NOTES

Network Programming

Shawn Ostermann
Associate Professor
Ohio University

A Little About Me

- ...

A Little About You

- Where are you from?
- Why are you here?
- What is your background in
  - Networking?
  - Programming
  - Network programming

NOTES

Theoretical Schedule

9:00-10:30	Background Review
	Application Interface
	Client Applications
10:30-10:45	*Break*
10:45-11:30	Lab Exercise
	Build a simple client
11:30-12:00	Debugging Strategies
12:00-1:00	*Lunch*
1:00-1:30	Lab Exercise
	Debugging Strategies
1:30-2:15	Server Applications
2:15-2:45	Lab Exercise
	Build a simple server
2:45-3:00	*Break*
3:00-3:45	Higher-level Client/Server
3:45-4:30	Lab Exercise
	build a simple RPC application
4:30-5:00	Misc
	Hints and Tips about good networking projects
	Wrapup

Separation of Functionality

- Network Protocol - IP
  - Delivers packets of data from one host to another
  - Each packet is individually addressed
  - Packets traverse the network separately
  - Best-effort delivery
- Transport Protocol UDP/TCP
  - Delivers data from one application to another
  - Handles identification of applications
  - May handle reliability

NOTES

The IP Internet Address Scheme - IPv4

- Hosts assigned a 32-bit, hardware independent Internet Protocol (IP) address
- Network divided into a prefix portion and a suffix portion
  - Prefix identifies the network
    - Prefixes assigned by central authority
  - Suffix identifies which computer on the network
    - Assigned locally

NOTES

Dotted Decimal Notation

- Syntactic form for expressing 32-bit address
- Represents each byte in decimal separated by dots
- Example
  - A 32-bit number in binary

    `10000100 11101011 00000001 00000010`

  - The same 32-bit number expressed in dotted decimal notation

    `132.235.1.2`

How to divide Prefix and Suffix The "old way", IP Address Classes

- IPv4 addresses are 32 bits wide
- Three principle classes

	01	8	16	24	31
Class A	0	netid	hostid		
Class B	10	netid		hostid	
Class C	110	netid			hostid

- Lesser-used classes

	01	8	16	24	31
Class D	1110	IP Multicast			
Class E	11110	reserved			

NOTES

How to divide Prefix and Suffix The "new way", Address Masks

- The address mask tells which bits correspond to the prefix and which to the suffix
  - In *subnetting*, which bits are which is arbitrary
  - Identified by a mask, as in `255.255.240.0`
- In *supernetting* (CIDR), the bits must be contiguous and start at "the left"
  - Identified with the syntax `132.235.3.1/24`

User Datagram Protocol

- Provides roughly the same service as IP
  - But extended to the user level (applications)
- Adds identification of applications
- Adds optional data integrity check
  - Checksum covering the headers and data

NOTES

Identifying Applications with UDP/TCP Ports

- Both UDP and TCP use the notion of a *port* to identify applications
  - A 16-bit field
- We use the lower addresses to identify *well-known* services
  - TELNET,Email,WWW, etc
  - Generally below 1024 or 2048
  - Fixed by Internet Assigned Numbers Authority
- Higher addresses are normally assigned at random to clients
  - Called dynamically assigned ports

Program Interface To Port Assignment

- Port numbers should not be hard-coded into programs as literal constants
  - Service names are better because they can be redefined for testing or to avoid conflicts
- Most systems lookup routines for mapping names to numbers

NOTES

Example Service Mapping Database (/etc/services in UNIX)

```
echo          7/tcp
echo          7/udp
ftp          21/tcp
telnet       23/tcp
smtp         25/tcp
time         37/tcp
time         37/udp
nameserver   53/tcp
nameserver   53/udp
myserver     2001/udp
```


Transmission Control Protocol TCP

- Used by “most” applications in TCP/IP
- Transfers a *stream* of data
- Provides reliability
- Full duplex connections
- Flow control, Congestion Control
- Uses IP for datagram transmission

Transmission Control Protocol Details

- Sender generates a stream of bytes
  - Granularity isn't important
- TCP Divides stream into small *segments* for transmission
  - Each segment goes in an IP datagram
- Segments are conceptually numbered
  - Ensures in-order delivery to application
  - Allows the sender and receiver to determine which segments did not arrive
  - Lost segments can be retransmitted

NOTES

Theoretical Schedule

9:00-10:30	Background Review
	Application Interface
	Client Applications
10:30-10:45	*Break*
10:45-11:30	Lab Exercise
	Build a simple client
11:30-12:00	Debugging Strategies
12:00-1:00	*Lunch*
1:00-1:30	Lab Exercise
	Debugging Strategies
1:30-2:15	Server Applications
2:15-2:45	Lab Exercise
	Build a simple server
2:45-3:00	*Break*
3:00-3:45	Higher-level Client/Server
3:45-4:30	Lab Exercise
	build a simple RPC application
4:30-5:00	Misc
	Hints and Tips about good networking projects
	Wrapup

NOTES

Program Interface To Protocol Software

- An application program uses procedure calls to interact with protocol software
- The specification of those procedures defines the *Application Program Interface* (API)
- Most communication protocols do not define the precise API that programs use to access them
  - API depends on the underlying operating system

Two Prominent APIs

- Socket Interface
  - From BSD UNIX
  - A *de facto* standard
  - Follows normal UNIX I/O conventions
    - open,read/write,close
  - Also provided by other systems
    - Windows, Java, ...
- Transport Layer Interface (TLI)
  - From AT&T System V UNIX

Socket I/O

- Program calls *socket* to create a socket
  - Returns an integer descriptor
- Program calls *recv* or *send* to read/write data
  - Socket descriptor is an argument in each call
  - read and write work too, but are not recommended
- Program calls *close* to terminate the communication
  - Descriptor is an argument to the call

NOTES

The Socket Interface

- Sockets are flexible; they can be used:
  - For connection-oriented communication (TCP)
  - For connectionless communication (UDP)
  - With TCP/IP protocols (e.g., TCP, UDP)
  - With other protocols
    - Unix *pipes*
    - Appletalk, etc...

The Socket Functions

Name	Meaning
socket	Create socket & descriptor
connect	Connect to a remote peer
close	Terminate all communication
bind	Bind a local address to socket
listen	Place a socket in passive mode
accept	Accept incoming connection (server)
recv	Receive the next incoming message
recvmsg	Receive the next incoming message
recvfrom	Receive the next incoming message & get source addr
send	Send outgoing message
sendmsg	Send outgoing message
sendto	Send outgoing message to a specified address
shutdown	Terminate communication in one or both directions
getpeername	Obtain remote machine's address
getsockopt	Obtain current options for a socket
setsockopt	Change options for a socket

NOTES

Theoretical Schedule

9:00-10:30	Background Review Application Interface Client Applications
10:30-10:45	*Break*
10:45-11:30	Lab Exercise Build a simple client
11:30-12:00	Debugging Strategies
12:00-1:00	*Lunch*
1:00-1:30	Lab Exercise Debugging Strategies
1:30-2:15	Server Applications
2:15-2:45	Lab Exercise Build a simple server
2:45-3:00	*Break*
3:00-3:45	Higher-level Client/Server
3:45-4:30	Lab Exercise build a simple RPC application
4:30-5:00	Misc Hints and Tips about good networking projects Wrapup

NOTES

Writing Client Software

- Client applications are generally much easier to write than server applications
  - This will be more clear when we discuss servers later today
- Many clients have a very simple algorithm
  - Find a server
  - Send it a question
  - Read the answer

Setting up a Client Socket

- To set up the client socket, we use the `socket` system call

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

- Argument details

domain	the *protocol family*, we'll always use `PF_INET`
type	the type of communication, we'll use either `SOCK_STREAM` (TCP) or `SOCK_DGRAM` (UDP)
protocol	disambiguates the *type*, we'll use either `IPPROTO_TCP` (6) or `IPPROTO_UDP` (17)

NOTES

Clients
Our First Problem

- Just as we're getting started, we need to decide whether to use UDP or TCP
  - TCP - Connection-oriented
    - Send and receive a stream of bytes
  - UDP - Connectionless
    - Send and receive a sequence of messages (datagrams)
- Fortunately, if we're only writing the client, then we should use the protocol that the server uses
  - Sometimes, the server can use either one
- Remember that with UDP, reliability is generally the responsibility of the client application

Client Socket Details

- Here's a full example of a client socket
  - For TCP protocols:

```
int s;
s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
```

  - For UDP protocols:

```
int s;
s = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

NOTES

Specifying the Server Address

- Whether we're using TCP or UDP, we need the server's name (or IP address) and port number
- We specify the address using the `connect()` system call

```
int connect(int s, const struct sockaddr *name,
            struct_t namelen);
```

- Argument details

s	the file descriptor that we got back from `socket()`
name	a pointer to a structure giving the IP address and port
namelen	size of the structure

Poor `Connect()` Example

- To give you the idea, here's a bad example:

```
struct sockaddr_in sin;
   /* fields: sin_family, sin_port, sin_addr */

memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_port = htons(13);
sin.sin_addr.s_addr = inet_addr("132.235.1.7");

connect(s, (struct sockaddr *)&sin, sizeof(sin));
```

What's Wrong with that Example

- Hardcoding the port number as a number is generally a bad idea
  - Better to look it up
  - Always want to let the user override the default
- Hardcoding the IP address is an even worse idea
  - Probably want the user to be able to specify a name **or** an IP address

NOTES

A Better Connect Example

- Here's a better, longer section of code that solves those problems

```
char *host, *service;
if ((pse = getservbyname(service, protocol)))
    sin.sin_port = pse->s_port;
else if ((sin.sin_port =
          htons((u_short)atoi(service))) == 0 )
    Oops("can't find port for '%s'", service);
if ((phe = gethostbyname(host)))
    bcopy(phe->h_addr, &sin.sin_addr, phe->h_length);
else if ((sin.sin_addr.s_addr =
          inet_addr(host)) == -1 )
    Oops("can't get '%s' host entry", host);
```

NOTES

Specifying the Local Address

- Normally, a client doesn't need to be operating on a particular port
  - Just let the Operating System pick a random port for you
  - It will do that if you haven't given it a local address before calling `connect()`
- Otherwise, you can use the `bind()` socket call
  - We'll talk about that when we discuss servers

Writing to a Socket

- For sending data from a socket, we use `send()`:

```
int send(int s, const void *buf, int len, int flags);
```

- Argument details

s	your socket fd
buf	buffer holding the data to write
len	number of bytes of data to write
flags	usually 0, check the man pages

- Returns the number of bytes written, -1 for error

Reading from a Socket

- For reading data from a socket, we use `recv()`

```
int recv(int s, void *buf, int len, int flags);
```

- Argument details

s	your socket fd
buf	buffer holding the data to write
len	size of the buffer (**MAXIMUM** data to receive)
flags	usually 0, check the man pages

- Returns the number of bytes read, -1 for error, and 0 for EOF

NOTES

Programming Hints
Sending Data with UDP

- Each call to `send()` generates and sends one UDP datagram
  - UDP datagrams are limited in size by the IP protocol to a little less than 64k bytes
  - Local Operating System may limit them further
- Recall that UDP is not reliable, so compensating for "unreliability" is the problem of the application
  - Loss
  - Duplication
  - Reordering
  - (Corruption)

Programming Hints
Reading Data with UDP

- Each call to `recv()` copies the data from a single UDP datagram into the buffer
  - Beware when the datagram is larger than the buffer

NOTES

Programming Hints
Sending Data with TCP

- Remember that TCP is a reliable stream protocol
  - We don't need to worry about reliability
- The Operating System takes care of breaking the data you send into TCP segments
  - Normally, each call to `send()` causes a TCP *PUSH*
- Some Application Protocols require that the client send an EOF before the server will answer
  - Accomplished with the `shutdown()` routine

NOTES

Shutting down a socket

- `close()` shuts down the socket for both reading and writing
- To just do one or the other, use `shutdown()`

```
int shutdown(int s, int how);
```

- Argument details

s	your socket FD
how==0	shut down reads (0==stdin)
how==1	shut down writes (1==stdout), sends EOF (FIN) with TCP
how==2	same as `close()`

Programming Hints
Reading Data with TCP

- There are no record boundaries with TCP, so the granularity of the `send()` calls doesn't correspond to the blocks of data that you'll get with *recv()*.
- Normally `recv()` will *block* if there's **NO** data, but otherwise it'll give you any amount up to the buffer size
  - Never depend on *recv* returning exactly the same sizes units as the data was written with
  - Beware of:

```
recv(s, &integer, 4);
```

Client Details
Connected and Unconnected UDP Sockets

- TCP sockets must be *connected*
- UDP sockets can be:
  - *Connected*: remote IP/port is set with `connect()`
  - *Unconnected*: remote endpoint address is not specified
    - Use the `sendto()` and `recvfrom()` variants

NOTES

Good Client Practice

- Allowing a dotted decimal address in place of a DNS name makes it easier to debug problems
- Clients should generally have a default port built in
- Always provide the user with the ability to specify an alternate port
  - Very useful in TELNET, for example

Client Programming Hints

- Read the manual pages for include files
- Remember that these are file descriptors, not streams!
- Always compile using gcc with `-Wall -Werror -O2`
- Comer has a nice book on client/server programming (in several Operating System versions) with useful libraries
  - Libraries are nice, but raw sockets makes a better first assignment
- Stevens has more documentation on individual socket calls
- In Unix, `read()` == `recv()`, `write()` == `send()`

NOTES

When Clients Become More Difficult

- The need for asynchronous reads and writes makes some kinds of clients difficult
  - TELNET
  - Asynchrony solutions
    - Threads
    - Multiple processes
    - Non-blocking `send()`s and `recv()`s
- Server location is a hard problem
  - Not addressed by TCP/IP
  - Needs a higher-level solution

NOTES

- Making UDP reliable is a challenging exercise if done correctly

General Reliability Problems

- If you're using UDP as the transport protocol, you (usually) need to implement your own reliability
  - When do you **NOT**?
- For the client side, it's often possible to retransmit the question if you don't receive the answer
  - Still need to estimate the RTT
  - Do some sort of backoff
  - What are the semantics for the server, then?

NOTES

Client Lab Assignments

- Write the following simple clients
  - Connect to the UDP *daytime* (13) port on a machine
    - Read the datagram that comes back and print it out (it's ASCII)
  - Connect to the TCP *finger* (79) port on a machine
    - Send a "username CRLF" (or just CRLF)
    - Read the text that comes back and print it out
- "extra credit"
  - Write a quick telnet-like client that can handle asynchronous input/output by using two processes (or threads, interrupts, or nonblocking I/O)
  - If you know the HTTP protocol, write a "one-shot" WWW client that simply displays the web server's response on the screen

Look at Ostermann's Solutions

- Available separately!!!!

Good Programming Projects for your Classes

- TFTP client/server
  - Forces the students to implement their own reliability
- WWW client
  - Not as difficult as it sounds if you stick to text
- Telnet-like client
  - More difficult because of the asynchronous data flow

NOTES

Theoretical Schedule

9:00-10:30	Background Review
	Application Interface
	Client Applications
10:30-10:45	*Break*
10:45-11:30	Lab Exercise
	Build a simple client
11:30-12:00	Debugging Strategies
12:00-1:00	*Lunch*
1:00-1:30	Lab Exercise
	Debugging Strategies
1:30-2:15	Server Applications
2:15-2:45	Lab Exercise
	Build a simple server
2:45-3:00	*Break*
3:00-3:45	Higher-level Client/Server
3:45-4:30	Lab Exercise
	build a simple RPC application
4:30-5:00	Misc
	Hints and Tips about good networking projects
	Wrapup

NOTES

Obvious Solution 1
Don't Make the Mistakes in the First Place!

- Always use a Makefile
  - Prevents partial recompilation errors
- If using gcc, always use the args:
  - -*Wall*
    - Anything suspicious generates a warning
  - -*Werror*
    - All warnings are fatal
  - -*O2*
    - Optimize a little more, catches more errors
    - Helps find uninitialized variables

Obvious Solution 2
Listen to the Operating System

- All of the socket library routines have return values
  - Check them all
  - No really, check them all!

Watch the Packets

- Often, you can narrow the problem down quickly with a packet capture
  - Using tcpdump/snoop/etc...
- Be sure to watch the packets from a machine that they must go through
  - Beware of snooping on the same computer as the sender/receiver
- Know the packet capture program's arguments to make it easier to find things

NOTES

Use the Debugger

- Gdb is wonderful

NOTES

System Call Traces

- Finds errors from system calls that you might have forgotten to check!
- `truss` in Solaris
- `strace` in some other Operating Systems

Other Clients

- Verify the server by using TELNET
- Most TELNET clients let you specify the port to connect to
- See if the server is doing what you think it is

Other Servers

- I use a quick Stevens program called `socket` to simulate quick servers
- Useful for checking to see what the client is sending

NOTES

Use Standard Services

- In most versions of Unix, you can enable helpful services:
  - daytime
  - discard
  - echo
  - chargen
- These are generally disabled by default for security reasons
  - See the file `/etc/inetd.conf`
    - Edit the file and then send a `HUP` to the `inetd` server
  - Be sure to re-disable them when finished!

NOTES

Theoretical Schedule

9:00-10:30	Background Review
	Application Interface
	Client Applications
10:30-10:45	*Break*
10:45-11:30	Lab Exercise
	Build a simple client
11:30-12:00	Debugging Strategies
12:00-1:00	*Lunch*
1:00-1:30	Lab Exercise
	Debugging Strategies
1:30-2:15	Server Applications
2:15-2:45	Lab Exercise
	Build a simple server
2:45-3:00	*Break*
3:00-3:45	Higher-level Client/Server
3:45-4:30	Lab Exercise
	build a simple RPC application
4:30-5:00	Misc
	Hints and Tips about good networking projects
	Wrapup

Server Software Overview

- Servers begin execution before the clients (usually)
- Servers receive questions and answer them
- Servers may themselves be clients of other servers
  - DNS
  - File systems
  - time of day
  - ...

Setting up a Server Socket

- Server sockets are created just like client sockets:

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

- Argument details

domain	the *protocol family*, we'll always use `PF_INET`
type	the type of communication, we'll use either `SOCK_STREAM` (TCP) or `SOCK_DGRAM` (UDP)
protocol	disambiguates the *type*, we'll use either `IPPROTO_TCP` (6) or `IPPROTO_UDP` (17)

NOTES

UDP or TCP

- The choice of protocol is an important design issue
- We'll come back to it later...

Specifying the Local Address

- Whether we're using TCP or UDP, we need to specify the local port number
- We specify the address using the `bind()` system call

```
int bind(int  s, struct sockaddr *name,
         socklen_t *namelen);
```

- Argument details

s	the file descriptor that we got back from `socket()`
name	a pointer to a structure giving the IP address and port
namelen	size of the structure

NOTES

Poor Server Bind Example

- To give you the idea, here's a bad example:

```
struct sockaddr_in sin;
   /* fields: sin_family, sin_port, sin_addr */

memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = htons(12345);

bind(s, (struct sockaddr *)&sin, sizeof(sin));
```

NOTES

What's Wrong with that Example

- Hardcoding the port number as a number is generally a bad idea
  - Better to look it up
  - Always want to let the user override the default
- Notice that we didn't specify the local IP address
  - Doing so could break things for *multi-homed* hosts

A Better Bind Example

```
char *service; /* name or number */
char *protocol; /* "UDP" or "TCP" */

if (service) {
    if ((pse = getservbyname(service, protocol)))
        sin.sin_port = htons(ntohs((u_short)pse->s_port));
    else if ((sin.sin_port =
             htons((u_short)atoi(service))) == 0 )
        Oops("can't get '%s' service entry", service);
} else {
    sin.sin_port = 0;  /* choose at random */
}

bind(s, (struct sockaddr *)&sin, sizeof(sin));
```

Waiting for a Question UDP

- UDP servers are typically easier as they normally just:
  - Get a question in a single datagram
  - Compute the answer
  - Send the answer back in a single datagram
- Many times, the reliability is up to the client
  - The implications of this are important!!!

NOTES

Unconnected Reading and Writing Sendto and Recvfrom

```
int sendto(int s, void *buf, int len, int flags,
           struct sockaddr *saddr, int addrlen);
int recvfrom(int s, void *buf, int len, int flags,
             struct sockaddr *saddr, int *addrlen);
```

- Argument details

s	your socket fd
buf	buffer holding the data to read/write
len	number of bytes of data to read/write
saddr	the remote address
addrlen	size of the remote address
flags	usually 0, check the man pages

Waiting for a Question
UDP Details

```
struct sockaddr_in fsin;
sock = passiveUDP(service);
char buf[BUFSIZE];

while (1) {
    int alen = sizeof(fsin);

    buflen = recvfrom(sock, buf, sizeof(buf), 0,
                      (struct sockaddr *)&fsin, &alen);

    sendto(sock, buf, buflen, 0,
           (struct sockaddr *)&fsin, sizeof(fsin));
}
```


NOTES

Waiting for a Question
TCP

- Because TCP is connection-oriented, you need to do a little more work
  - Open a *passive* TCP connection
  - Create a new TCP connection for each request

Simple TCP Server Example

```
struct  sockaddr_in fsin;
char    *service = "daytime";
int     msock, ssock, alen;

msock = passiveTCP(service);

while (1) {
  char *time_string;
  time_t now;
  ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
       time(&now);
       time_string = ctime(&now);
       write(ssock, time_string, strlen(time_string));
  close(ssock);
}
```


NOTES

Good Server Practice

- Good servers "fork themselves off" into the background when started
  - That's usually what you want them to do anyway
  - But, it makes them more difficult to debug, so there should be a way to defeat that (normally by running them in debugging mode).
- Always provide the user with the ability to specify an alternate port

NOTES

Useful Socket Calls for Servers Listen

```
int listen(int sock, int backlog);
```

- Argument details

sock	the socket to control
backlog	the number of “unaccepted” TCP connections to allow

Useful Socket Calls for Servers Allowing Port Reuse

```
int toggle = 1;
int setsockopt(int sock, SOL_SOCKET, SO_REUSEADDR, (void
*)&toggle, sizeof(toggle));
```

- Argument details

sock	the socket to control
toggle	whether to turn it on or off
len	size of the toggle

- Should be called on a passive socket after the call to `socket()` and before using it for anything

Useful Socket Calls for Servers
Setting Buffer Sizes

```
int bufsize;
int len = sizeof(bufsize);
setsockopt(sock, SOL_SOCKET, SO_RCVBUF, &bufsize, len);
setsockopt(sock, SOL_SOCKET, SO_SNDBUF, &bufsize, len);
getsockopt(sock, SOL_SOCKET, SO_RCVBUF, &bufsize, &len);
getsockopt(sock, SOL_SOCKET, SO_SNDBUF, &bufsize, &len);
```

- Argument details

sock	the socket to control
bufsize	where to read/write the size of the buffer
len	the size of the buffer size to read/write

NOTES

Server Programming Hints

- Comer's book on client/server programming (in several Operating System versions) has some very useful server examples
- Debugging running servers is difficult
  - Use gdb's `connect` routine
  - Learn how to debug a core dump file

When Server Become More Difficult

- A good server needs to be able to handle lots of requests
  - If the requests can come at the same time and it takes a while to compute the answers, you can create an unnecessary bottleneck
    - Multiple processes/threads can help, but have their own dangers
- Maintaining *state* in a server is problematic
- Server location is a hard problem

NOTES

Multi-Threaded Servers

- The simplest way to handle multiple clients at the same time is to create a process/thread for each new request
- Simply using "fork()" for each connection is reasonable for a class assignment
- Writing a good, multi-threaded server is a very complex task
  - Bounding the number of concurrent users is challenging
  - Creating new processes/threads is expensive

Server Lab Assignments

- Write the following simple clients
  - Finish the TCP *daytime* (13) server
    - Much of the code is above
  - Create a TCP *capitalize* server
    - Choose a random port number
    - Accept a stream of text
    - Convert it to all capital letters
    - Echo it back to the user
- "Extra Credit"
  - Write a TCP echo server that can talk to multiple clients at once

NOTES

Theoretical Schedule

9:00-10:30	Background Review
	Application Interface
	Client Applications
10:30-10:45	*Break*
10:45-11:30	Lab Exercise
	Build a simple client
11:30-12:00	Debugging Strategies
12:00-1:00	*Lunch*
1:00-1:30	Lab Exercise
	Debugging Strategies
1:30-2:15	Server Applications
2:15-2:45	Lab Exercise
	Build a simple server
2:45-3:00	*Break*
3:00-3:45	Higher-level Client/Server
3:45-4:30	Lab Exercise
	build a simple RPC application
4:30-5:00	Misc
	Hints and Tips about good networking projects
	Wrapup

Socket-Level Programming Advantages and Disadvantages

- Advantages
  - Gives the students a good understanding of the interaction with the Operating System
  - Gives them an appreciation for the complexity involved
- Disadvantages
  - Lots of details to worry about
  - Server location is very low level

NOTES

Alternatives

- There are two general types of higher-level abstractions for network programming
  - Report procedure call
    - We'll talk about Sun RPC
  - Remote objects
    - We'll discuss JAVA RMI

Sun RPC

- Sun's RPC system has a formal Internet specification
- Most widely know for providing the Network File System (NFS)
- Consists of:
  - Data representation
    - External Data Representation
  - Server location
    - Port mapper
  - Procedure call invocation, argument marshalling

NOTES

External Data Representation (XDR)

- Standard for TCP/IP protocol suite
- Specification for symmetric data conversion
  - There are other alternatives with advantages and disadvantages

XDR Characteristics

- Defines a machine-independent representation
- Includes base data types (e.g., integer, Boolean, string)
- Defines representations for data aggregates (e.g., arrays and structures)
- Represents items in multiples of 32 bits
- Uses implicit types
  - Encoded item does **not** include type
  - Receiver must know what items to expect

NOTES

XDR Data Types

Data	Size	Description
int	32 bits	32-bit signed binary integer
unsigned int	32-bits	32-bit unsigned binary integer
bool	32 bits	Boolean value (*false* or *true* represented by *0* or *1*
enum	32 bits	Enumeration type with values defined by ints (e.g., *RED=1, WHITE=2*
hyper	64 bits	64-bit signed binary integer
unsign. hyper	64-bits	64-bit unsigned binary integer
float	32 bits	Single precision floating pt. number
double	64 bits	Double precision floating pt. number
opaque	arb.	Unconverted data (i.e., data in the sender's native representation)
string	arb.	String of ASCII characters

XDR Data Types (continued)

Data	Size	Description
fixed array	arb.	A fixed-size array of a given data type
counted array	arb.	Array in which the type has a fixed upper limit, but individual arrays may vary up to that size
structure	arb.	A data aggregate, like C's *struct*
discrim. union	arb.	A data structure that allows one of several alternative forms, like C's union or Pascal's variant record
void	0	For optional item when no data is present (e.g., in a structure)
sym. constant	arb.	A symbolic constant and associated value
opt. data	arb.	Allows zero or one occurrences of an item

NOTES

Using Rpcgen

- Programmer writes:
  - Specification of remote program
  - Declaration of remote procedures
  - Declaration of formal parameters
- Rpcgen generates:
  - Client communication stub
  - Server communication stub
  - Server dispatcher
  - Code to marshal arguments
  - Interface procedures for called and calling programs

NOTES

Interface Procedures

- Preserve the original procedure call interface
- Fit between original caller and client communication stub
- Fit between server communication stub and the original called procedure
- Allow rpcgen to choose the calling sequence for communication stubs independent of the original procedure calling sequence

Example Program
string.x

```
const MAXSTRING = 257;

struct args {
    int number;
    string str<MAXSTRING>;
};

struct retval {
    int number;
    string str<MAXSTRING>;
};

program STRINGPROG {
       version STRINGVERS {
           retval STRINGPROC_ONE(args) = 1;
       } = 1;
} = 0x80010695;
```

Generating the RPC pieces

- From a Unix machine, you can say:

```
rpcgen -Ca string.x
```


NOTES

Generating the RPC pieces
Resulting Files

- Rpcgen creates the following files:
  - makefile.string
    - A sample makefile, rename it to `Makefile`
  - string.h
    - A c-includable header file generated from *string.x*
  - string_client.c
    - Skeleton for a client program
  - string_server.c
    - Skeleton for a server program
  - string_clnt.c
    - Client interface routines
  - string_svc.c
    - Server interface routines
  - string_xdr.c
    - Argument marshalling routines

Rpcgen Conventions For File Names

- Name of specification file determines names of generated files
- If specification file is named `Q.x`:
  - Client communication stub in file `Q_clnt.c`
  - Server communication stub in file `Q_svc.c`
  - Declarations and headers in file `Q.h`
  - All XDR procedure calls in file `Q_xdr.c`

NOTES

Generated Header string.h

```
#include <rpc/rpc.h>

#define MAXSTRING 257
struct args {
        int number;
        char *str;
};
typedef struct args args;

struct retval {
        int number;
        char *str;
};
typedef struct retval retval;

#define STRINGPROG ((unsigned long)(0x80010695))
#define STRINGVERS ((unsigned long)(1))
```

NOTES

Generated Header string.h (continued)

```
#if defined(__STDC__) || defined(__cplusplus)
#define STRINGPROC_ONE ((unsigned long)(1))
extern  retval * stringproc_one_1(args *, CLIENT *);
extern  retval * stringproc_one_1_svc(args *, struct svc_req *);
extern int stringprog_1_freeresult(SVCXPRT *, xdrproc_t, caddr_t);
#else /* K&R C */
#define STRINGPROC_ONE ((unsigned long)(1))
extern  retval * stringproc_one_1();
extern  retval * stringproc_one_1_svc();
extern int stringprog_1_freeresult();
#endif /* K&R C */

#if defined(__STDC__) || defined(__cplusplus)
extern  bool_t xdr_args(XDR *, args*);
extern  bool_t xdr_retval(XDR *, retval*);
#else /* K&R C */
extern bool_t xdr_args();
extern bool_t xdr_retval();
#endif /* K&R C */
```


string_client.c

```
#include "string.h"

void stringprog_1(char *host)
{
        CLIENT *clnt;
        retval  *result_1;
        args  stringproc_one_1_arg;

        clnt = clnt_create(host, STRINGPROG, STRINGVERS, "netpath");
        if (clnt == (CLIENT *) NULL) {
                clnt_pcreateerror(host);
                exit(1);
        }

        result_1 = stringproc_one_1(&stringproc_one_1_arg, clnt);
        if (result_1 == (retval *) NULL) {
                clnt_perror(clnt, "call failed");
        }
        clnt_destroy(clnt);
}
```

string_client.c (continued)

```
main(int argc, char *argv[])
{
        char *host;
        if (argc < 2) {
                printf("usage:  %s server_host\n", argv[0]);
                exit(1);
        }
        host = argv[1];
 )        stringprog_1(host);
}
```


NOTES

string_client.c

- Here's a version that I modified:

```
void
stringprog_1(char *host)
{
   CLIENT *clnt;
   retval  *result_1;
   args  stringproc_one_1_arg;

   clnt = clnt_create(host, STRINGPROG, STRINGVERS,
                      "netpath");
   if (clnt == (CLIENT *) NULL) {
           clnt_pcreateerror(host);
           exit(1);
   }

   stringproc_one_1_arg.str = "This is a test";
   stringproc_one_1_arg.number = 42;
```

string_client.c (continued)

```
    result_1 = stringproc_one_1(&stringproc_one_1_arg,
                                 clnt);
    if (result_1 == (retval *) NULL) {
            clnt_perror(clnt, "call failed");
    }

    printf("Return string is '%s'\n", result_1->str);
    printf("Return number is %d\n", result_1->number);

    clnt_destroy(clnt);
}
```


NOTES

string_server.c

```
#include "string.h"

retval *
stringproc_one_1_svc(args *argp, struct svc_req *rqstp)
{
       static retval  result;

       /*
        * insert server code here
        */

       return (&result);
}
```

NOTES

string_server.c

- Here's a version that I modified

```
#include "string.h"

retval *
stringproc_one_1_svc(args *argp,
                     struct svc_req *rqstp)
{
        static retval  result;

        printf("server: arg string is '%s'\n",
               argp->str);
        printf("server: arg number is %d\n",
               argp->number);
        result.str = "This is the answer";
        result.number = 99;

        return (&result);
}
```


string_clnt.c

```
#include <memory.h> /* for memset */
#include "string.h"

retval *
stringproc_one_1(args *argp, CLIENT *clnt)
{
    static retval clnt_res;

    memset((char *)&clnt_res, 0, sizeof (clnt_res));
    if (clnt_call(clnt, STRINGPROC_ONE,
            (xdrproc_t) xdr_args, (caddr_t) argp,
            (xdrproc_t) xdr_retval, (caddr_t) &clnt_res,
            TIMEOUT) != RPC_SUCCESS) {
            return (NULL);
    }
    return (&clnt_res);
}
```

makefile.string

```
CLIENT = string_client
SERVER = string_server
SOURCES_CLNT.c =
SOURCES_CLNT.h =
SOURCES_SVC.c =
SOURCES_SVC.h =
SOURCES.x = string.x
TARGETS_SVC.c = string_svc.c string_server.c string_xdr.c
TARGETS_CLNT.c = string_clnt.c string_client.c string_xdr.c
TARGETS = string.h string_xdr.c string_clnt.c string_svc.c string_client.c string_server.c

OBJECTS_CLNT = $(SOURCES_CLNT.c:%.c=%.o) $(TARGETS_CLNT.c:%.c=%.o)
OBJECTS_SVC = $(SOURCES_SVC.c:%.c=%.o) $(TARGETS_SVC.c:%.c=%.o)
# Compiler flags

CFLAGS += -g
LDLIBS += -lnsl
RPCGENFLAGS =

# Targets
all : $(CLIENT) $(SERVER)
$(TARGETS) : $(SOURCES.x)
```


NOTES

```
	rpcgen $(RPCGENFLAGS) $(SOURCES.x)
$(OBJECTS_CLNT) : $(SOURCES_CLNT.c) $(SOURCES_CLNT.h) $(TARGETS_CLNT.c)
$(OBJECTS_SVC) : $(SOURCES_SVC.c) $(SOURCES_SVC.h) $(TARGETS_SVC.c)
$(CLIENT) : $(OBJECTS_CLNT)
	$(LINK.c) -o $(CLIENT) $(OBJECTS_CLNT) $(LDLIBS)
$(SERVER) : $(OBJECTS_SVC)
	$(LINK.c) -o $(SERVER) $(OBJECTS_SVC) $(LDLIBS)
 clean:
	 $(RM) core $(TARGETS) $(OBJECTS_CLNT) $(OBJECTS_SVC) $(CLIENT) $(SERVER)
```

string_svc.c

- This is a long, ugly file
- It includes the main procedure, dispatch code, and etc
- You shouldn't need to edit it

NOTES

string_svc.c (Interesting bits)

```
static void
stringprog_1(struct svc_req *rqstp,
             register SVCXPRT *transp)
{
    union {
            args stringproc_one_1_arg;
    } argument;
    char *result;
    xdrproc_t xdr_argument, xdr_result;
    char *(*local)(char *, struct svc_req *);

    switch (rqstp->rq_proc) {
    case NULLPROC:
            (void) svc_sendreply(transp,
                    (xdrproc_t) xdr_void, (char *)NULL);
            return;
```

string_svc.c (More interesting bits)

```
    case STRINGPROC_ONE:
            xdr_argument = (xdrproc_t) xdr_args;
            xdr_result = (xdrproc_t) xdr_retval;
            local = (char *(*)(char *, struct svc_req *))
                     stringproc_one_1_svc;
            break;

    default:
            svcerr_noproc(transp);
            return;
    }
```


NOTES

string_svc.c (More interesting bits 2)

```
    (void) memset((char *)&argument, 0, sizeof (argument));
    if (!svc_getargs(transp, xdr_argument,
                (caddr_t) &argument)) {
            svcerr_decode(transp);
            return;
    }
    result = (*local)((char *)&argument, rqstp);
    if (result != NULL && !svc_sendreply(transp,
                        xdr_result, result)) {
            svcerr_systemerr(transp);
    }
    if (!svc_freeargs(transp, xdr_argument,
                (caddr_t) &argument)) {
            _msgout("unable to free arguments");
            exit(1);
    }
}
```

string_svc.c (Still more interesting bits)

```
main()
{
#ifndef RPC_SVC_FG
        int size;
        struct rlimit rl;
        pid = fork();
        if (pid < 0) {
                perror("cannot fork");
                exit(1);
        }
        if (pid)
                exit(0);
        rl.rlim_max = 0;
        getrlimit(RLIMIT_NOFILE, &rl);
        if ((size = rl.rlim_max) == 0)
                exit(1);
        for (i = 0; i < size; i++)
                (void) close(i);
        i = open("/dev/console", 2);
        (void) dup2(i, 1);
        (void) dup2(i, 2);
```


NOTES

string_svc.c (Still more interesting bits)

```
        setsid();
        openlog("string", LOG_PID, LOG_DAEMON);
#endif
    if (!svc_create(stringprog_1, STRINGPROG,
                    STRINGVERS, "netpath")) {
            _msgout("unable to create ... for netpath.");
            exit(1);
    }
    svc_run();
}
```

Simple Example with Variant Record (From nfs_prot.x)

```
/* Error status */
enum nfsstat {
    NFS_OK= 0,              /* no error */
    NFSERR_PERM=1,          /* Not owner */
    NFSERR_NOENT=2,         /* No such file or directory */
    NFSERR_NAMETOOLONG=63,  /* File name too long */
};

/* Reply status with file attributes */
union attrstat switch (nfsstat status) {
case NFS_OK:
    fattr attributes;
default:
    void;
};
```


NOTES

Simple Example with Variant Record (continued)

```
/* Remote file service routines */
program NFS_PROGRAM {
    version NFS_VERSION {
            void
            NFSPROC_NULL(void) = 0;

            attrstat
            NFSPROC_GETATTR(nfs_fh) =       1;

            attrstat
            NFSPROC_WRITE(writeargs) = 8;
            }
}
```

NOTES

Theoretical Schedule

9:00-10:30	Background Review
	Application Interface
	Client Applications
10:30-10:45	*Break*
10:45-11:30	Lab Exercise
	Build a simple client
11:30-12:00	Debugging Strategies
12:00-1:00	*Lunch*
1:00-1:30	Lab Exercise
	Debugging Strategies
1:30-2:15	Server Applications
2:15-2:45	Lab Exercise
	Build a simple server
2:45-3:00	*Break*
3:00-3:45	Higher-level Client/Server
3:45-4:30	Lab Exercise
	build a simple RPC application
4:30-5:00	Misc
	Hints and Tips about good networking projects
	Wrapup

What Makes a good programming Project

- Not too much busy work
- Many possibilities for extensions
- Group projects in which both a client and server fit together work well
- RPC is nice, because all of the students can implement to the same interface and then the programs all fit together