Elements of a Protocol Implementation

- Outline
  - Service Interface
  - Process Model
  - Common Subroutines
  - Example Protocol

Socket API

- Creating a socket

```
int socket(int domain, int type, int protocol)
```

  - domain=PF_INET, PF_UNIX
  - type=SOCK_STREAM, SOCK_DGRAM

- Passive open on server

```
int bind(int socket, struct sockaddr *address,
         int addr_len)
int listen(int socket, int backlog)
int accept(int socket, struct sockaddr *address,
           int *addr_len)
```

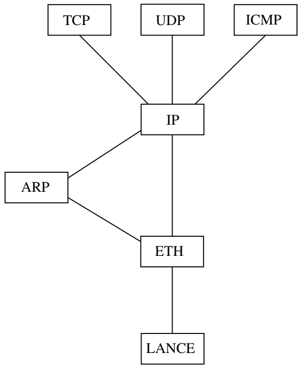
- Active open on client

```
int connect(int socket, struct sockaddr *address,
            int addr_len)
```

- Sending and receiving messages

```
int send(int socket, char *buf, int len, int flags)
int recv(int socket, char *buf, int len, int flags)
```

Protocol Objects




```
name=lance;
name=eth protocols=lance;
name=arp protocols=eth;
name=ip protocols=eth,arp;
name=icmp protocols=ip;
name=udp protocols=ip;
name=tcp protocols=ip;
```


Operations on Protocol Objects

- Active Open

```
Sessn xOpen(Protl hlp,
            Protl llp,
            Part *participants)
```

- Passive Open

```
XkReturn xOpenEnable(Protl hlp,
                     Protl llp,
                     Part *participant)

XkReturn xOpenDone(Protl hlp,
                   Protl llp,
                   Sessn session,
                   Part *participants)
```

- Demultiplexing

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

Session Objects

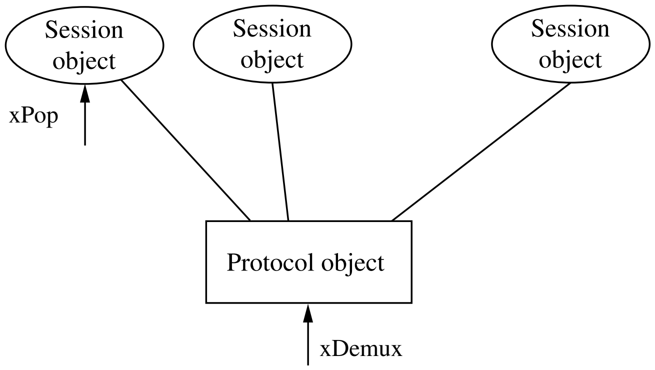
- Local end-point of a channel
- Interpret messages and maintain channel state
- Export operations for sending and receiving messages
- Operations
  - send a message

```
XkReturn xPush(Sessn lls,
               Msg *message)
```

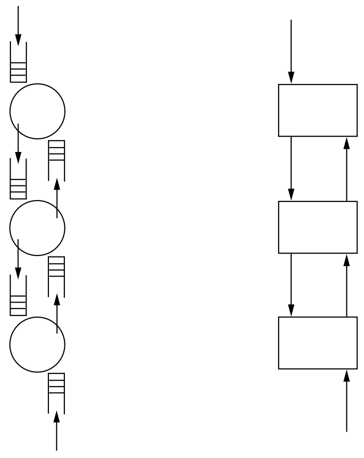
  - deliver a message

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


Relationship Between Protocols and Sessions



Process Model



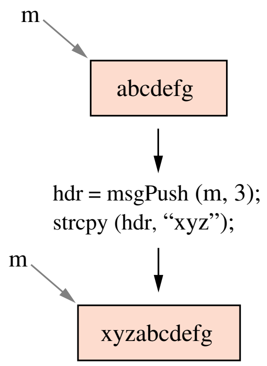
(a) (b)

- Synchronization Primitives

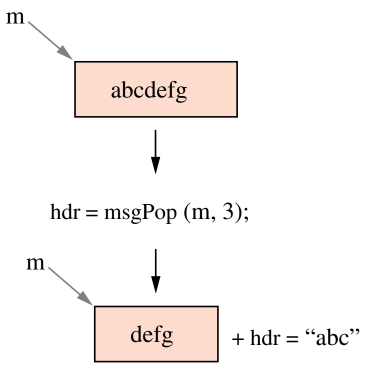
```
void semInit(Semaphore *s, int count)
void semSignal(Semaphore *s)
void semWait(Semaphore *s)
```


Message Library

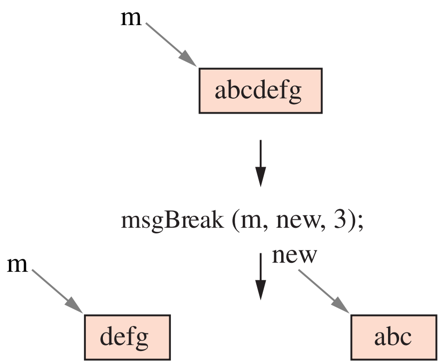
- Add a header to the front of a message



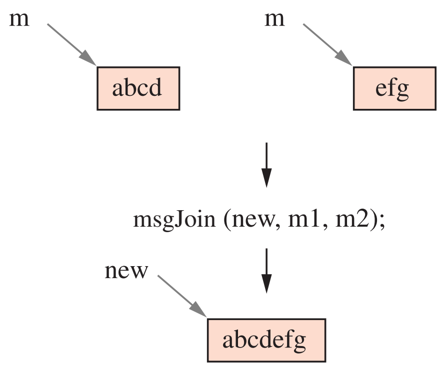
- Strip a header from the front of a message



- Fragment a message into two messages



- Reassemble two messages into one message

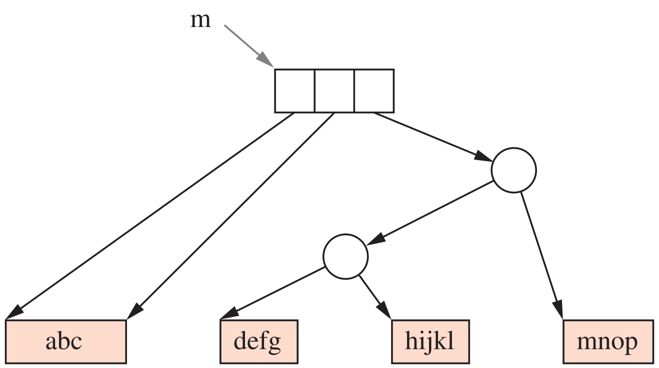


- Other Operations

```
void msgConstructEmpty(Msg *message)

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

char *msgConstructAllocate(Msg *message,
                           int length)

void msgAssign(Msg *message_1,
               Msg *message_2)

int msgLength(Msg *message)
```

Implementating Messages

- Tree Data Structure
  - avoid data copies



- Traversal Operations

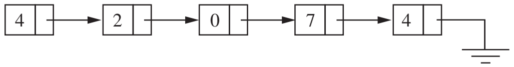
```
void msgWalkInit(MsgWalk *cxt,
                 Msg *message)

char *msgWalkNext(MsgWalk *cxt,
                  int *len)

void msgWalkDone(MsgWalk *cxt)
```


Event Library

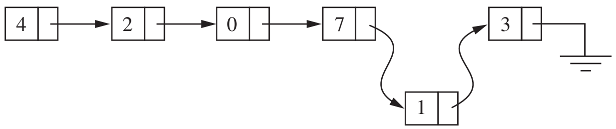
- Scheduling Timeouts

```
Event evSchedule(EvFunc function,
                 void *argument,
                 int time)

EvCancelReturn evCancel(Event event)
```

- Implementation
  - delta list



  - insert event at time 14



Map Library

- Demultiplexing Packets

```
Map mapCreate(int number,
              int size)

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

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

- Usage
  - one map to remember active opens (active map)
  - one map to remember passive opens (passive map)
- Implementation
  - hash table
  - specialized code based on key size
  - one-element cache

Example

```
static Sessn
aspOpen(Protl self, Protl hlp, Part *p)
{
    Sessn       asp_s;
    Sessn       lls;
    ActiveId    key;
    ProtlState  *pstate = (ProtlState *)self->state;

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

    /* High level protocol must specify both */
    /* local and remote ASP port */
    key.localport = *((AspPort *) partPop(p[0]));
    key.remoteport = *((AspPort *) partPop(p[1]));

    /* Assume failure until proven otherwise */
    asp_s = XK_FAILURE;
```

```
    /* Open session on low-level protocol */
    lls = xOpen(self, xGetDown(self, 0), p);
    if ( lls != XK_FAILURE )
    {
        key.lls = lls;
        /* Check for session in active map */
        if (mapResolve(pstate->activemap, &key,
            (void **)&asp_s) == XK_FAILURE)
        {
         /* Session not in map; initialize it */
            asp_s = asp_init_sessn(self, hlp, &key);
            if ( asp_s != XK_FAILURE )
            {
                /* A successful open */
                return asp_s;
            }
        }

        /* Error has occurred */
        xClose(lls);
    }
    return asp_s;
}
```



```
static XkReturn
aspPush(Sessn s, Msg *msg)
{
    SessnState *sstate;
    AspHdr     hdr;
    void       *buf;

    /* create a header */
    sstate = (SessnState *) s->state;
    hdr = sstate->hdr;
    hdr.ulen = msgLen(msg) + HLEN;

    /* attach header and send */
    buf = msgPush(msg, HLEN);
    aspHdrStore(&hdr, buf, HLEN, msg);
    return xPush(xGetDown(s, 0), msg);
}
```

```
static XkReturn
aspDemux(Protl self, Sessn lls, Msg *msg)
{
    AspHdr       h;
    Sessn        s;
    ActiveId     activeid;
    PassiveId    passiveid;
    ProtlState   *pstate;
    Enable       *e;
    void         *buf;

    pstate = (ProtlState *)self->state;

    /* extract the header from the message */
    buf = msgPop(msg, HLEN);
    aspHdrLoad(&h, buf, HLEN, msg);

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



```
    /* see if demux key is in the active map */
    if (mapResolve(pstate->activemap, &activeid,
        (void **)&s) == 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 = asp_init_sessn(self, e->hlp, &activeid);
        xOpenDone(e->hlp, s, self);
    }
    /* pop the message to the session */
    return xPop(s, lls, msg, &h);
}
```

```
static XkReturn
aspPop(Sessn s, Sessn ds, Msg *msg, void *inHdr)
{
    AspHdr    *h = (AspHdr *) inHdr;

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

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