passiveSpecMap = mapCreate(IP_PASSIVE_SPEC_MAP_SZ, sizeof(PassiveSpecId));
    ps->fragMap = mapCreate(IP_FRAG_MAP_SZ, sizeof(FragId));
    ps->netMaskMap = NULL;
    ps->ipSiteGateway = ipSiteGateway;

    ipProcessRomFile(self);

    if (!xIsProtl(xGetProtlDown(self, 0))) {
            xError("No llp configured below IP");
            return;
    }
    xTrace1(ipp, 2, "IP has %d protocols below\n", self->numdown);

#ifdef XNETSIM
    if (!bcmp(&ps->ipSiteGateway, &ipSiteGateway, sizeof(IPhost))) {
      if ( xControlProtl(xGetProtlDown(self, 0), GETMYHOST, (char *)&host,
              sizeof(host)) == (int)sizeof(host) ) {
        host.d = 100;
        ps->ipSiteGateway = host;
      }
    }
#endif

    /*
     * openenable physical network protocols
     */
    partInit(&part, 1);
    partPush(part, ANY_HOST, 0);
    if ( xOpenEnable(self, self, xGetProtlDown(self, 0), &part) == XK_FAILURE ) {
            xTrace0(ipp, TR_ERRORS, "ip_init : can't openenable net protocols");
    }
    /*
     * Determine number of interfaces used by the lower protocol --
     * knowing this will simplify some of our routing decisions
     */
    if ( xControlProtl(xGetProtlDown(self, 0), VNET_GETNUMINTERFACES,
                        (char *)&ps->numIfc, sizeof(int)) <= 0 ) {
            xError("Couldn't do GETNUMINTERFACES control op");
            ps->numIfc = 1;
    } else {
            xTrace1(ipp, TR_MAJOR_EVENTS, "llp has %d interfaces", ps->numIfc);
    }
    /*
     * initialize route table and set up default route
     */
    if ( rt_init(ps, &(ps->ipSiteGateway)) ) {
```

```
            xTrace0(ipp, TR_MAJOR_EVENTS, "IP rt_init -- no default gateway");
    }
    /*
     * set up function pointers for IP protocol object
     */
    self->open = ipOpen;
    /* self->close = ipCloseProtl; */
    self->controlprotl = ipControlProtl;
    self->opencnable = ipOpenEnable;
    self->opendisable = ipOpenDisable;
    self->demux = ipDemux;
    self->opendisableall = ipOpenDisableAll;
    schedulclpFragCollector(ps);
    initSessionCollector(ps->activeMap, SESSN_COLLECT_INTERVAL, destroyNormalSessn, "ip");
    initSessionCollector(ps->fwdMap, SESSN_COLLECT_INTERVAL, destroyForwardSessn, "ip forwarding");
    ipProcessRomFile2(self);
    xTrace0(ipp, 1, "IP init done");
}

static IPhost *
getHost( p )
    Part    *p;
{
    IPhost  *h;

    if ( !p || (partLength(p) < 1) ) {
            xTrace0(ipp, TR_SOFT_ERRORS, "ipGetHost: participant list error");
            return 0;
    }
    h = (IPhost *)partPop(p[0]);
    if ( h == 0 ) {
            xTrace0(ipp, TR_SOFT_ERRORS, "ipGetHost: empty participant stack");
    }
    return h;
}

/*
 * ipOpen
 */
static Sessn
ipOpen(self, hlp, hlpType, p)
    Protl self, hlp, hlpType;
    Part *p;
{
    Scssn   ip_s;
    IPhost  *remoteHost;
    IPhost  *localHost = 0;
    ActiveId    activeid;
    long    hlpNum;

    xTrace0(ipp, 3, "IP open");
    if ( (remotel Iost = getI Iost(p)) == 0 ) {
            return ERR_SESSN;
    }
    if ( (hlpNum = getRelProtNum(hlpType, self, "open")) == -1 ) {
            return ERR_SESSN;
    }
    if ( partLength(p) > 1 ) {
            /*
             * Local participant has been explicitly specified
             */
            localHost = (IPhost *)partPop(p[1]);
            if ( localHost == (IPhost *)ANY_HOST ) {
                localIost = 0;
            }
    }
    xTrace2(ipp, 5, "IP sends to %s, %d", ipHostStr(remoteHost), hlpNum);

    /*
     * key on hlp prot number, destination addr, and local addr (if given)
     */
    bzero((char *)&activeid, sizeof(ActiveId));
```

```
    activeid.protNum = hlpNum;
    activeid.remote = *remoteHost;
    if ( localHost ) {
            activeid.local = *localIlost;
    }
    ip_s = createLocalSessn( self, hlp, hlpType, &activeid, localHost );
    if ( ip_s != ERR_SESSN ) {
            ip_s->idle = FALSE;
    }
    xTrace1(ipp, 3, "IP open returns %x", ip_s);
    return ip_s;
}

/*
 * Create an IP session which sends to remote host key->dest.  The
 * 'rem' and 'prot' fields of 'key' will be used as passed in.
 *
 * 'localHost' specifies the host to be used in the header for
 * outgoing packets.  If localHost is null, an appropriate localHost will
 * be selected and used as the 'local' field of 'key'.  If localIlost
 * is non-null, the 'local' field of 'key' will not be modified.
 */
static Sessn
createLocalSessn( self, hlp, hlpType, key, localIlost )
    Protl       self, hlp, hlpType;
    ActiveId    *key;
    IPhost      *localHost;
{
    PState      *ps = (PState *)self->state;
    SState      *ss;
    IPheader    *iph;
    IPhost      host;
    Sessn       s;

    s = ipCreateSessn(self, hlp, hlpType, localSessnInit, &key->remote);
    if ( s == ERR_SESSN ) {
            return s;
    }
    /*
     * Determine my host address
     */
    if ( localHost ) {
            if ( ! ipIsMyAddr(self, localHost) ) {
                xTrace1(ipp, TR_SOFT_ERRORS, "%s is not a local IP host",
                            ipHostStr(localHost));
                return ERR_SESSN;
            }
    } else {
            if ( xControlScssn(xGetScssnDown(s, 0), GETMYHOST, (char *)&host,
                            sizeof(host)) < (int)sizeof(host) ) {
                xTrace0(ipp, TR_SOFT_ERRORS,
                            "IP open could not get interface info for remote host");
                destroyNormalScssn(s);
                return ERR_SESSN;
            }
            localIlost = &host;
            key->local = *localHost;
    }
    s->binding = mapBind(ps->activeMap, (char *)key, s);
    if ( s->binding == ERR_BIND ) {
            XkReturn  res;

            xTrace0(ipp, TR_MAJOR_EVENTS, "IP open -- session already existed");
            destroyNormalSessn(s);
            res = mapResolve(ps->activeMap, key, &s);
            xAssert( res == XK_SUCCESS );
            return s;
    }
    ss = (SState *)s->state;
    iph = &ss->hdr;
    iph->source = *localHost;
    /*
```

```
    * fill in session template header
    */
   iph->vers_hlen = IPVERS;
   iph->vers_hlen |= 5;              /* default hdr length */
   iph->type = 0;
   iph->time = IPDEFAULTDGTTL;
   iph->prot = key->protNum;
   xTrace1(ipp, 4, "IP open: my ip address is %s",
           ipHostStr(&iph->source));
   return s;
}

static Sessn
ipCreateSessn( self, hlp, hlpType, f, dst )
   Protl     self, hlp, hlpType;
   void      (*f)();
   IPhost    *dst;
{
   Sessn     s;
   SState    *ss;

   s = xCreateSessn(f, hlp, hlpType, self, 0, 0);
   ss = X_NEW(SState);
   s->state = (VOID *)ss;
   bzero((char *)ss, sizeof(SState));
   ss->hdr.dest = *dst;
   if ( ipHandleRedirect(s) ) {
           xTrace0(ipp, 3, "IP open fails");
           destroyNormalSessn(s);
           return ERR_SESSN;
   }
   return s;
}

static void
localSessnInit(self)
   Sessn self;
{
   self->push = ipPush;
   self->pop = ipStdPop;
   self->controlsessn = ipControlSessn;
   self->getparticipants = ipGetParticipants;
   self->close = ipCloseSessn;
}

static void
fwdSessnInit(self)
   Sessn self;
{
   self->pop = ipForwardPop;
}

/*
* ipOpenEnable
*/
static XkReturn
ipOpenEnable(self, hlp, hlpType, p)
   Protl self, hlp, hlpType;
   Part *p;
{
   PState    *pstate = (PState *)self->state;
   IPhost    *localHost;
   long      protNum;

   xTrace0(ipp, TR_MAJOR_EVENTS, "IP open enable");
   if ( (localHost = getHost(p)) == 0 ) {
           return XK_FAILURE;
   }
   if ( (protNum = getRelProtNum(hlpType, self, "ipOpenEnable")) == -1 ) {
           return XK_FAILURE;
   }
   if ( localHost == (IPhost *)ANY_HOST ) {
```

```
            xTrace1(ipp, TR_MAJOR_EVENTS, "ipOpenEnable binding protocol %d",
                    protNum);
            return defaultOpenEnable(pstate->passiveMap, hlp, hlpType,
                                     &protNum);
    } else {
            PassiveSpecId           key;

            if ( !ipIsMyAddr(self, localHost) ) {
              xTrace1(ipp, TR_MAJOR_EVENTS,
                      "ipOpenEnable -- %s is not one of my hosts",
                      ipHostStr(localHost));
              return XK_FAILURE;
            }
            key.host = *localHost;
            key.prot = protNum;
            xTrace2(ipp, TR_MAJOR_EVENTS,
                    "ipOpenEnable binding protocol %d, host %s",
                    key.prot, ipHostStr(&key.host));
            return defaultOpenEnable(pstate->passiveSpecMap, hlp, hlpType,
                                     &key);
    }
}

/*
 * ipOpenDisable
 */
static XkReturn
ipOpenDisable(self, hlp, hlpType, p)
    Prot self, hlp, hlpType;
    Part *p;
{
    PState   *pstate = (PState *)self->state;
    IPhost   *localHost;
    long     protNum;

    xTrace0(ipp, 3, "IP open disable");
    xAssert(self->state);
    xAssert(p);

    if ( (localHost = getLlost(p)) == 0 ) {
            return XK_FAILURE;
    }
    if ( (protNum = getRelProtNum(hlpType, self, "ipOpenDisable")) == -1 ) {
            return XK_FAILURE;
    }
    if ( localHost == (IPhost *)ANY_HOST ) {
            xTrace1(ipp, TR_MAJOR_EVENTS,
                    "ipOpenDisable unbinding protocol %d", protNum);
            return defaultOpenDisable(pstate->passiveMap, hlp, hlpType,
                                      &protNum);
    } else {
            PassiveSpecId           key;

            key.host = *localHost;
            key.prot = protNum;
            xTrace2(ipp, TR_MAJOR_EVENTS,
                    "ipOpenDisable unbinding protocol %d, host %s",
                    key.prot, ipHostStr(&key.host));
            return defaultOpenDisable(pstate->passiveSpecMap, hlp, hlpType,
                                      &key);
    }
}

/*
 * ipCloseScssn
 */
static XkReturn
ipCloseSessn(s)
    Scssn s;
{
  xTrace1(ipp, 3, "IP close of session %x (does nothing)", s);
  xAssert(xIsSessn(s));
```

```
    xAssert( s->rcnt == 0 );
    return XK_SUCCESS;
}

static void
destroyForwardSessn(s)
    Sessn s;
{
    PState   *ps = (PState *)(xMyProtl(s))->state;

    destroySessn(s, ps->fwdMap);
}

static void
destroySessn(s, map)
    Sessn    s;
    Map      map;
{
    int      i;
    Sessn    lls;

    xTrace1(ipp, 3, "IP DestroySessn %x", s);
    xAssert(xIsSessn(s));
    if ( s->binding && s->binding != ERR_BIND ) {
            mapRemoveBinding(map, s->binding);
    }
    for (i=0; i < s->numdown; i++ ) {
            lls = xGetSessnDown(s, i);
            if (xIsSessn(lls)) {
              xClose(lls);
            }
    }
    xDestroySessn(s);
}

static void
destroyNormalSessn(s)
    Sessn s;
{
    PState   *ps = (PState *)(xMyProtl(s))->state;

    destroySessn(s, ps->activeMap);
}


/*
 * ipCloseProtl
 */
/*
static XkReturn
ipCloseProtl(self)
    Protl self;
{
    PState      *pstate;

    xAssert(xIsProtl(self));
    xAssert(self->rcnt==1);

    pstate = (PState *) self->state;
    mapClose(pstate->activeMap);
    mapClose(pstate->passiveMap);
    mapClose(pstate->fragMap);
    xFree((char *) pstate);
    xDestroyProtl(self);
    return XK_SUCCESS;
}
*/

/*
 * ipPush
 */
static XkHandle
```

```
ipPush(s, msg)
   Sessn s;
   Msg *msg;
{
   SState   *sstate;
   IPheader          hdr;

   xAssert(xIsSessn(s));
   sstate = (SState *) s->state;

   hdr = sstate->hdr;
   hdr.ident = get_ident(s);
   hdr.dlen = msgLength(msg) + (GET_HLEN(&hdr) * 4);
   return ipSend(s, xGetSessnDown(s, 0), msg, &hdr);
}

/*
 * Send the msg over the ip session's down session, fragmenting if necessary.
 * All header fields not directly related to fragmentation should already
 * be filled in.  We only reference the 'mtu' field of s->state (this
 * could be a forwarding session with a vestigial header in s->state,
 * so we use the header passed in as a parameter.)
 */
XkHandle
ipSend(s, lls, msg, hdr)
   Sessn    s, lls;
   Msg      *msg;
   IPheader *hdr;
{
   int      hdrLen;
   int      len;
   SState   *sstate;
   void     *buf;

   sstate = (SState *)s->state;
   len = msgLength(msg);
   hdrLen = GET_HLEN(hdr);
   if ( len + hdrLen * 4 <= sstate->mtu ) {
           /*
            * No fragmentation
            */
           xTrace0(ipp,5,"IP send : message requires no fragmentation");

           buf = msgPush(msg, hdrLen*4);
           xAssert(buf);
           ipHdrStore(hdr, buf, hdrLen*4, 0);

           xIfTrace(ipp,5) {
             xTrace0(ipp,5,"IP send unfragmented datagram header: \n");
             ipDumpHdr(hdr);
           }
           return xPush(lls, msg);
   } else {
           /*
            * Fragmentation required
            */
           int          fragblks;
           int          fragsize;
           Msg          fragmsg;
           int          offset;
           int          fraglen;
           XkHandle     hmdle = XMSG_NULL_HANDLE;

           if ( hdr->frag & DONTFRAGMENT ) {
             xTrace0(ipp,5,
                     "IP send: fragmentation needed, but NOFRAG bit set");
             return XMSG_NULL_HANDLE;  /* drop it */
           }
           fragblks = (sstate->mtu - (hdrLen * 4)) / 8;
           fragsize = fragblks * 8;
           xTrace0(ipp,5,"IP send : datagram requires fragmentation");
           xIfTrace(ipp,5) {
```

```
            xTrace0(ipp,5,"IP original datagram header :");
            ipDumpHdr(hdr);
          }
          /*
           * fragmsg = msg;
           */
          xAssert(xIsSessn(lls));
          msgConstructEmpty(&fragmsg);
          for( offset = 0; lcn > 0; lcn -= fragsize, offset += fragblks) {
            IPheader              hdrToPush;

            hdrToPush = *hdr;
            fraglcn = lcn > fragsize ? fragsize : lcn;
            msgBreak(msg, &fragmsg, fraglcn);
            /*
             * eventually going to need to selectively copy options
             */
            hdrToPush.frag += offset;
            if ( fraglcn != lcn ) {
                  /*
                   * more fragments
                   */
                  hdrToPush.frag |= MOREFRAGMENTS;
            }
            hdrToPush.dlcn = hdrLcn * 4 + fraglcn;
            xIfTrace(ipp,5) {
                  xTrace0(ipp,5,"IP datagram fragment header: \n");
                  ipDumpHdr(&hdrToPush);
            }

            buf = msgPush(&fragmsg, hdrLen * 4);
            xAssert(buf);
            ipHdrStore(&hdrToPush, buf, hdrLen * 4, 0);

            if ( (handle = xPush(lls, &fragmsg)) == XMSG_ERR_HANDLE ) {
                  break;
            }
          }
          msgDestroy(&fragmsg);
          return ( handle == XMSG_ERR_HANDLE ) ? handle : XMSG_NULL_HANDLE;
    }
}

Enable *
ipFindEnable( self, hlpNum, localHost )
    ProtI self;
    Int   hlpNum;
    IPhost *localHost;
{
    PState    *ps = (PState *)self->state;
    Enable  *e = ERR_ENABLE;
    PassiveId key = hlpNum;
    PassiveSpecId specKey;

    if (mapResolve(ps->passiveMap, &key, &e) == XK_SUCCESS) {
          xTrace1(ipp, TR_MAJOR_EVENTS,"Found an enable object for prot %d", key);
    } else {
          bzero((char *)&specKey, sizeof(PassiveSpecId));
          specKey.prot = key;
          specKey.host = *localHost;
          if (mapResolve(ps->passiveSpecMap, &specKey, &e) == XK_SUCCESS) {
            xTrace2(ipp, TR_MAJOR_EVENTS,"Found an enable object for prot %d host %s",
                  specKey.prot, ipHostStr(&specKey.host));
          }
    }
    return e;
}

static Sessn
localPassiveSessn( self, actKey, localHost )
    ProtI self;
    ActiveId *actKey;
```

```
    IPhost    *localHost;
{
    Enable  *e;

    e = ipFindEnable(sclf, actKey->protNum, localHost);
    if ( e == ERR_ENABLE ) {
            return ERR_SESSN;
    }
    return createLocalSessn(sclf, e->hlp, e->hlpType, actKey, localHost);
    /*
     * openDone will get called in validateOpenEnable
     */
}

static Sessn
fwdBcastSessn( self, llsIn, actKey, fwdKey )
    Protl       sclf;
    Sessn       llsIn;
    ActiveId  *actKey;
    FwdId     *fwdKey;
{
    Sessn       s;
    Part        p;
    Sessn       lls;
    IPhost    localHost;

    xTrace0(ipp, TR_MAJOR_EVENTS, "creating forward broadcast session");
    if ( xControlSessn(llsIn, GETMYHOST, (char *)&localHost, sizeof(IPhost)) < 0 ) {
            return ERR_SESSN;
    }
    if ( (s = localPassiveSessn(self, actKey, &localHost)) == ERR_SESSN ) {
            /*
             * There must not have been an openenable for this msg type --
             * this will just be a forwarding session
             */
            if ( (s = forwardSessn(self, actKey, fwdKey)) == ERR_SESSN ) {
              return ERR_SESSN;
            }
            xSetSessnDown(s, 1, xGetSessnDown(s, 0));
            xSetSessnDown(s, 0, 0);
    } else {
            /*
             * This will be a local session with an extra down session for
             * the forwarding of broadcasts
             */
            partInit(&p, 1);
            partPush(p, &actKey->local, sizeof(IPhost));
            if ( (lls = xOpen(self, self, xGetProtlDown(self, 0), &p)) == ERR_SESSN ) {
              xTrace0(ipp, TR_ERRORS, "ipFwdBcastSessn couldn't open lls");
              return ERR_SESSN;
            }
            xSetSessnDown(s, 1, lls);
    }
    s->pop = ipFwdBcastPop;
    return s;
}

static Sessn
forwardSessn( self, actKey, fwdKey )
    Protl self;
    ActiveId *actKey;
    FwdId   *fwdKey;
{
    PState    *ps = (PState *)self->state;
    Sessn       s;

    xTrace2(ipp, TR_MAJOR_EVENTS,
            "creating forwarding session to net %s (host %s)",  ipHostStr(fwdKey), ipHostStr(&actKey->local));
    s = ipCreateSessn(self, xNullProtl, xNullProtl, fwdSessnInit, &actKey->local);
    if ( s == ERR_SESSN ) {
            return s;
    }
```

```
    s->binding = mapBind(ps->fwdMap, (char *)fwdKey, s);
    xAssert( s->binding != ERR_BIND );
    return s;
}

Sessn
ipCreatePassiveSessn( self, lls, actKey, fwdKey )
    Protl       self;
    Sessn       lls;
    ActiveId*actKey;
    FwdId   *fwdKey;
{
    PState              *ps = (PState *)self->state;
    VnetClassBuf        buf;
    Sessn               s = ERR_SESSN;

    buf.host = actKey->local;
    if ( xControlProtl(xGetProtlDown(self, 0), VNET_GETADDRCLASS,
                        (char *)&buf, sizeof(buf)) < (int)sizeof(buf) ) {
            xTrace0(ipp, TR_ERRORS,
                    "ipCreatePassiveSessn: GETADDRCLASS failed");
            return ERR_SESSN;
    }
    switch( buf.class ) {
      case LOCAL_ADDR_C:
            /*
             * Normal session
             */
            s = localPassiveSessn(self, actKey, &actKey->local);
            break;

      case REMOTE_HOST_ADDR_C:
      case REMOTE_NET_ADDR_C:
            s = forwardSessn(self, actKey, fwdKey);
            break;

      case BCAST_SUBNET_ADDR_C:
            if ( ps->numIfc > 1 ) {
              /*
               * Painfully awkward forward/local consideration session
               */
              s = fwdBcastSessn(self, lls, actKey, fwdKey);
              break;
            }
            /*
             * Else fallthrough
             */

      case BCAST_LOCAL_ADDR_C:
      case BCAST_NET_ADDR_C:
            {
              IPhost    localHost;

              /*
               * Almost a normal session -- need to be careful about our
               * source address
               */
              if ( xControlSessn(lls, GETMYHOST, (char *)&localHost, sizeof(IPhost))
                      < 0 ) {
                      return ERR_SESSN;
              }
              s = localPassiveSessn(self, actKey, &localHost);
            }
            break;

    }
    return s;
}

/*
 * ipHandleRedirect -- called when the ip session's lower session needs
 * to be (re)opened.  This could be when the ip session is first created
```

```
 * and the lower session is nonexistent, or when a redirect is received
 * for this session's remote network.  The router is
 * consulted for the best interface.  The new session is assigned to
 * the first position in the ip session's down vector.  The old session,
 * if it existed, is freed.
 *
 * Note that the local IPhost field of the header doesn't change even
 * if the route changes.
 *
 * preconditions:
 *          s->state should be allocated
 *          s->state->hdr.dest should contain the ultimate remote address or net
 *
 * return values:
 *          0 if lower session was succesfully opened and assigned
 *          1 if lower session could not be opened -- old lower session is
 *                  not affected
 */
static int
ipIlandleRedirect(s)
  Scssn s;
{
  Protl     ip = xMyProtl(s);
  Protl     llp = xGetProtlDown(ip, 0);
  Scssn     lls, llsOld;
  SState    *ss = (SState *)s->state;
  PState    *pstate=(PState *)s->myprotl->state;
  route     *rt;
  Part      p;
  int               res;

  /*
   * 'host' is the remote host to which this session sends packets,
   * not necessarily the final destination
   */
  xAssert(xIsSessn(s));
  partInit(&p, 1);
  partPush(p, &ss->hdr.dest, sizeof(IPhost));
  if ( (lls = xOpen(ip, ip, llp, &p)) == ERR_SESSN ) {
        xTrace0(ipp, TR_EVENTS, "ipIlandleRedirect could not get direct lower session");
        if ( (rt = rt_get(pstate, &ss->hdr.dest)) == 0 ) {
          xTrace0(ipp, TR_SOFT_ERRORS, "ipHandleRedirect could not find route");
          return 1;
        }
        partInit(&p, 1);
        partPush(p, &rt->gw, sizeof(IPhost));
        if ( (lls = xOpen(ip, ip, llp, &p)) == ERR_SESSN ) {
          xTrace0(ipp, TR_ERRORS, "ipIlandleRedirect could not get gateway lower session");
          return 1;
        }
  }
  xTrace0(ipp, 5, "Successfully opened lls");
  /*
   * Determine mtu for this interface
   */
  res = xControlSessn(lls, GETMAXPACKET, (char *)&ss->mtu, sizeof(int));
  if (res < 0 || ss->mtu <= 0) {
        xTrace0(ipp, 3, "Could not determine interface mtu");
        ss->mtu = IPOPTPACKET;
  }
  if ( s->numdown && xIsScssn(llsOld = xGetScssnDown(s, 0)) ) {
        xClose(llsOld);
  }
  xSetSessnDown(s, 0, lls);
  return 0;
}

/*
 * Misc. routines
 */
static int
get_ident( s )
```

```
    Sessn    s;
{
    static int n = 1;
    return n++;
}

static void
callRedirect(ev, s)
    Event    ev;
    VOID     *s;
{
    xTrace1(ipp, 4, "ip: callRedirect runs with session %x", s);
    ipHandleRedirect((Sessn) s);
    xClose((Sessn)s);
    return;
}

typedef struct {
    int         (* affected)(
#ifdef __STDC__
    PState *, IPhost *, route *
#endif
    );
    route       *rt;
    PState      *pstate;
} RouteChangeInfo;

/*
 * ipRouteChanged -- For each session in the active map, determine if a
 * change in the given route affects that session.  If it does, the
 * session is reconfigured appropriately.
 */
void
ipRouteChanged(pstate, rt, routeAffected)
    PState *pstate;
    route *rt;
    int (*routeAffected)(
#ifdef __STDC__
    PState *, IPhost *, route *
#endif
);
{
    RouteChangeInfo    rInfo;

    rInfo.affected = routeAffected;
    rInfo.rt = rt;
    rInfo.pstate = pstate;
    mapForEach(pstate->activeMap, routeChangeFilter, &rInfo);
    mapForEach(pstate->fwdMap, routeChangeFilter, &rInfo);
}

static int
routeChangeFilter(key, value, arg)
    VOID *key, *value, *arg;
{
    RouteChangeInfo    *rInfo = (RouteChangeInfo *)arg;
    Sessn              s = (Sessn)value;
    SState             *state;

    xAssert(xIsSessn(s));
    state = (SState *)s->state;
    xTrace3(ipp, 4, "ipRouteChanged does net %s affect ses %x, dest %s?",  ipHostStr(&rInfo->rt->net), s, ipHostStr(&state->hdr.dest));
    if ( rInfo->affected(rInfo->pstate, &state->hdr.dest, rInfo->rt) ) {
            xTrace1(ipp, 4,
                    "session %x affected -- reopening lower session", s);
            xDuplicate(s);
            evDetach( evSchedule(callRedirect, s, 0) );
    } else {
            xTrace1(ipp, 4, "session %x unaffected by routing change", s);
    }
    return MFE_CONTINUE;
}
```

```
            /*
             * Functions used as arguments to ipRouteChanged
             */
/*
 * Return true if the remote host is not connected to the local net
 */
int
ipRemoteNet( ps, dest, rt )
   PState   *ps;
   IPhost   *dest;
   route    *rt;
{
   return ! ipHostOnLocalNet(ps, dest);
}

/*
 * Return true if the remote host is on the network described by the
 * route.
 */
int
ipSameNet(pstate, dest, rt)
   PState *pstate;
   IPhost *dest;
   route *rt;
{
   return ( (dest->a & rt->mask.a) == (rt->net.a & rt->mask.a) &&
            (dest->b & rt->mask.b) == (rt->net.b & rt->mask.b) &&
            (dest->c & rt->mask.c) == (rt->net.c & rt->mask.c) &&
            (dest->d & rt->mask.d) == (rt->net.d & rt->mask.d) );
}
```