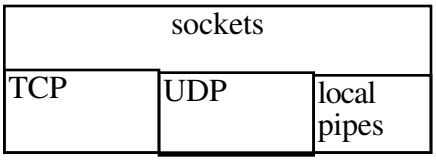
Lecture 7: Remote Procedure Call

7.0 Main Points

Sockets API
Send/receive messaging
Remote procedure call

7.1 Sockets

A socket is an abstract OS entity representing a communication endpoint.



API (application programming interface)

server (passive open):
- bind -- bind local end of socket to a specific port (e.g., 80 for web server)
- listen -- process is ready for connections
- accept -- return next connection

client (active open):
- connect -- pick a random port & connect to a specific port at far end
- TCP -- send SYN, wait for SYN/ACK
- UDP -- return immediately!

both:
- socket -- create a socket (called before bind and connect)
- write -- put bytes onto socket (after connect/accept)
- read -- take bytes off socket (after connect/accept)
- send/receive -- same as read/write, but in terms of messages
- close -- all done reading and writing

Note: once open, socket can be written/read by any number of processes/threads; usually only one thread writing or reading at each end.

	1 client	> 1 client
1 server thread	easy	each client waits turn (queue of SYN packets) or event loop -- server accepts both new connections and requests on old connections
> 1 server thread	N/A	1 per client connection or 1 per client request

		(if parallel requests on a connection) or >1 per client request (e.g., to use parallel CPUs to service request)

7.2 Send/Receive Message Programming Model

send(message, socket) -- put message on network

receive(buffer, socket) -- read arriving message into buffer

Send and receive are **atomic**

never get portion of message (allor nothing)
two receiver threads don't get the same message

Message styles

1 way -- messages flow in one direction (UNIX pipes)
2 way -- request/response (remote procedure call)

1-way communication: producer/consumer

```
Producer:
  int msg1[1000]

  while (1)
     prepare message  // make coke
     send(msg1, socket)

Consumer:
  int msg2[1000]

  while (1)
     receive(msg2, socket)
     process message  // drink coke
```

No need for producer/consumer to keep track of space in mailbox -- handled by send/receive.

What about two-way communication?

Request/response. For instance, "read a file" stored on a remote machine.

Also called: client-server. Client = requester; server = responder. Server provides "service" (file storage) to the client

Request/response:

```
Client: (requesting the file)
   char response[1000];

   send("read rutabaga", socket);
   receive(response, socket);

Server:
   char command[1000], answer[1000];

   receive(command, socket);
   decode command
   read file into answer
   send(answer, socket);
```

Server has to decode command, just as OS has to decode message to find socket (port #)

What if file is too big to fit in response buffer? Fragment!

7.3 Remote procedure call.

Call a procedure on a remote machine.

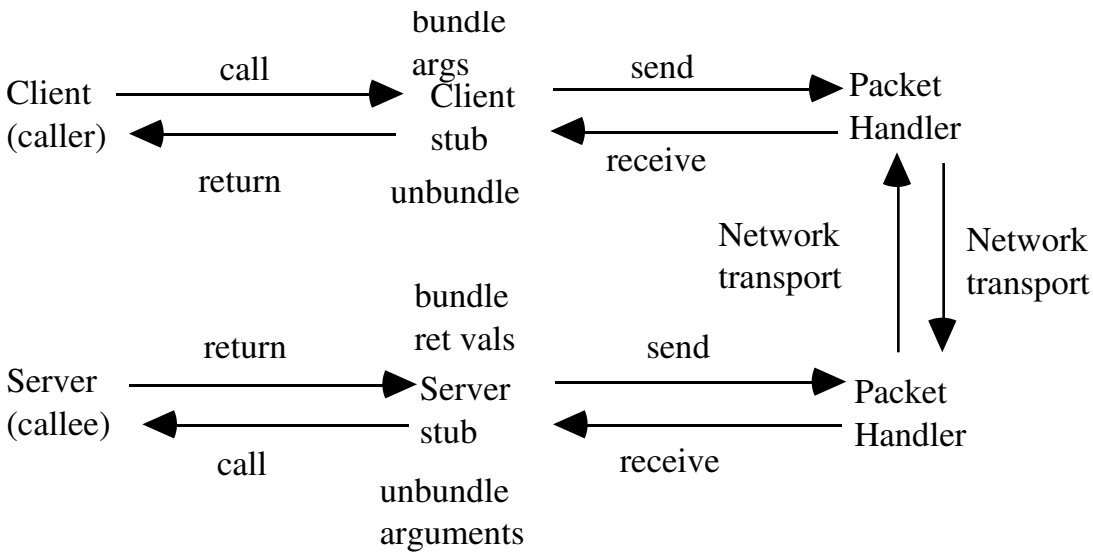
Client calls:

```
remoteFileSys->Read("rutabaga")
```

Translated into call on server:

```
fileSys->Read("rutabaga")
```

Implementation:
request-response message passing
"stub" provides glue on client/server




```
Client stub:
  build message
  send message
  wait for response
  unpack reply
  return result

Server stub:
Create N threads to wait for work to do
  loop:
    wait for command
    decode and unpack request parameters
    call procedure
    build reply message with results
    send reply
```

Comparison between RPC and procedure call

What's equivalent?

Parameters -- request message

Result -- reply message

Name of procedure -- passed in request message

return address -- source port