# The ASP Protocol Explained

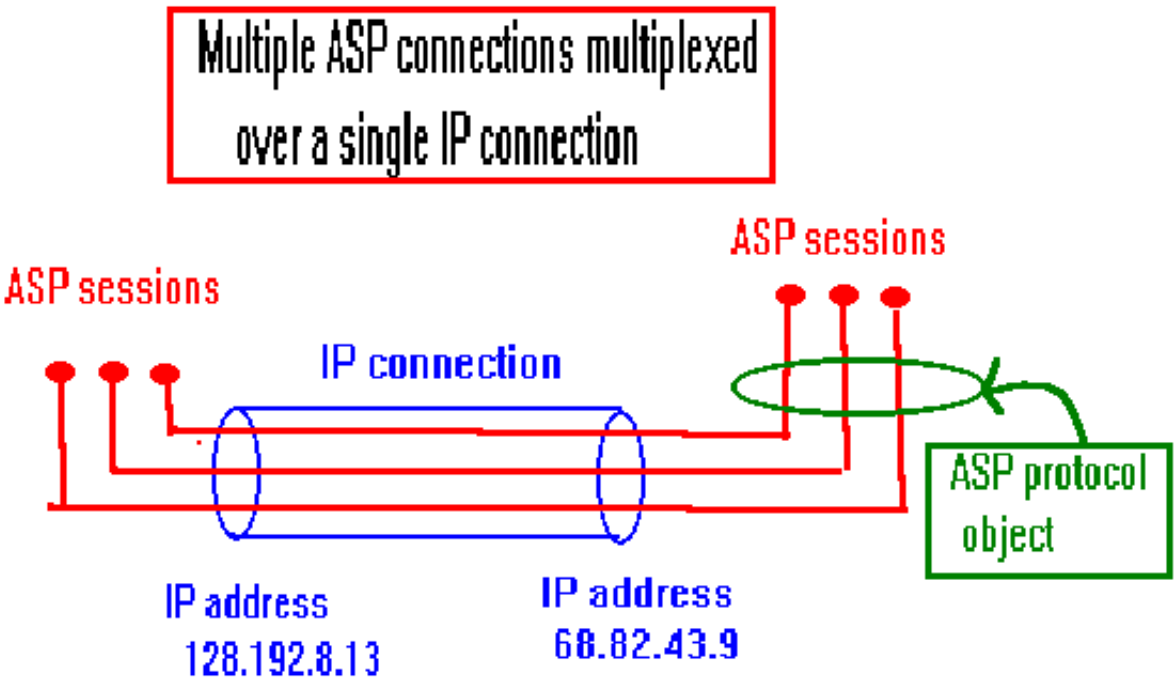
*By Dr. Dan Everett, University of Georgia*

ASP is a bare-bones transport-layer protocol which uses IP to transmit messages to an ASP "port" at the destination host. ASP is somewhat similar to UDP in the real world.

The main purpose of ASP is as a simple example, to demonstrate the infrastructure which has to be in place to implement even the simplest protocol. As such, it makes a good template for constructing other protocols. ASP itself is really not that simple, at least not until you understand the x-kernel!

## What does ASP do?

ASP performs a simple **multiplexing** function. That is, there can be several ASP sessions between different pairs of processes on the same two computers. In this case, there is an IP "connection" between the two computers -- the two endpoints are represented by the IP addresses of the two machines. But each endpoint of a separate ASP session has a distinct **ASP port address**. The ASP port address, plus the IP address, uniquely identifies the endpoint of the ASP connection.




This very schematic diagram shows the crucial difference between **session objects** and **protocol objects**

- Each **session object** represents the endpoint of an ASP connection. As ASP is such a simple protocol, it turns out that the session needs to hold no data except for a template message header.
- The **ASP protocol object** is responsible for directing incoming ASP messages to the

- The **ASP protocol object** is responsible for directing incoming ASP messages to the appropriate connections. The protocol object contains the **ID map** which is essentially a list of all open connections.

## The External ASP header file

```
/*
 * asp.h
 *
 * x-kernel v3.3
 *
 * Copyright (c) 1993,1991,1990,1996  Arizona Board of Regents
 */

typedef u_short ASPport;
```

--Is that all? Yes, this is all that any other protocol needs to know about ASP.

A higher-level protocol or application (HLP), which wants to use ASP to transmit messages to its peer on another machine, would need to know the IP address of the remote machine and the ASP port number of its peer. The ASP port address must have some data type, which is what the external header file provides.

But doesn't the HLP need to know what functions are provided by ASP, such as opening and closing connections, transmitting messages, etc? In ASP, these functions all follow a standard interface. For example, to open an ASP connection the HLP will contain code which looks something like this:

- xOpen(myProtl, LLP, p)

where myProtl is the HLP protocol object, LLP is the lower-level protocol object (in this case the ASP protocol object), and p is the **participant list**, which contains the addresses of the "participants" on both ends of the conversation.

The generic "xOpen" call gets linked into the aspOpen() call which will appear later as part of the ASP implementation. This linking is performed by a complex "make" operation, which we'll discuss later.

Next: The asp internal header file, asp_internal.h

# ASP Internal Header file

This file contains the declarations which are used by the ASP implementation file, **asp.c**. As such, it's much more detailed than the external header file.

```
/*
 * $RCSfile: asp_internal.h,v $
 *
 * x-kernel v3.3
 *
 * Copyright (c) 1993,1991,1990,1996  Arizona Board of Regents
 */

#include "xkernel.h"
#include "ip.h"
#include "asp.h"

/* ASP message header definition */

typedef struct header {
    ASPport sport;  /* source port      */
    ASPport dport;  /* destination port */
    u_short ulen;   /* ASP length       */
} ASPhdr;
```

Examine the header structure carefully. This is the information which the transmitting end of an ASP connection attaches to the data which it is sending to the receiving end. For this simple protocol, we send only the source and destination port addresses and ulen, the data length of the ASP message itself.

```
#define HLEN (sizeof(ASPhdr))

/* protocol and session states */

typedef struct pstate {
    Map activemap;
    Map passivemap;
} ProtlState;
```

In order for a connection to be established between sender and receiver, two things have to happen in this order:

1. First, the receiver must indicate that it is willing to accept a connection. This operation is called a **passive open** or **open enable**.
2. Then, the sender **actively opens** a connection to that port.

The ASP protocol object stores a list of all the passively opened ports in its **passive map**, and a list of all the actually open connections in its **active map**, as shown in the ProtlState structure above.

```
typedef struct sstate {
    ASPhdr hdr;
} SessnState;
```

```
typedef struct sstate {
    ASPhdr hdr;
} SessnState;
```

The Session State object is the only place where you can store information which persists over the lifetime of a connection. For example, if you wanted a counter which showed the number of the next message which you were going to send, or a queue of messages waiting to be sent, it would go here.

The ASP session state consists of a template header file, with the source and destination port filled in and the data length, ulen, left blank. Note that all the messages sent through a given session will always have the same source and destination addresses, but may have different message lengths.

```
/* active and passive maps */

typedef struct {
    Sessn   lls;
    ASPport localport;
    ASPport remoteport;
} ActiveId;

typedef ASPport PassiveId;
```

This shows the information contained in the active and passive maps. The "lls" member of ActiveID is the lower-level session. From the viewpoint of ASP, "lls" is the entry point into an IP connection.

A passive map entry (PassiveId) is just the port number of an ASP port which has been enabled for some remote participant to open an ASP connection to it.

```
#define ACTIVE_MAP_SIZE  101
#define PASSIVE_MAP_SIZE 23
```

These numbers for the size of the respective maps appear to have been chosen at random.

```
/* UPI function declarations */

void            asp_init(Protl);
static Sessn    aspOpen(Protl, Protl, Protl, Part *);
static XkReturn aspOpenEnable(Protl, Protl, Protl, Part *);
static XkReturn aspDemux(Protl, Sessn, Msg *);
static XkHandle aspPush(Sessn, Msg *);
static XkReturn aspPop(Sessn, Sessn, Msg *, void *);
static Sessn    aspCreateSessn(Protl, Protl, Protl, ActiveId *);
static XkReturn aspClose(Sessn);
static int      aspControlProtl(Protl, int, char *, int) ;
static int      aspControlSessn(Sessn, int, char *, int) ;
static Part    *aspGetParticipants(Sessn);
```

These are the functions, implemented in asp.c, which are called by the HLP which wants to use ASP. In other words, they are the Service Access Points (SAP) of ASP.

```
/* internal function declarations */
```

```
/* internal function declarations */

static void     getproc_protl(Protl);
static void     getproc_sessn(Sessn);
static long     aspHdrLoad(ASPhdr *, char *, long);
static void     aspHdrStore(ASPhdr *, char *, long);
```

These "internal" functions are called only within asp.c, and are not exposed to the outside world.

```
/* trace variable used in conjunction with xTrace stmts */
int traceaspp;
```

This variable controls the level of detail of the debugging output statements which xkernel generates internally. A recommended initial setting is TR_MAJOR_EVENTS. See the X-kernel manual for more detail.

Next: The asp implementation file, asp.c

# ASP Implementation file, asp.c

This file actually does the work of defining the actions of ASP. It's worth studying in some detail.

---

```
/*
 * $RCSfile: asp.c,v $
 *
 * x-kernel v3.3
 *
 * Copyright (c) 1993,1991,1990,1996  Arizona Board of Regents
 */

#include "asp_internal.h"

void
asp_init(Protl self)
```

---

This will initialize the protocol object itself. The steps are as follows:

1. The active and passive maps will be created, in an initially empty state
2. The lower-level protocol will be passively opened (in order to open an ASP connection, we must have an IP connection too).

---

```
{
    ProtlState *pstate;
    Protl      llp;
    Part       part;

    getproc_protl(self);

    /* create and initialize protocol state */
    pstate = X_NEW(ProtlState);
    bzero((char *)pstate, sizeof(ProtlState));
    self->state = (void *)pstate;
    pstate->activemap  = mapCreate(ACTIVE_MAP_SIZE, sizeof
(ActiveId));
    pstate->passivemap = mapCreate(PASSIVE_MAP_SIZE, sizeof
(PassiveId));

    /* find lower level protocol and do a passive open on it */
    llp = xGetProtlDown(self, 0);
    if (!xIsProtl(llp))
        Kabort("ASP could not get lower protocol");
```

---

---

Here ASP is using the special x-kernel functions which find the lower-level protocol which is associated with it. You provide this information in the **protocol graph** file during the x-kernel build process.

---

```
    partInit(&part, 1);
    partPush(part, ANY_HOST, 0);
```

---

Here we are creating a participant list whose elements are this host ("1") and any other host. When we call xOpenEnable with this participant list, we'll be allowing participants from any other host to make an IP connection to this one.

However, before a participant can make an ASP connection, some HLP must passively open an ASP port.

---

```
    if (xOpenEnable(self, self, llp, &part) == XK_FAILURE) {
        xTrace0(aspp, TR_ALWAYS,
                "asp_init: openenable on lower protocol
failed");
        xFree((char *) pstate);
        return;
    }
}

static void
getproc_protl(Protl p)
```

---

Here below is where the protocol object gets loaded with pointers to its methods. (It is like overloading a function in a derived class in C++, but this is in C).

Note that the protocol object's methods are all actions which do not take place in the context of an already open connection. They are:

- Opening a new connection (active open)
- Enabling a port for a remote participant to open a connection (passive open)
- Directing an incoming message to the appropriate receiving ASP session (demultiplexing)
- Miscellaneous control operations such as: get the maximum allowed packet size, find the local host id, *etc.*

---

```
{
    /* fill in the function pointers to implement protocol
operations */
```

```
    p->open         = aspOpen;
    p->openenable   = aspOpenEnable;
    p->demux        = aspDemux;
    p->controlprotl = aspControlProtl;
}


static Sessn
aspOpen(Protl self, Protl hlp, Protl hlpType, Part *p)
```

---

Here is where we perform an active open. That is, we want to create an ASP connection between this host and another remote host. We will have to provide a port number for this host and a remote port number, as well as the IP address of this host and a remote IP number. All this information will be packaged in the parameter p.

If successful, aspOpen() will return a pointer to a "session" data structure. Recall that a session represents one end of an active connection. If we fail, an error code will be returned.

---

```
{
ActiveId key;
Sessn asp_s, lls;

ProtlState *pstate = (ProtlState *)self->state;


bzero((char *)&key, sizeof(key));

/* high level protocol must specify both local and remote ASP
port */

key.remoteport = *((ASPport *)partPop(p[0]));
key.localport = *((ASPport *)partPop(p[1]));
```

---

We pop off the ASP port address of both the local port p[1] and the remote port p[0]. What's left is the IP address. Next, we will attempt to open an IP connection to the remote host.

---

```
/* attempt to open session on protocol below this one */
lls = xOpen(self, self, xGetProtlDown(self, 0), p);
if (lls != ERR_SESSN) {
key.lls = lls;
/* check for this session in the active map */
if (mapResolve(pstate->activemap, &key, (void **)&asp_s)
== XK_FAILURE){
```

---

Now we're going to create a session and declare the open a success. You may notice that

Now we're going to create a session and declare the open a success. You may notice that we aren't checking the other side to make sure they have open-enabled the port to which we are trying to connect! ASP is too primitive a protocol to include a method for checking this.

---

```
/* session wasn't already in map, so initialize it */
asp_s = aspCreateSessn(self, hlp, hlpType, &key);
if (asp_s != ERR_SESSN) /* A successful open! */
return asp_s;
}
/* if control makes it this far, an error has occurred */
xClose(lls);
}
return ERR_SESSN;
}

static XkReturn
aspOpenEnable(Protl self, Protl hlp, Protl hlpType, Part *p)
```

---

This is where we "passively open" a port, that is, we make it accessible to be actively opened by a remote port. The parameter "p" contains the number of the remote port which will be authorized to open a connection to us; this value might be the special constant ANY_PORT.

The procedure is simple: check if the combination of local and remote ports is already in the passive map; if not, put it there.

We maintain a reference count rcnt which counts the number of HLP entities which have open-enabled this port. If one of the HLP entities changes its mind and issues and OpenDisable call, the reference count is decremented. When the reference count becomes zero, the entry will be removed from the passive map.

---

```
PassiveId key;
ProtlState *pstate = (ProtlState *)self->state;
Enable *e;

key = *((ASPport *)partPop(*p));

/* check if this port has already been openenabled */
if (mapResolve(pstate->passivemap, &key, (void **)&e)
!= XK_FAILURE) {
if (e->hlp == hlp) {
/* this port was openenabled previously by the same hlp */
e->rcnt++;
return XK_SUCCESS;
}
/* this port was openenabled previously by a different hlp - error
*/
return XK_FAILURE;
}
```

```
return XK_FAILURE;
}
```

---

The Enable object created below will be stored in the passive map by the mapBind call. It contains all the information needed to use the entry. The "e->hlp" member is needed when a message comes in on this port - this tells xkernel what higher level protocol should handle this message.

---

```
/* this will be a new enabling, so create and initialize Enable
object, */
/* and enter the binding of port/enable object in the passive
map */
e = X_NEW(Enable);
e->hlp = hlp;
e->hlpType = hlpType;
e->rcnt = 1;
e->binding = mapBind(pstate->passivemap, &key, e);

if (e->binding == ERR_BIND) {
xFree((char *)e);
return XK_FAILURE;
}
return XK_SUCCESS;
}

static Sessn
aspCreateSessn(Protl self, Protl hlp, Protl hlpType, ActiveId
*key)
```

---

This is called by aspOpen() above to create a session data structure when a new connection is opened. The session data structure contains a list of the protocols involved, plus a template message header with the source and destination addresses. These addresses are contained in the "key" parameter.

Since we are creating an active connection, the key is entered into the active map.

---

```
{
Sessn s;
ProtlState *pstate = (ProtlState *)self->state;
SessnState *sstate;
ASPhdr *asph;
```

---

First create a generic xkernel session object s. Then, create a SessnState structure (defined in asp_internal.h). Recall that the ASP session state contains only a template header with the source and destination addresses. Finally, the generic xkernel session contains a "state" pointer, which we attach to the SessnState structure.

---

```
/* create the session object and initialize it */
```

```
/* create the session object and initialize it */
s = xCreateSessn(getproc_sessn, hlp, hlpType, self, 1, &key->lls);

s->binding = mapBind(pstate->activemap, key, s);
sstate = X_NEW(SessnState);
s->state = (char *)sstate;

/* create an ASP header */
asph = &(sstate->hdr);
asph->sport = key->localport;
asph->dport = key->remoteport;
asph->ulen = 0;
```

---

The header template defines the message length as zero; this will be overwritten when we actually send a message in aspPush() below.

---

```
return s;
}

static void
getproc_sessn(Sessn s)
```

---

Here is where we attach the methods to the Session object. If this were a C++ program we would do this by overloading functions, but it isn't.

Unlike the methods in the Protocol object, these are the methods which make sense in the context of an open connection. "Push" sends data into the session, for delivery on the other end; "Pop" gets data out of the session which was pushed into it on the other end.

---

```
{
/* fill in the function pointers to implement session operations
*/
s->push = aspPush;
s->pop = aspPop;
s->controlsessn = aspControlSessn;
s->getparticipants = aspGetParticipants;
s->close = aspClose;
}

static XkReturn
aspDemux(Protl self, Sessn lls, Msg *dg)
```

---

aspDemux is one of the methods of the protocol object. We look at the header of the message and look for its destination port in the active map (the list of all open connections). If it is there, we will deliver the message to the appropriate session.

If the session doesn't exist, this isn't necessarily an error. ASP doesn't have a connection-establishment phase; the receiving side just passively opens a port and waits for data to arrive. If this is the first data to arrive and the port has been passively opened, then we will

arrive. If this is the first data to arrive and the port has been passively opened, then we will go ahead and create a connection.

But if the receiving port has not been passively opened, the attempt to send a message will fail. A more sophisticated protocol might return a "connection refused" message over the network back to the sender, but ASP simply discards the message and gives it no further thought.

The "demux" routine is actually called by the lower-level protocol when a message arrives from the other participant. "lls" is the session for the lower-level protocol.

---

```
{
char *buf;
ASPhdr h;
ActiveId activeid;
ProtlState *pstate = (ProtlState *)self->state;
Sessn s;
PassiveId passiveid;
Enable *e;

/* extract the header from the message */
buf = msgPop(dg, HLEN);
```

---

```
Recall that HLEN is the length of the ASP header. The header
information
will be removed from the message before passing it up the protocol
stack to the HLP.
```

---

```
if (buf == NULL)
return XK_FAILURE;
aspHdrLoad(&h, buf, HLEN);


Now we copy the header from "buf" to "h".
The reason we need to do this will be shown when we discuss aspHdrLoad
()
below.


/* construct a demux key from the header */
bzero((char *)&activeid, sizeof(activeid));
activeid.lls = lls;
activeid.localport = h.dport;
activeid.remoteport = h.sport;

/* see if demux key is in the active map */
if (mapResolve(pstate->activemap, &activeid, (void **)&s)
== XK_FAILURE) {
```

```
== XK_FAILURE) {
/* didn't find an active session, so check passive map */
passiveid = h.dport;
if (mapResolve(pstate->passivemap, &passiveid, (void **)&e)
==
XK_FAILURE) {
/* drop the message */
return XK_SUCCESS;
}
/* port was enabled, so create a new session and inform hlp */

s = aspCreateSessn(self, e->hlp, e->hlpType, &activeid);

if (s == ERR_SESSN)
return XK_SUCCESS;
```

---

I do not know why we should return a "success" code in this case.

---

```
xDuplicate(lls);
```

---

The xDuplicate() call increments the reference count on the session lls. The reference count is the number of sessions that use lls as their lower-level session, and will increase and decrease as sessions are opened and closed. When the reference count reaches zero, lls should itself be closed.

---

```
xOpenDone(e->hlp, self, s);
```

The xOpenDone is a method of the higher level protocol. By default, it does nothing, but you can use this if your protocol needs to take some special action whenever a session is opened. Note that this particular session is opened by a remote participant.

---

```
}

/* found (or created) an appropriate session, so pop to it */

return xPop(s, lls, dg, &h);
}

static long
aspHdrLoad(ASPhdr *hdr, char *src, long len)
```

---

The following two functions are necessary because the two byte order in which the receiver stores data might be different from the sender. For example the sender might be an IBM PC with 16-bit integers and the receiver might be a Unix host with 32-bit integers.

---

```
{
/* copy from src to hdr, then convert network byte order to host
```

```
{
/* copy from src to hdr, then convert network byte order to host
order */
bcopy(src, (char *)hdr, HLEN);
hdr->ulen = ntohs(hdr->ulen);
hdr->sport = ntohs(hdr->sport);
hdr->dport = ntohs(hdr->dport);
return HLEN;
}

static void
aspHdrStore(ASPhdr *hdr, char *dst, long len)
{
/* convert host byte order to network order, then copy from hdr
to dst */
/* (note: argument 'hdr' is changed by the following code) */

hdr->ulen = htons(hdr->ulen);
hdr->sport = htons(hdr->sport);
hdr->dport = htons(hdr->dport);
bcopy((char *)hdr, dst, HLEN);
}
```

---

aspPush is where the HLP "pushes" data into an ASP session for delivery to the other side. The function returns a memory handle to the header which was attached to the message. The "msg" parameter contains the actual data which is transmitted by ASP.

The "push" procedure is simple:

1. Attach a header to the message (converting it to network byte order using aspHdrStore);
2. Push the resulting message into the lower-level session for this protocol, "lls"

Note "lls" was created when we actively opened the connection.

---

```
static XkHandle aspPush(Sessn self, Msg *msg)
{
SessnState *sstate = (SessnState *)self->state;
ASPhdr hdr;
char *buf;

/* create a header by inserting length into header template */

hdr = sstate->hdr;
hdr.ulen = msgLength(msg) + HLEN;

/* attach header to message and pass it on down the stack */
buf = msgPush(msg, HLEN);
aspHdrStore(&hdr, buf, HLEN);
return xPush(xGetSessnDown(self, 0), msg);
}
```

---

The aspPop routine is called from the lower-level session when a message arrives which is destined for this ASP session. In ASP it simply truncates any bytes which were added to the message and passes the result up the protocol stack.

---

```
static XkReturn
aspPop(Sessn self, Sessn lls, Msg *msg, void *hdr)
{
ASPhdr *h = (ASPhdr *)hdr;

/* truncate message to length shown in header */
if (h->ulen - HLEN < msgLength(msg))
msgTruncate(msg, (int)h->ulen);

/* pass the message to the next protocol up the stack */
return xDemux(xGetUp(self), self, msg);
}
```

---

ControlProtl() is used by the HLP to query this protocol. The result of the query will be put into "buf", a return buffer of length "len". The function should return an error code of -1 if the buffer is not long enough to hold the result.

If the HLP asks for the maximum or optimum packet size, then the answer is HLEN less than the maximum or optimum packet size of the LLP (because ASP will attach a header of size HLEN bytes to the message it gets from HLP, and this becomes a message pushed to LLP). Note that the buffer must be at least the size of an integer in this case.

Otherwise, the control request is passed along to the LLP.

The function returns the number of bytes it writes to the buffer.

---

```
static int
aspControlProtl(Protl self, int opcode, char *buf, int len)
{
switch (opcode) {
      case GETMAXPACKET:
      case GETOPTPACKET:
         checkLen(len, sizeof(int));
         if (xControlProtl(xGetProtlDown(self, 0), opcode, buf, len)
              < sizeof(int))
             return -1;
          *(int *)buf -= HLEN;
          return sizeof(int);
       default: return
             xControlProtl(xGetProtlDown(self, 0), opcode, buf, len);
}
int aspControlSessn(Sessn self, int opcode, char *buf, int len)
SessnState *sstate = (SessnState *)self->
{
    SessnState *sstate = (SessnState *)self->state;
    ASPhdr     *hdr;

    hdr   &(sstate->hdr);
```

```
    hdr = &(sstate->hdr);
    switch (opcode) {
        case GETMYPROTO:
            checkLen(len, sizeof(long));
            *(long *)buf = sstate->hdr.sport;
            return sizeof(long);
        case GETPEERPROTO:
            checkLen(len, sizeof(long));
            *(long *)buf = sstate->hdr.dport;
            return sizeof(long);
        case GETMAXPACKET:
        case GETOPTPACKET:
            checkLen(len, sizeof(int));
            if (xControlSessn(xGetSessnDown(self, 0), opcode, buf,
len) <
                  sizeof(int))
                return -1;
            *(int *)buf -= HLEN;
            return sizeof(int);
        default:
            return xControlSessn(xGetSessnDown(self, 0), opcode, buf,
len);
    }
}

static Part *
aspGetParticipants(Sessn self)
{
    Part       *p;
    int        numParts;
    SessnState *sstate = (SessnState *)self->state;
    long       localPort, remotePort;

    p = xGetParticipants(xGetSessnDown(self, 0));
    if (!p)
        return NULL;
    numParts = partLength(p);
    if (numParts > 0 && numParts <= 2) {
        if (numParts == 2) {
            localPort = (long)sstate->hdr.sport;
            partPush(p[1], (void *)&localPort, sizeof(long));
        }
        remotePort = (long)sstate->hdr.dport;
        partPush(p[0], (void *)&remotePort, sizeof(long));
       return p;
    }
    else    /* Bad number of participants */
        return NULL;
}

static XkReturn
aspClose(Sessn s)
{
     ProtlState *pstate = (ProtlState *)xMyProtl(s)->state;

     /* remove this session from the active map */
     mapRemoveBinding(pstate->activemap, s->binding);
```

```
	mapRemoveBinding(pstate->activemap, s->binding);

	/* close the lower level session on which it depends */
	xClose(xGetSessnDown(s, 0));

	/* de-allocate the session object itself */
	xDestroySessn(s);

	return XK_SUCCESS;
}
```

Next: The ASP test program, asptest.c