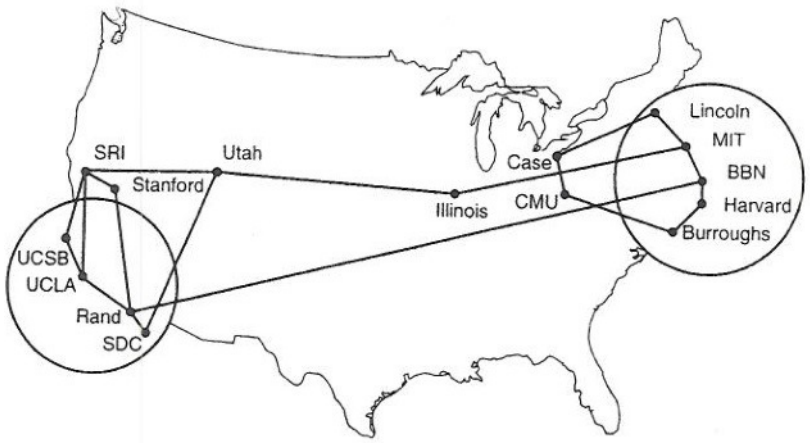
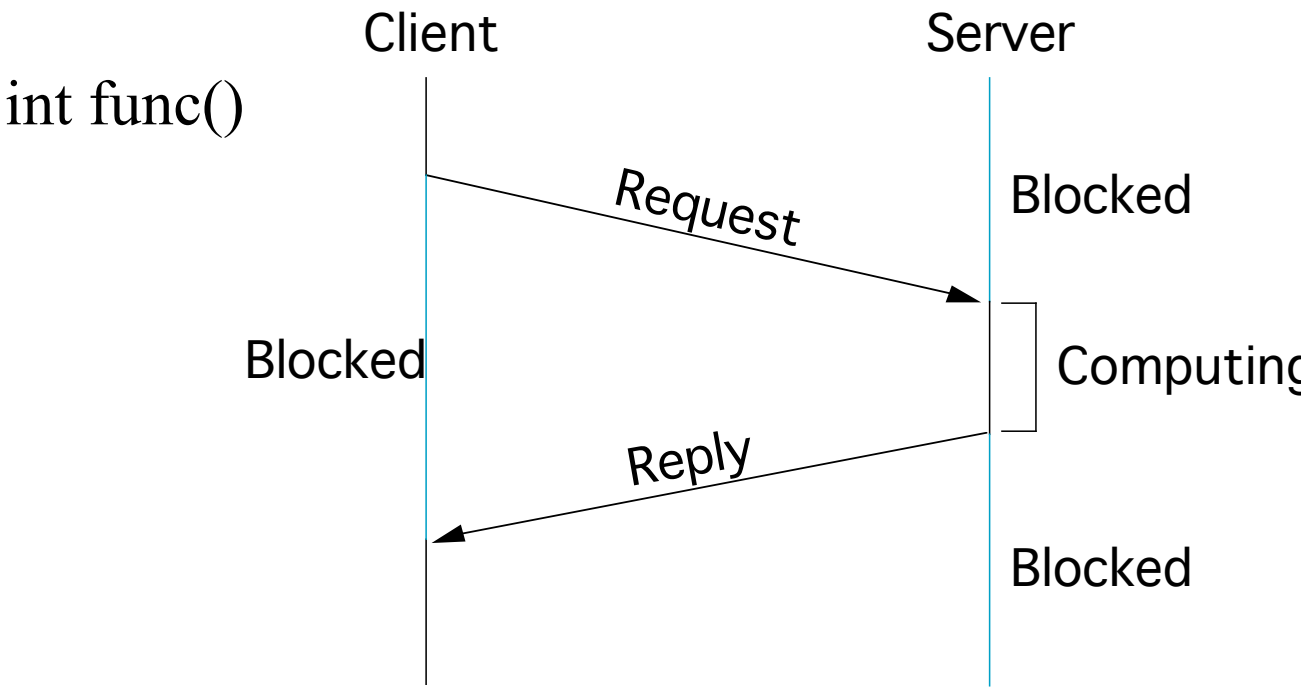


Remote Procedure Call

Outline

- Protocol Stack
- Presentation Formatting
- Readings 5.3
- Why? - Research Approach

RPC Timeline
Paradigm of Distributed Computing



World of Distributed Systems
enabled by Internet

RPC Issues

RPC travels over a network, not a computer backplane

lost, reorder, message size, etc.

need reliable transport, but low overhead

Caller and Callee computers may have significantly different Architectures

why care?

RPC: Transport Issues

- UDP Message in 1 direction followed by UDP Message in other direction
  - Lose
    - Guaranteed messages
    - Order
    - Single Copy
    - Need TCP, but appears expensive and overkill ?

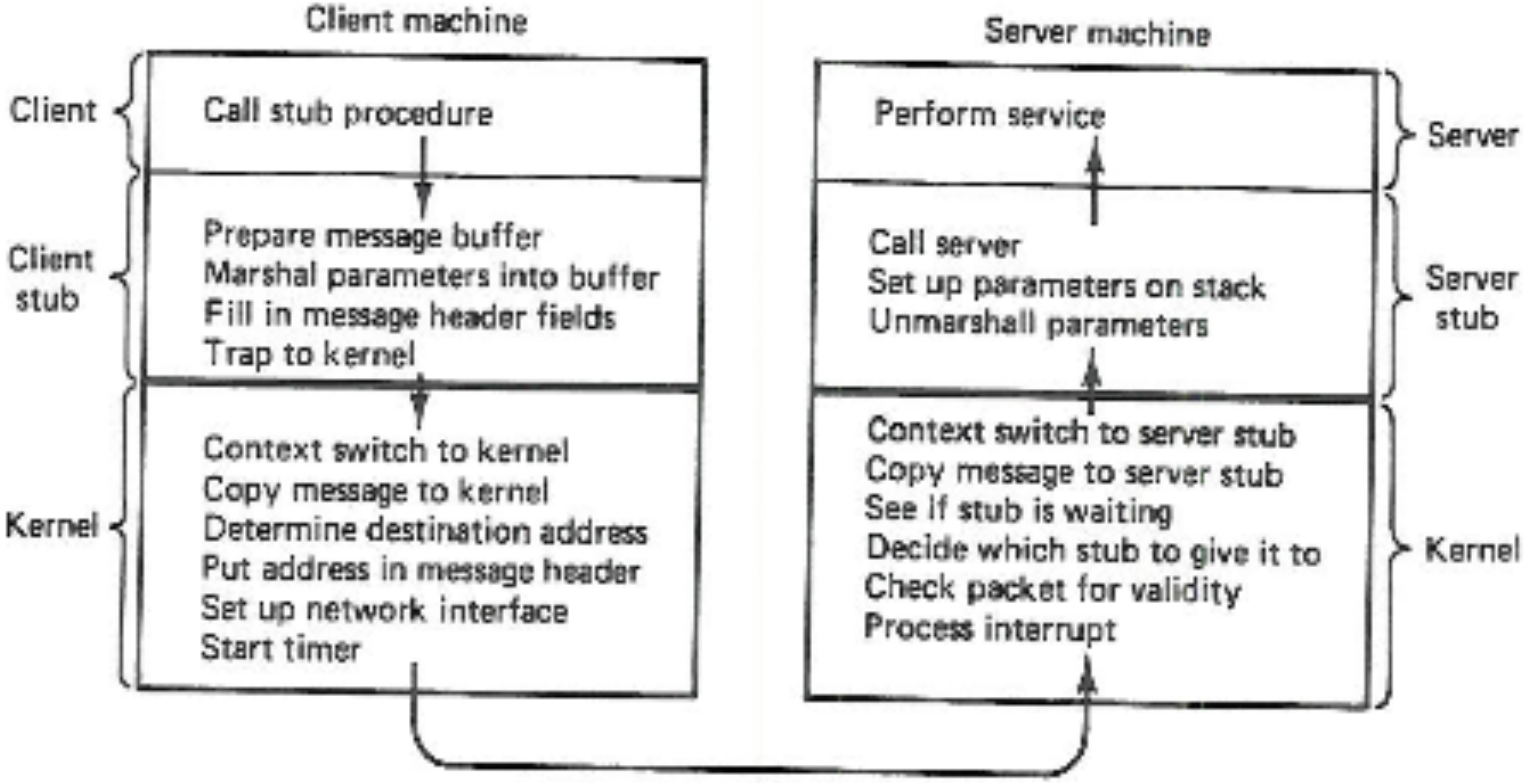


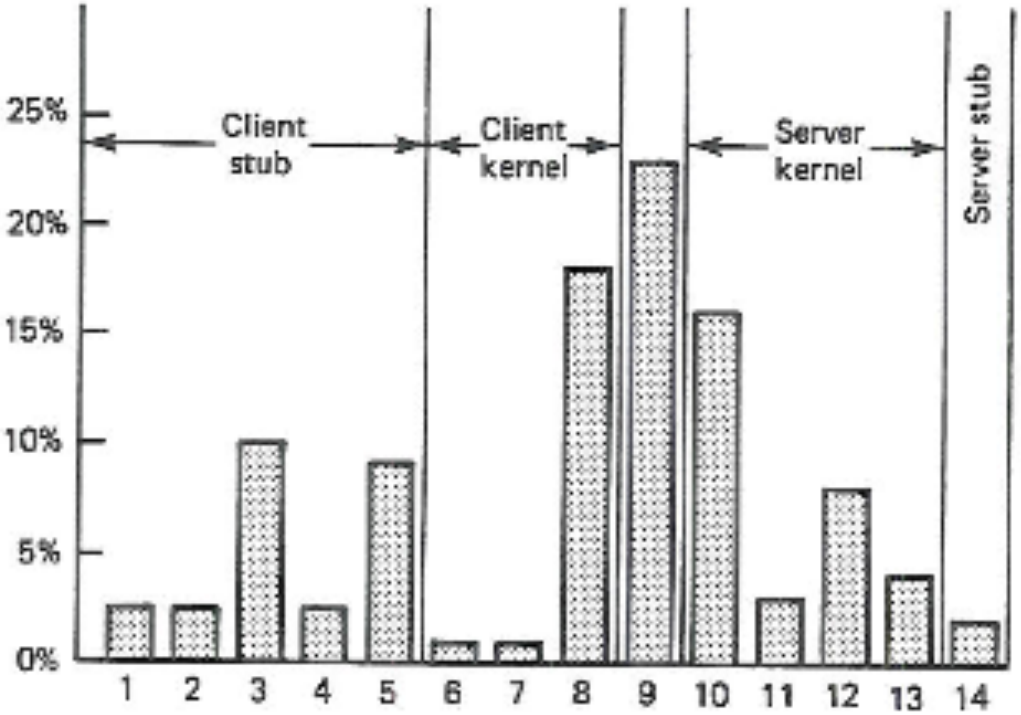
Fig. 10-23. Critical path from client to server.

RPC: Issues

- Client unable to locate server
  - exposes RPC vs normal procedure call
  - server dies, out of date, etc.
  - client needs to handle via exceptions
- Lost Request messages
  - kernel timer
- Lost Reply messages
  - kernel timer, but unsure why message lost
  - sequence #
- idempotent –request that has no side effects and can be executed as often as necessary without any hard state, e.g., read 1st 1k bytes of a file

RPC: Issues

- Server crashes
  - at least once semantics
  - at most once semantics, try once and forget it
  - guarantee nothing
- Client crashes
  - orphan process on server
  - extermination – reboot and kill client process because you have logged process RPC calls
  - reincarnation – reboot, broadcast to server, who kills process as orphan
  - gentle reincarnation – find parent
  - expiration – RPC given time to complete, if not, it dies....



(b)

1440 Byte Array

1. Call stub
2. Get message buffer
3. Marshal parameters
4. Fill in headers
5. Compute UDP checksum
6. Trap to kernel
7. Queue packet for transmission
8. Move packet to controller over the QBus
9. Ethernet transmission time
10. Get packet from controller
11. Interrupt service routine
12. Compute UDP checksum
13. Context switch to user space
14. Server stub code

RPC 14 Steps, time percentage

RPC: Transport Issues

TCP provides expected features

I.e., expected transport support

But Stream not exactly best for single message exchange

- Time to setup and tear down TCP for each message
- Could use ‘push’ or ‘urgent’

RPC - What is needed

Protocol that

Manages RPC messages

Handles network problems, e.g., sequence numbers

Pgm Lang, compiler and stub compiler that supports RPC

Approach:

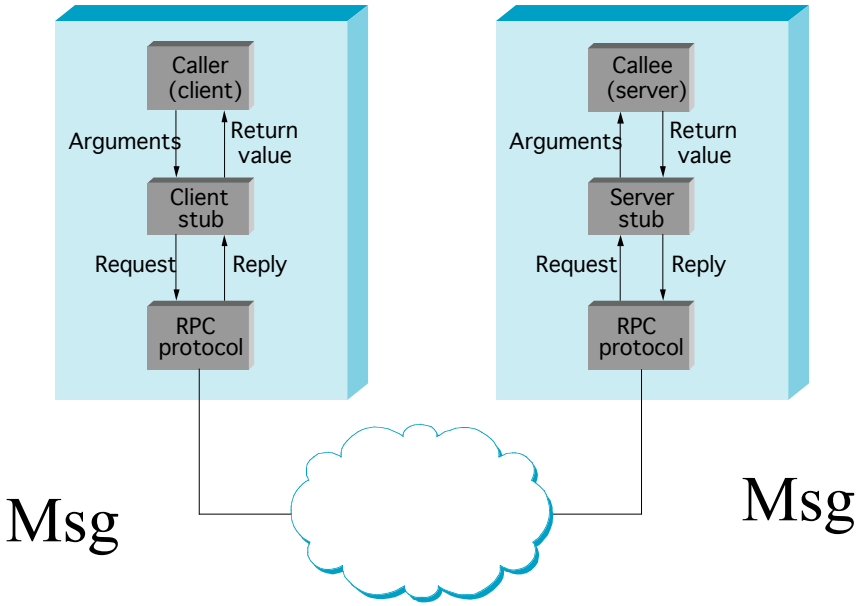
Micro-Protocols for transport of RPC messages

Micro-Protocols: protocols with a single function, RFC 3439 & RFC 3724.

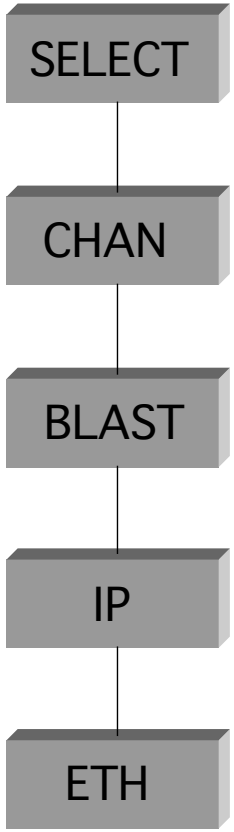
(Micro-protocols are a research approach to networking research)

An Approach: RPC Components

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

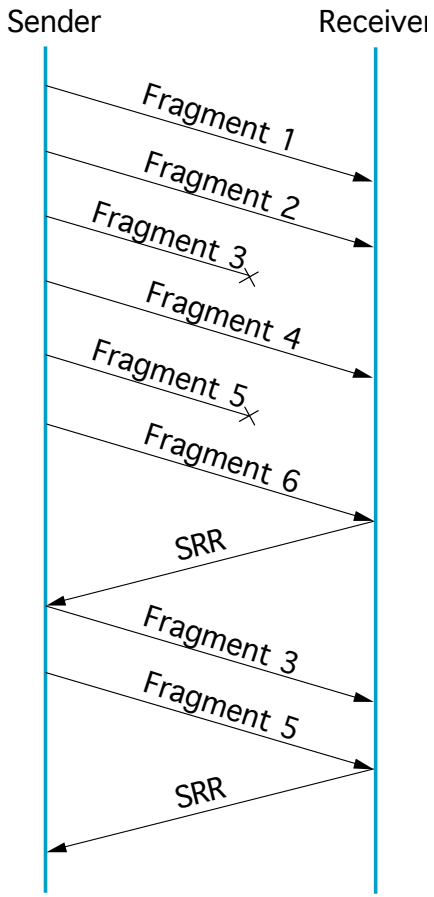


Simple RPC Stack



Bulk Transfer (BLAST)

- Unlike IP, tries to recover from lost fragments (UDP +)
- Strategy
  - selective retransmission
  - supports partial acknowledgements
  - Break large msg into small fragments
  - Use selective retransmission
  - Do not wait for Acks



What triggers SRR

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
  - I.E, will try, but unlike TCP no guarantee…

BLAST Details (cont)

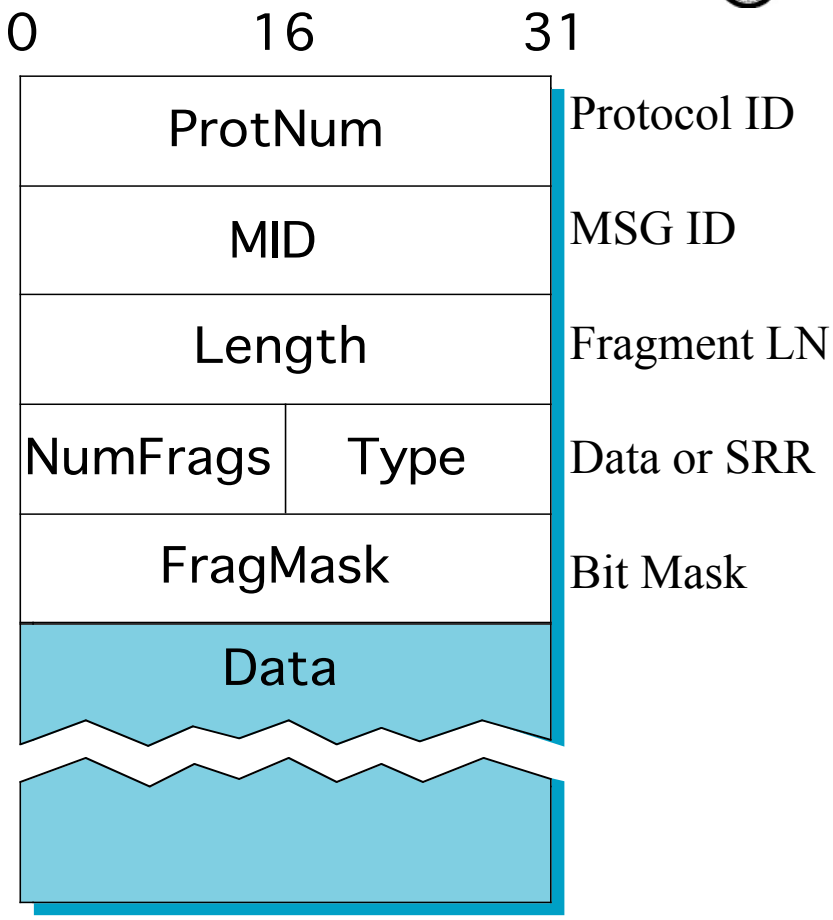
- Receiver:
  - when first fragments arrives, set timer LAST_FRAG
    - Not "last fragment" but last one received
  - when all fragments present, reassemble and pass up
  - four exceptional conditions:
    - if last fragment arrives but message not complete
      - send SRR and set timer RETRY
    - 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 a third time
      - give up and free partial message
  - Persistent, but not infinite

Blast Notes

- 2 events trigger SRR transmission
  - Arrival of last fragment
  - Last_Frag timer
- Timers
  - Done - give up, large value, why?
  - Retry - Performance not an issue
  - Last_Frag sent and lost
- Persistent, but no guarantee
  - Let higher level protocol provide guarantee message delivery

BLAST Header Format

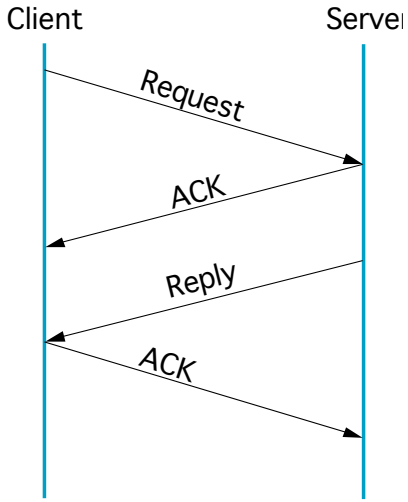
- MSG ID must protect against wrap around
- TYPE = DATA or SRR
- NumFrags indicates number of fragments
- FragMask distinguishes among fragments
  - if Type=DATA, identifies this fragment
  - if Type=SRR, identifies missing fragments



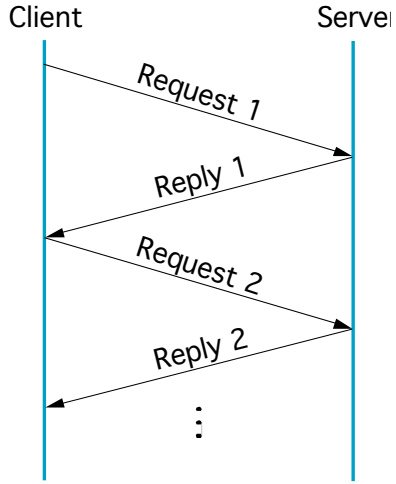
Request/Reply (CHAN)

- Guarantees message delivery
- Synchronizes client with server
- Supports *at-most-once* semantics vs *0 or more*
  - At most 1 msg delivered at most 1 call made

Simple case



Implicit Acks



CHAN Details

- Lost message (request, reply, or ACK)
  - set RETRANSMIT timer
  - use message id (MID) field to distinguish, I.e., buffer single copy
- Slow (long running) server
  - client periodically sends "are you alive" probe, or Ack
  - 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, therefore 64 bits to distinguish Msg: BID/MID

CHAN Timers

- Retransmit on client and server
- Probe on client
- Retransmit affects performance
  - LAN vs WAN
  - Msg size - 1kB to 32kB

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;
```

```
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 vs Asynchronous Protocols

- Asynchronous interface

```
send(Protocol llp, Msg *message)
deliver(Protocol llp, Msg *message)
```

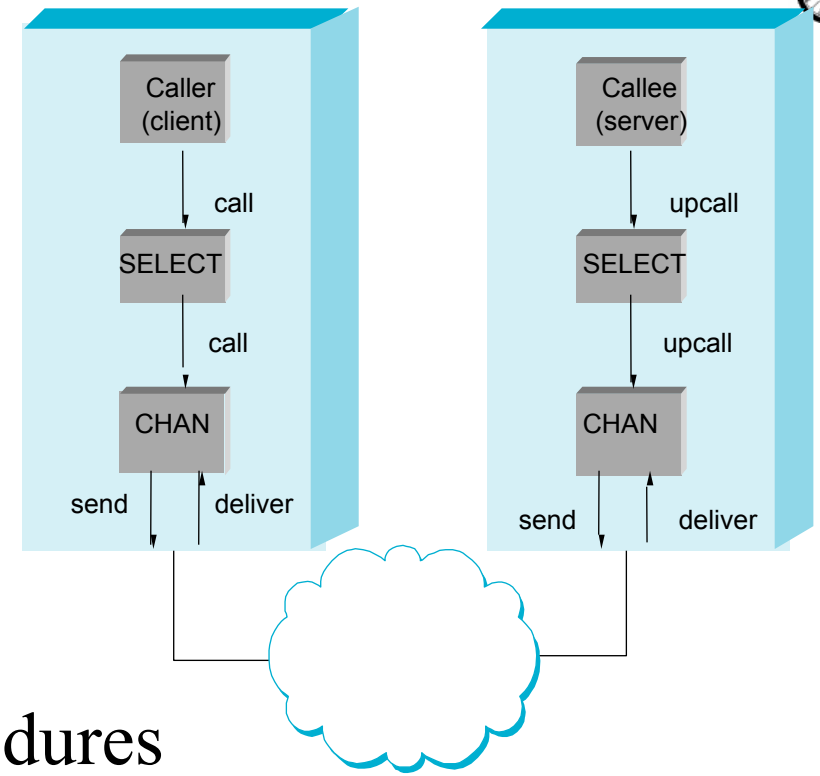
- Synchronous interface

```
call(Protocol llp, Msg *request, Msg *reply)
upcall(Protocol hlp, Msg *request, Msg *reply)
```

- CHAN is a hybrid protocol
  - synchronous from above: **`call`**
  - asynchronous from below: **`deliver`**

Dispatcher (SELECT)

- Dispatch to appropriate procedure
- Synchronous counterpart to UDP
- Implement concurrency (open multiple CHANs)



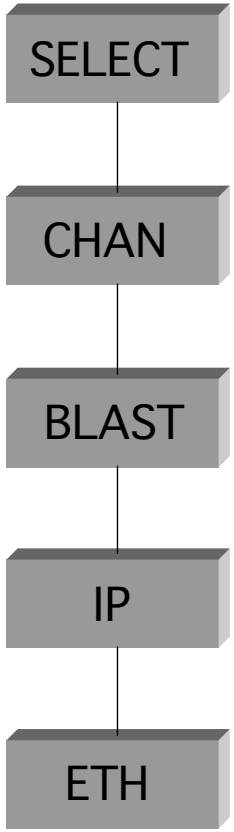
- Address Space for Procedures
  - flat: unique id for each possible procedure
  - hierarchical: program + procedure number

Why Select

Different Procedure address spaces can be chosen by inserting different Selects into protocol graph

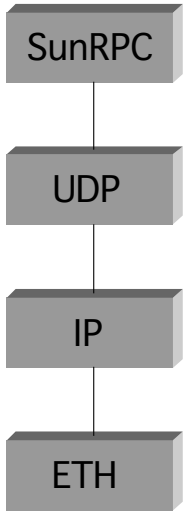
Can be used to support concurrency, I.e., make multiple outstanding calls to same remote procedure (Chan does not allow this)

Simple RPC Stack



SunRPC

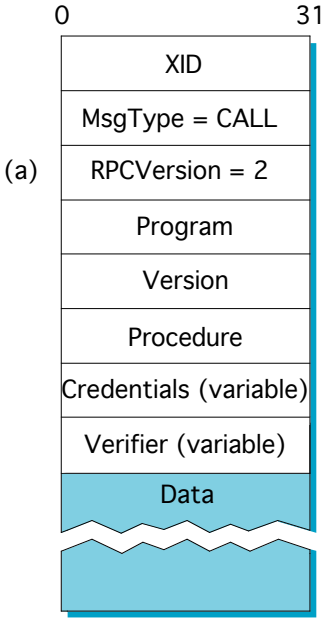
- IP implements BLAST-equivalent
  - except no selective retransmit
- SunRPC implements CHAN-equivalent
  - except not at-most-once

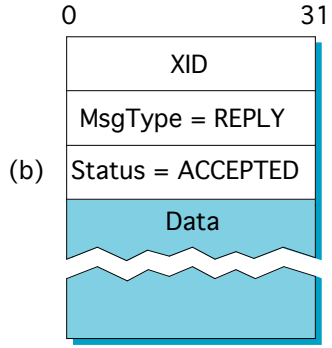


- UDP + SunRPC implement SELECT-equivalent
  - UDP dispatches to program (ports bound to programs)
  - SunRPC dispatches to procedure within program

SunRPC Header Format

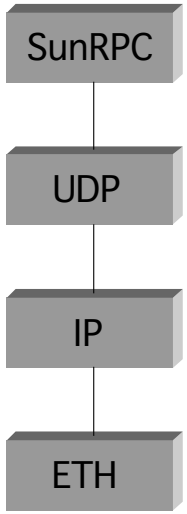
- XID (transaction id) is similar to CHAN's MID
- Server does not remember last XID it serviced
- Problem if client retransmits request while reply is in transit





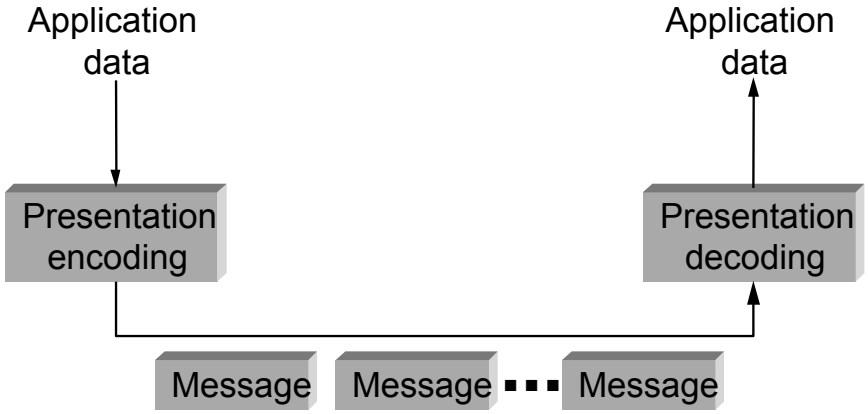
SunRPC

- 2 tier procedure address
  - 32 bit program #
  - 32 bit procedure #
- NFS
  - 00100003 program number
  - getattr – procedure 1
  - setattr – procedure 2
- Messages are sent to UDP well known port



Presentation Formatting

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

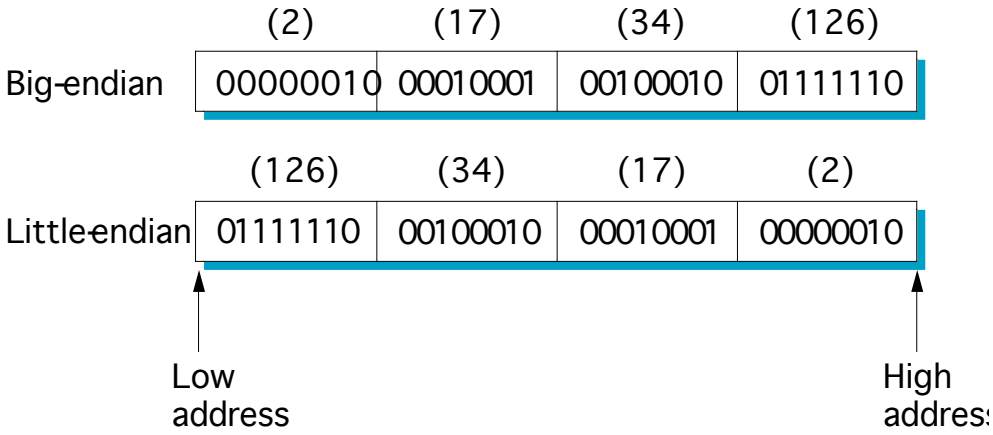


- Data types we consider
  - integers
  - floats
  - strings
  - arrays
  - structs

- Types of data we do not consider
  - images
  - video
  - multimedia documents

Difficulties

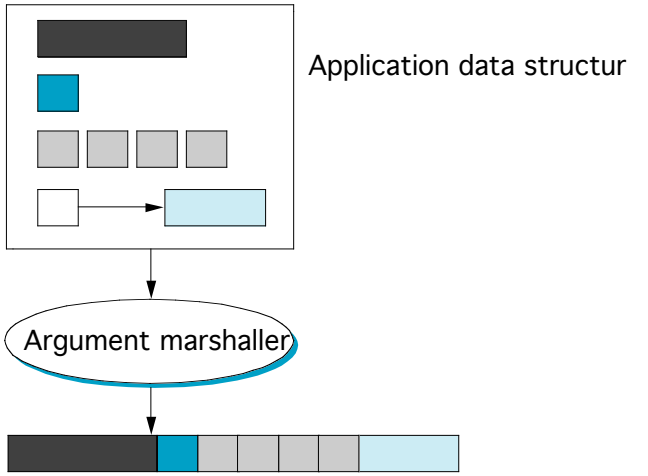
- Representation of base types
  - floating point: IEEE 754 versus non-standard
  - integer: big-endian versus little-endian (e.g., 34,677,374)



- Compiler layout of structures

Taxonomy

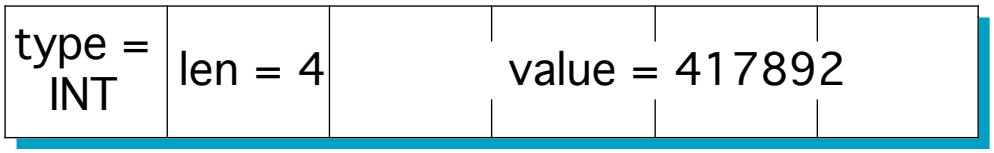
- Data types
  - base types (e.g., ints, floats); must convert
  - flat types (e.g., structures, arrays); must pack
  - complex types (e.g., pointers); must linearize



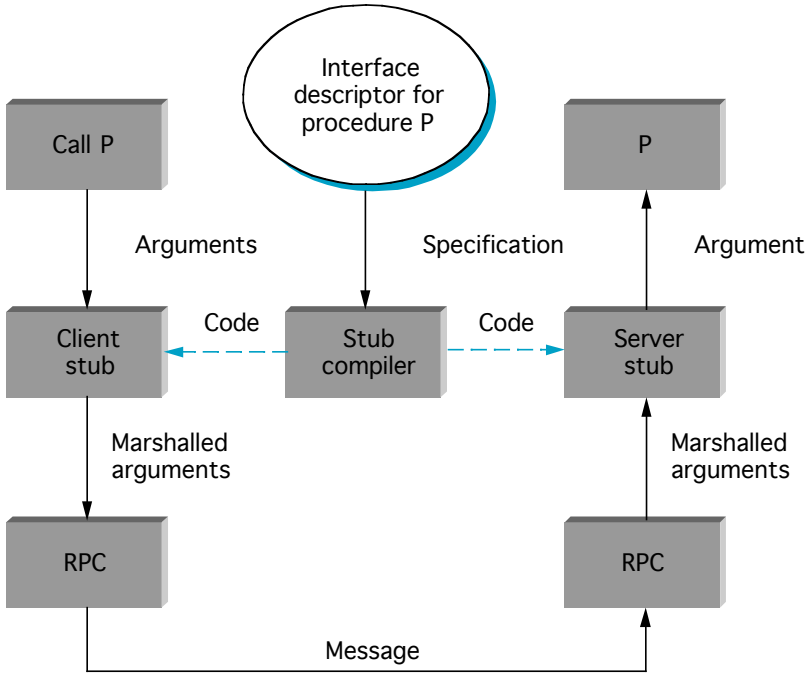
- Conversion Strategy
  - canonical intermediate form
  - receiver-makes-right (an *N* x *N* solution)

Taxonomy (cont)

- Tagged versus untagged data



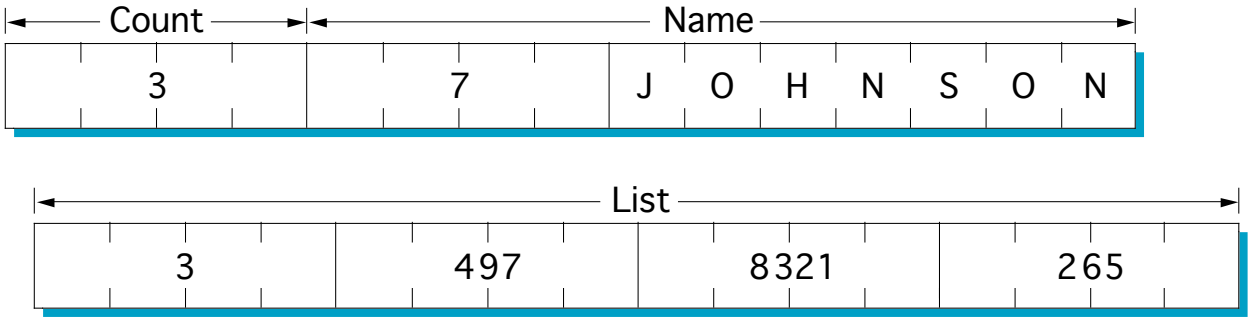
- Stubs
  - compiled
  - interpreted



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

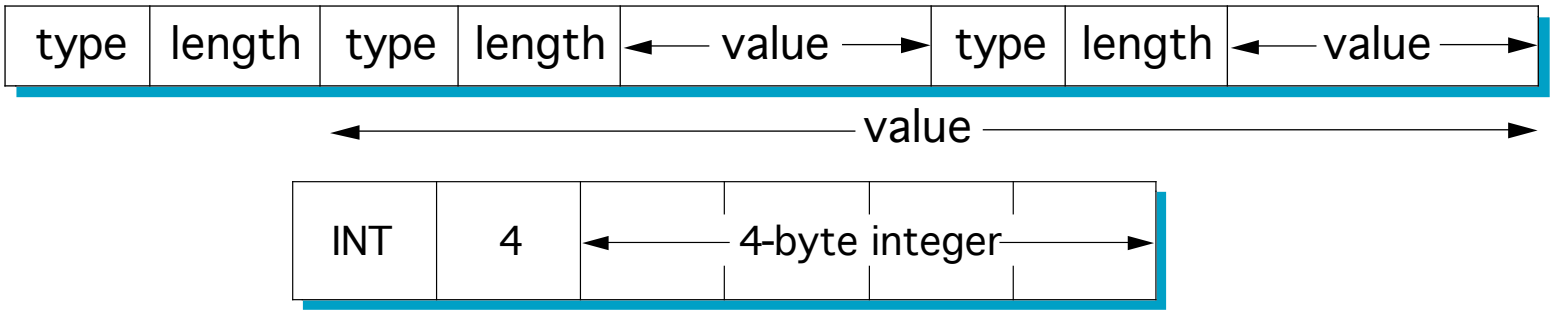

```
#define MAXNAME 256;
#define MAXLIST 100;

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));
}
```



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

(tag, length, value)



The End