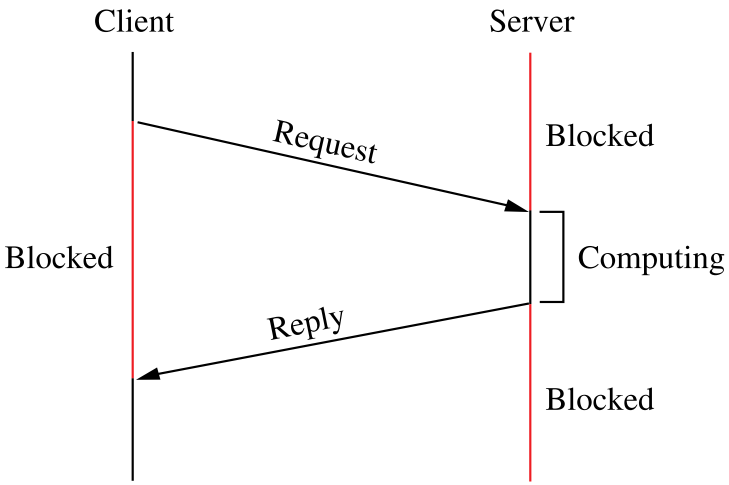
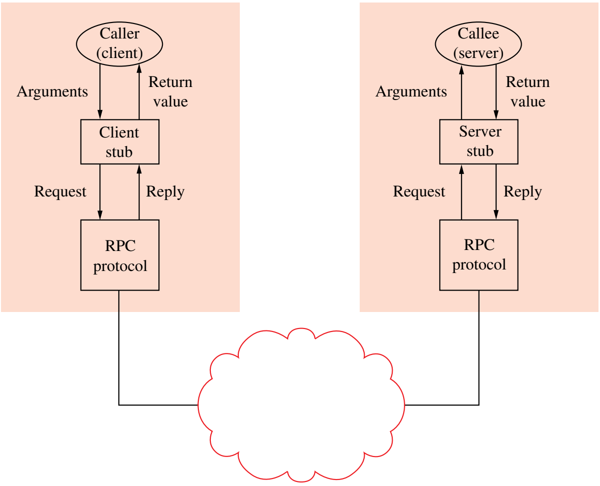
Remote Procedure Call

- Outline
  - Protocol Stack
  - Presentation Formatting

RPC Timeline



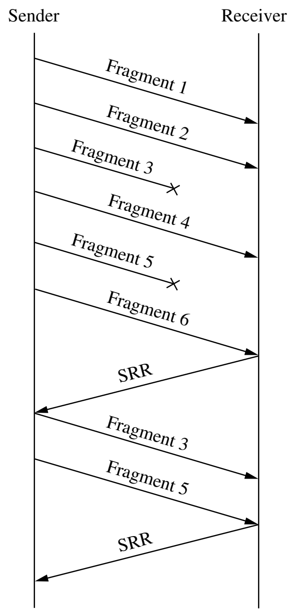
RPC Components



- Protocol Stack
  - BLAST: fragments and reassembles large messages
  - CHAN: synchronizes request and reply messages
  - SELECT: dispatches request to the correct process
- Stubs

Bulk Transfer (BLAST)

- Unlike AAL and IP, tries to recover from lost fragments
- Strategy
  - selective retransmission
  - aka partial acknowledgements



BLAST Details

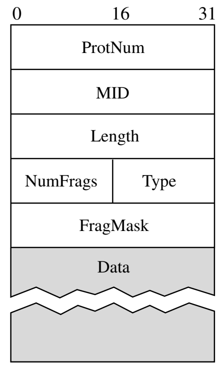
Sender:

- After sending all fragments, set timer DONE
- If receive SRR, send missing fragments and reset DONE
- If timer DONE expires, free fragments

Receiver:

- When first fragment arrives, set timer LAST_FRAG
- When all fragments present, reassemble and pass up
- Four exceptional conditions
  - if last fragment arrives but message not complete
    - send SRR and set timer RETRY
  - if timer LAST_FRAG expires
    - send SRR and set timer RETRY
  - if timer RETRY expires for first or second time
    - send SRR and set timer RETRY
  - if timer RETRY expires for third time
    - give up and free partial message

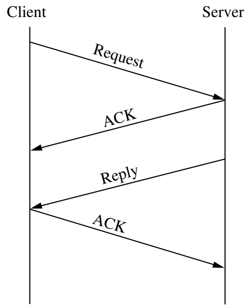
BLAST Header Format



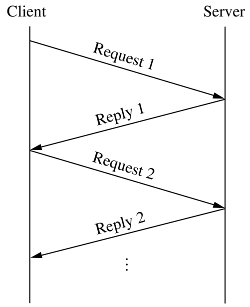
- `MID` must protect against wrap around
- `Type` = `DATA` or `SRR`
- `NumFrags` indicates number of fragments in message
- `FragMask` distinguishes among fragments:
  - if `Type=DATA`, identifies this fragment
  - if `Type=SRR`, identifies missing fragments

Request/Reply (CHAN)

- Guarantees message delivery
- Synchronizes client with server
- Supports *at-most-once* semantics
- Simple case



- Implicit Acknowledgements:



CHAN Details

- Lost message (request, reply, or ACK)
  - set RETRANSMIT timer
  - use message id (MID) field to distinguish
- Slow (long running) server
  - client periodically sends "are you alive" probe, or
  - server periodically sends "I'm alive" notice
- Want to support multiple outstanding calls
  - use channel id (CID) field to distinguish
- Machines crash and reboot
  - use boot id (BID) field to distinguish

CHAN Header Format

```
typedef struct {
    u_short  Type;    /* REQ, REP, ACK, PROBE */
    u_short  CID;     /* unique channel id */
    int      MID;     /* unique message id */
    int      BID;     /* unique boot id */
    int      Length;  /* length of message */
    int      ProtNum; /* high-level protocol */
} ChanHdr;
```

CHAN Session State

```
typedef struct {
    u_char    type;          /* CLIENT or SERVER */
    u_char    status;        /* BUSY or IDLE */
    int       retries;       /* number of retries */
    int       timeout;       /* timeout value */
    XkReturn  ret_val;       /* return value */
    Msg       *request;      /* request message */
    Msg       *reply;        /* reply message */
    Semaphore reply_sem;     /* client semaphore */
    int       mid;           /* message id */
    int       bid;           /* boot id */
} ChanState;
```


Synchronous versus Asynchronous Protocols

- Asynchronous Interface

```
xPush(Sessn s, Msg *msg)
xPop(Sessn s, Msg *msg, void *hdr)
xDemux(Protl hlp, Sessn s, Msg *msg)
```

- Synchronous Interface

```
xCall(Sessn s, Msg *req, Msg *rep)
xCallPop(Sessn s, Msg *req, Msg *rep,
         void *hdr)
xCallDemux(Protl hlp, Sessn s, Msg *req,
           Msg *rep)
```

- CHAN is a Hybrid Protocol
  - Synchronous from above: `xCall`
  - Asynchronous from below: `xPop/xDemux`

```
chanCall(Sessn self, Msg *msg, Msg *rmsg)
{
    ChanState   *state = (ChanState *)self->state;
    ChanHdr     *hdr;
    char        *buf;

    /* ensure only one transaction per channel */
    if ((state->status != IDLE))
        return XK_FAILURE;
    state->status = BUSY;

    /* save copy of req msg and ptr to rep msg*/
    msgConstructCopy(&state->request, msg);
    state->reply = rmsg;

    /* fill out header fields */
    hdr = state->hdr_template;
    hdr->Length = msgLen(msg);
    if (state->mid == MAX_MID)
        state->mid = 0;
    hdr->MID = ++state->mid;
```



```
    /* attach header to msg and send it */
    buf = msgPush(msg, HDR_LEN);
    chan_hdr_store(hdr, buf, HDR_LEN);
    xPush(xGetDown(self, 0), msg);

    /* schedule first timeout event */
    state->retries = 1;
    state->event = evSchedule(retransmit, self,
                              state->timeout);

    /* wait for the reply msg */
    semWait(&state->reply_sem);

    /* clean up state and return */
    flush_msg(state->request);
    state->status = IDLE;
    return state->ret_val;
}
```

```
retransmit(Event ev, int *arg)
{
    Sessn        s = (Sessn)arg;
    ChanState    *state = (ChanState *)s->state;
    Msg          tmp;

    /* see if event was cancelled */
    if ( evIsCancelled(ev) ) return;

    /* unblock client if we've retried 4 times */
    if (++state->retries > 4) {
       state->ret_val = XK_FAILURE;
       semSignal(state->rep_sem);
       return;
    }

    /* retransmit request message */
    msgConstructCopy(&tmp, &state->request);
    xPush(xGetDown(s, 0), &tmp);

    /* reschedule event with exponential backoff */
    evDetach(state->event);
    state->timeout = 2*state->timeout;
    state->event = evSchedule(retransmit, s,
                              state->timeout);
}
```

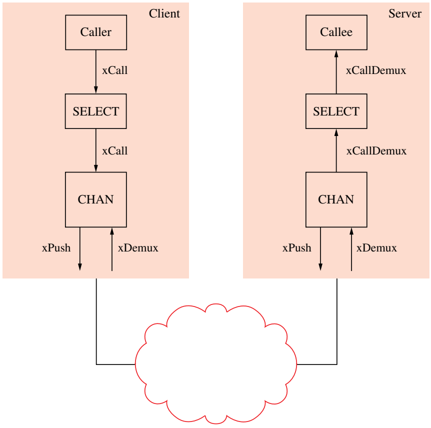


```
chanPop(Sessn self, Sessn lls, Msg *msg, void *inHdr)
{
    /* see if this is a CLIENT or SERVER session */
    if (self->state->type == SERVER)
       return(chanServerPop(self, lls, msg, inHdr));
    else
       return(chanClientPop(self, lls, msg, inHdr));
}
```

```
chanClientPop(Sessn self, Sessn lls, Msg *msg,
              void *inHdr)
{
    ChanState    *state = (ChanState *)self->state;
    ChanHdr      *hdr = (ChanHdr *)inHdr;

    /* verify correctness of msg header */
    if (!clnt_msg_ok(state, hdr))
        return XK_FAILURE;

    /* cancel retransmit timeout event */
    evCancel(state->event);

    /* if ACK, then schedule PROBE and exit*/
    if (hdr->Type == ACK)
    {
       state->event = evSchedule(probe, s, PROBE);
       return XK_SUCCESS;
    }

    /* save reply and signal client */
    msgAssign(state->reply, msg);
    state->ret_val = XK_SUCCESS;
    semSignal(&state->reply_sem);
    return XK_SUCCESS;
}
```


Dispatcher (SELECT)

- Dispatches to approprirate procedure
- Synchronous counterpart to UDP



- Address Space for Procedures
  - Flat: unique id for each possible procedure
  - Hierarchical: program + procedure within program

Example Code

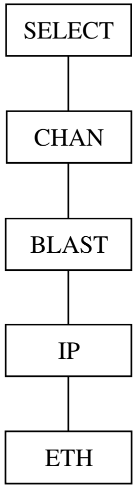
Client Side

```
static XkReturn
selectCall(Sessn self, Msg *req, Msg *rep)
{
    SelectState *state=(SelectState *)self->state;
    char        *buf;

    buf = msgPush(req, HLEN);
    select_hdr_store(state->hdr, buf, HLEN);
    return xCall(xGetDown(self, 0), req, rep);
}
```

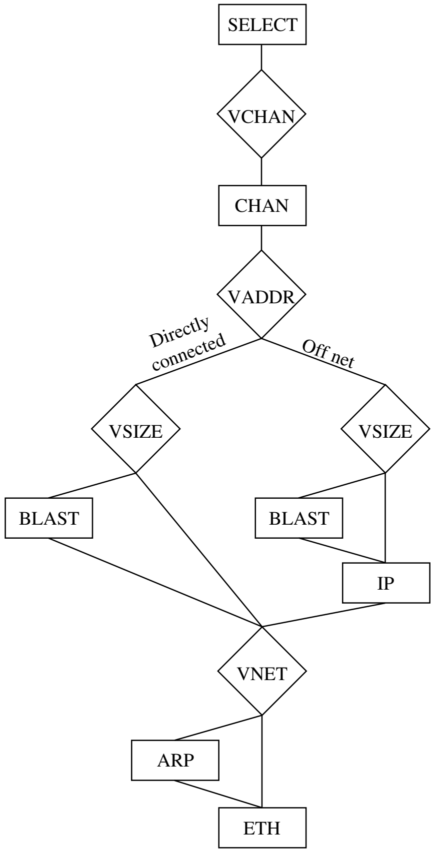
Server side

```
static XkReturn
selectCallPop(Sessn s, Sessn lls, Msg *req,
              Msg *rep, void *inHdr)
{
    return xCallDemux(xGetUp(s), s, req, rep);
}
```


Simple RPC Stack



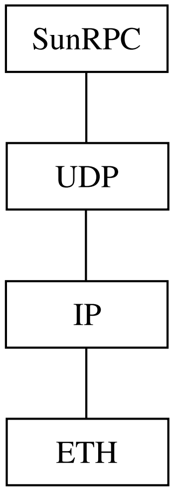
A More Interesting RPC Stack



VCHAN: A Virtual Protocol

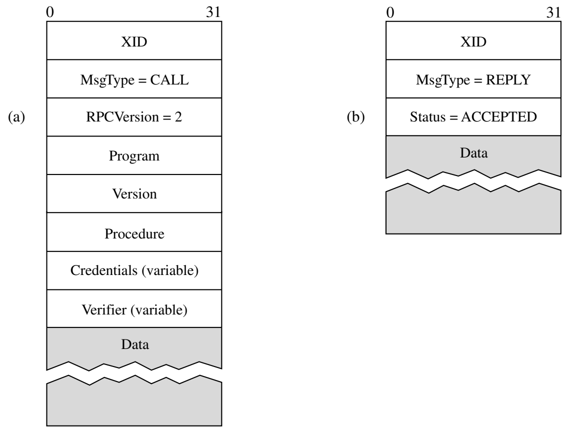
```
static XkReturn
vchanCall(Sessn s, Msg *req, Msg *rep)
{
    Sessn      chan;
    XkReturn   result;
    VchanState *state=(VchanState *)s->state;

    /* wait for an idle channel */
    semWait(&state->available);
    chan = state->stack[--state->tos];

    /* use the channel */
    result = xCall(chan, req, rep);

    /* free the channel */
    state->stack[state->tos++] = chan;
    semSignal(&state->available);

    return result;
}
```

SunRPC



- IP implements BLAST-equivalent
  - except no selective retransmit
- SunRPC implements CHAN-equivalent
  - except not at-most-once
- UDP + SunRPC implement SELECT-equivalent
  - UDP dispatches to program (ports bound to programs)
  - SunRPC dispatches to procedure w/in program

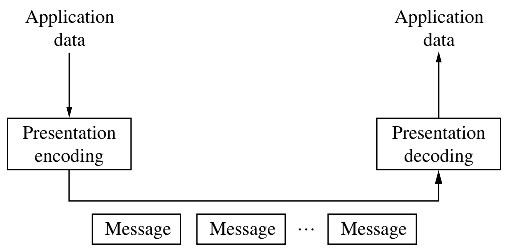
SunRPC Header Format



- `XID` (transaction id) is similar to CHAN's `MID`
- Server does not remember last `XID` it serviced
- Problem if client retransmits request while reply is in

Presentation Formatting

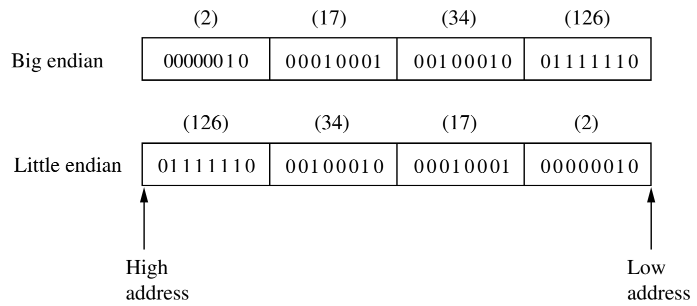
- Marshalling (encoding) application data into messages
- Unmarshalling (decoding) messages into application data



- Data types we consider:
  - integers
  - floating point numbers
  - character strings
  - arrays
  - structures
- Types of data we do not consider (now):
  - images
  - video
  - multimedia documents

Difficulties

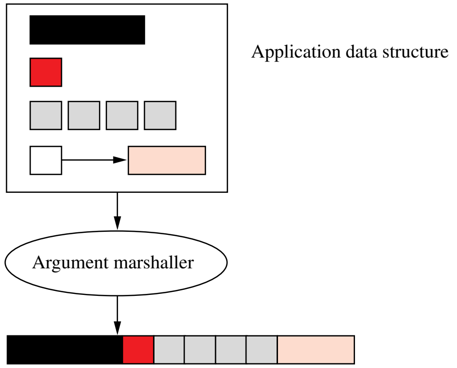
- Representation of base types
  - floating point: IEEE 754 versus non-standard
  - integer: big-endian versus little-endian (e.g., 34,677,374)



- Compiler layout of structures

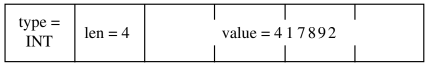
Taxonomy

- Data types
  - base types (e.g., ints, floats); must convert
  - flat types (e.g., structures, arrays); must pack
  - complex types (e.g., pointers); must linearize

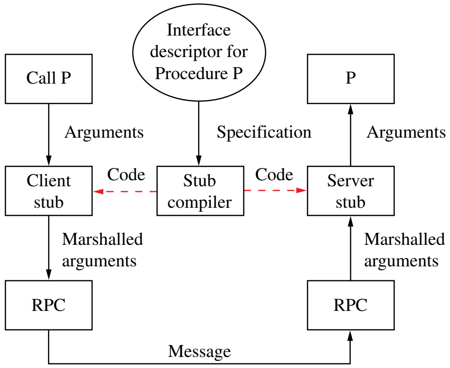


- Conversion Strategy
  - canonical intermediate form
  - receiver-makes-right (an $N \times N$ solution)

- Tagged versus Untagged data



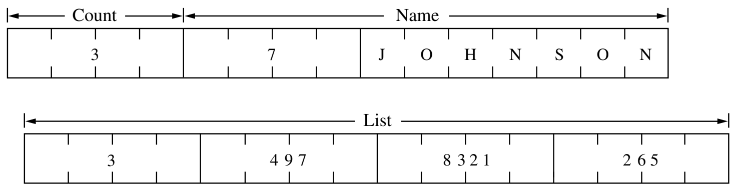
- Stubs
  - compiled
  - interpreted



Example: eXternal Data Representation (XDR)

- Defined by Sun for use with SunRPC
- C type system (without function pointers)
- Canonical intermediate form
- Untagged (except array length)
- Compiled stubs

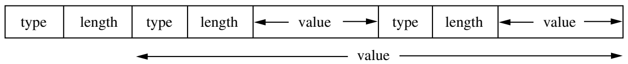

```
#define MAXNAME 256;
#define MAXLIST 100;

struct item {
   int     count;
   char    name[MAXNAME];
   int     list[MAXLIST];
};

bool_t
xdr_item(XDR *xdrs, struct item *ptr)
{
    return(xdr_int(xdrs, &ptr->count) &&
       xdr_string(xdrs, &ptr->name, MAXNAME) &&
       xdr_array(xdrs, &ptr->list, &ptr->count,
                 MAXLIST, sizeof(int), xdr_int));
}
```

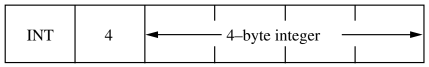


Example: Abstract Syntax Notation One (ASN.1)

- An ISO standard
- Essentially the C type system
- Canonical intermediate form
- Tagged
- Compiled or interpretted stubs
- BER: Basic Encoding Rules

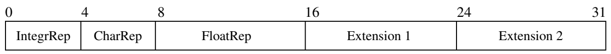
$\langle$`tag, length, value`$\rangle$





Example: Network Data Representation (NDR)

- Defined by DCE
- Essentially the C type system
- Receiver-makes-right (architecture tag)
- Individual data items untagged
- Compiled stubs from IDL
- 4-byte architecture definition tag



- `IntegrRep`
  - 0 = big-endian
  - 1 = little-endian
- `CharRep`
  - 0 = ASCII
  - 1 = EBCDIC
- `FloatRep`
  - 0 = IEEE 754
  - 1 = VAX
  - 2 = Cray
  - 3 = IBM