CSE/EE 461 Lecture 14
Sockets and RPC

Tom Anderson
tom@cs.washington.edu
Peterson, Chapter 5.3

IP Packet Header Limitations

- Fixed size fields in IPv4 packet header
  - source/destination address (32 bits)
    - limits to ~ 4B unique public addresses; about 800M allocated
    - NATs map multiple hosts to single public address
  - IP ID field (16 bits)
    - limits to 65K fragmented packets at once
    - in practice, fewer than 1% of all packets fragment
  - Type of service (8 bits)
    - unused until recently; needed to express priorities
  - TTL (8 bits)
    - limits maximum Internet path length to 255; typical is 30
  - Length (16 bits)
    - Much larger than most link layer MTU's; path MTU discovery

TCP Packet Header Limitations

- Fixed size fields in TCP packet header
  - seq #/ack # -- 32 bits (can't wrap within MSL)
    - T1 ~ 6.4 hours; OC-192 ~ 3.5 seconds
  - source/destination port # -- 16 bits
    - limits # of connections between two machines (NATs)
    - ok to give each machine multiple IP addresses
  - header length
    - limits # of options
  - receive window size -- 16 bits (64KB)
    - rate = window size / delay
    - Ex: 100ms delay => rate ~ 5Mb/sec

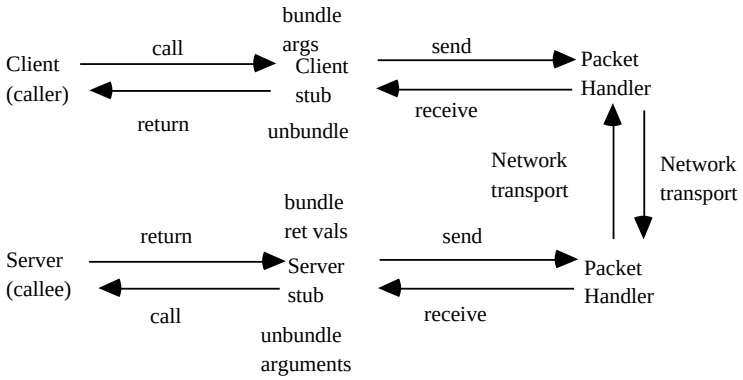
Sockets

- OS abstraction for a communication endpoint
  - Layer on top of TCP, UDP, local pipes
- server (passive open)
  - bind -- socket to specific local port (e.g., 80)
  - listen -- wait for client to connect
  - read/write or send/receive data
- client (active open)
  - connect – to specific remote port
    - TCP: send a SYN
    - UDP: return
  - read/write or send/receive data

Remote Procedure Call

- Abstraction: call a procedure on remote machine
  - client calls: Get("foo")
  - server invoked as: Get("foo")
- Implementation
  - request-response message passing
  - "stub" routines provide glue
- Design pattern used in many different contexts
  - HTTP; NFS (remote file service); DNS

Remote Procedure Call



RPC Stubs

On client:

```
get(filename) {
    build message
    send message
    wait for response
    unpack reply
    return result
}
```

On server:

```
loop {
    wait for command
    decode command
    unpack arguments
    call procedure (get)
    build reply message
    send reply
}
```

Procedure Call vs. RPC

- Parameters
  - request message
- Return value
  - reply message
- Name of the procedure
  - passed in request message
- Return address
  - source port

Two Examples

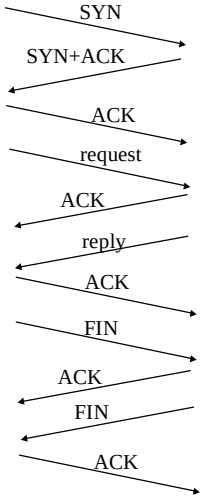
- HTTP search engine query
  - URL encodes parameters as text string
  - On server, page causes search program to be run
  - Reply encoded as web page
- NFS (Network File system)
  - UDP messages
  - Request packet contains command (read/write/stat), name of file, offset, data (if write), …
  - Reply contains data (if read), return value

Object Oriented RPC

- Two approaches:
  - Every object has local stub object (Jini)
    - stub object translates local calls into RPCs
  - Every object pointer is globally valid
    - pointer = machine # + address on machine
    - compiler translates pointer dereference into RPC
- Function shipping vs. data shipping
  - ship request to the data or data to the request?

RPC on TCP

- How do we reduce the # of messages?
  - Delayed ack: wait for 200ms for reply or another pkt arrival
  - UDP: reply serves as ack
    - RPC system provides retries, duplicate supression, etc.
    - Typically, no congestion control



Reducing TCP packets for RPCs

- For repeated connections between the same pair of hosts
  - Persistent HTTP (1.1)
    - Keep connection open after web request, in case there's more
  - T/TCP -- "transactional" TCP
    - Use handshake to init seq #s, recover from crash
    - after init, request/reply = SYN+data+FIN

RPC Failure Models

- How many times is an RPC done?
  - Exactly once?
    - Server crashes before request arrives
    - server crashes after ack, but before reply
    - server crashes after reply, but reply dropped
  - At most once?
    - If server crashes, can't know if request was done
  - At least once?
    - Keep retrying across crashes, but may be done multiple times
    - Example: NFS idempotent ops (ex: read/write file block)

Exactly Once RPC

- Example: buy something over Ebay, Amazon
  - want exactly one widget, book, 100 shares of kozmo
- Want RPC to be
  - done exactly once
  - done completely or not at all
  - done atomically with respect to other requests
  - once done, stays done (independent of later crashes)
- Analogous to distributed database transactions

Exactly Once RPC

- Can implement using disk on both ends
  - client writes “about to make request” to disk
    - keep retrying until there is a reply (done/abort)
  - client sends request
  - server gets request; computes result
  - server writes “about to reply” to disk
    - along with contents of reply message
  - server sends reply
  - client writes “got response” to disk
    - to remove request; if crash, don’t want to retry

General’s Paradox

Can we use messages and retries to synchronize two machines so they are guaranteed to do some operation at the same time?

- No.