COMP/ELEC 429/556
Introduction to Computer Networks

Creating a Network Application

Some slides used with permissions from Edward W. Knightly, T. S. Eugene Ng, Ion Stoica, Hui Zhang
How to Programmatically Use the DNS System?

Operating system comes to the rescue

• getaddrinfo()
• getnameinfo()

• On CLEAR, you can type “man getaddrinfo” to get a detailed description of the function
  – “man” stands for manual
How to Programmatically Send/Receive Data over the Internet?
Operating System comes to the rescue

• Reliable byte stream network communications service
  – The most common model
  – Underlies almost every network application you use
**Reliable Byte Stream Model**

- The elemental transmission unit is a byte (8 bits)
- Bytes are delivered reliably, first in first out
  - We will learn how this is accomplished later

```
0x3a  0xf2  0x12
```

Network Abstraction

```
0  15  16
size

0
numeric field 1

15 16
numeric field 2

31
some other data
```
Reliable Byte Stream

- Bytes are delivered without any notion of application message units
  - A recv() call may receive an arbitrary portion of a message

Example 1

```
0x3a 0xf2 0x12
1st recv() got 0x12
2nd recv() got 0xf2, 0x3a
```

Example 2

```
0x3a 0xf2 0x12
1st recv() got nothing due to timeout
2nd recv() got 0x12, 0xf2
3rd recv() got 0x3a
```

T. S. Eugene Ng
eugeneng at cs.rice.edu
Rice University
Application decides message format and how to interpret the byte stream

E.g.
• Each message is a null (0x00) terminated string
• In this case, a program reads in bytes until encountering 0x00 to form a valid message

E.g.
• Each message carries its own size
• In this case, a program interprets the size in the message to determine where the message ends
Messages with Numeric Values

![Diagram of message structure with size, numeric fields, and some other data]

- 2 size bytes: 0xf2 0x12

What is the size?

0x12f2 = 4,850
0xf212 = 61,970

When sending numeric value over the network, must adopt a consistent byte ordering on both sender and receiver.

Network byte order (a.k.a. Big endian) – first byte sent/received is the most significant

size is 0x12f2 = 4,850
Why the need to specify network byte ordering convention for numbers?

• CPUs may choose little-endian or big-endian format when storing numbers in registers and RAM
  – Most Intel CPUs are “little-endian” (1st byte least significant)
  – Network byte ordering convention is “big-endian” (1st byte most significant)
• The number 4,850 (0x12f2), which one below is stored in “big-endian”?

```
<table>
<thead>
<tr>
<th></th>
<th>0x12</th>
<th>0xf2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lower address</td>
<td>higher address</td>
<td>Big-endian</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>0xf2</th>
<th>0x12</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lower address</td>
<td>higher address</td>
<td>Little-endian</td>
<td></td>
</tr>
</tbody>
</table>
```
Byte ordering conversion functions

- host order to network order
  - htons() for short integer (i.e. 16 bit)
  - htonl() for long integer (i.e. 32 bit)
  - htonl64() for 64 bit integer

- network order to host order
  - ntohs() for short integer (i.e. 16 bit)
  - ntohl() for long integer (i.e. 32 bit)
  - be64toh() for 64 bit integer

- Must be careful in deciding whether conversion is required
  - Is a value numeric?
  - Is a numeric value already stored in network byte order?

- These functions are “no-op” on CPUs that use network byte ordering (big-endian) for internal number representation
  - But a program must compile and work regardless of CPU, so these functions have to be used for compatibility
What about bit ordering?

- Which bit in a byte received over the network is most significant?
- OMG, not again!

- Bit ordering within a byte is consistently handled by hardware
  - Byte is the basic unit in hardware
  - In contrast, 16-bit, 32-bit, 64-bit… values are not the most basic units
How to Programmatically Send/Receive Data over the Internet?
Operating System comes to the rescue

- Socket API
  - Refresh your memory of the mechanics of using socket for network communications
Overview of the Socket API

Client

- create a descriptor
  - socket
- connect
- send, recv
- close

Server

- create a descriptor
  - socket
- bind
- listen
- accept
- send, recv
- close

Connection request

A new descriptor is created for this connection.
How to support multiple simultaneous connections?
Operating System comes to the rescue

• Event-driven concurrency
  – Refresh your memory of one way to create the illusion of concurrently handling multiple network conversations even on a uni-processor

• Another way is multi-threading
  – Can achieve real concurrency on multi-processors
  – Refer to COMP321 material

• A combination of event-driven concurrency and multi-thread concurrency is needed to achieve the highest server scalability
Event-Based Concurrent Servers

• Maintain a pool of descriptors
• Repeat the following forever:
  – Block until:
    • New connection request arrives on the listening descriptor
    • New data arrives on an existing connected descriptor
    • A connected descriptor is ready to be written to
    • A timeout occurs (can be configured to have no timeout)
  – If new connection request, add the new connection to the pool of connections
  – If new data arrives, read any available data from the connection
  – If descriptor is ready to be written, write whatever data is pending
  – If timeout, do whatever is appropriate for timeout
• Can wait for input from local I/O (standard input) and remote I/O (socket) simultaneously!
• Many implementations: select(), poll(), epoll(), etc.
Event-Based Concurrent I/O

```c
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

- **readfds, writefds**
  - Opaque bit vector (max `FD_SETSIZE` bits) that indicates membership in a descriptor set
  - If bit $k$ is 1, then descriptor $k$ is a member of the descriptor set

- **nfds**
  - Maximum descriptor value + 1 in the set
  - Tests descriptors 0, 1, 2, ..., $nfds$ - 1 for set membership

- **select()** returns the number of ready descriptors and sets each bit of `readfds, writefds` to indicate the ready status of its corresponding descriptor
Macros for Manipulating Set Descriptors

- `void FD_ZERO(fd_set *fdset);`
  - Turn off all bits in `fdset`

- `void FD_SET(int fd, fd_set *fdset);`
  - Turn on bit `fd` in `fdset`

- `void FD_CLR(int fd, fd_set *fdset);`
  - Turn off bit `fd` in `fdset`

- `int FD_ISSET(int fd, *fdset);`
  - Is bit `fd` in `fdset` turned on?