Computer Networking

CS2 Recitation Series
Wednesday, February 19, 2014

Administrivia

- Assignment 7 is coming out soon
  - There will be an email when it is available.
  - email <cs2-tas@ugcs.caltech.edu> with any bug reports
  - Two days of recitation on this set.
- Othello tournament is approaching!
  - weeks 9/10
  - teams of 2 encouraged (solo permitted)
  - start thinking about teams and team names

Overview

- What is a computer network?
- How can applications communicate on the network?
- What do we need to do to write a networked application?

What is a computer network?

- A collection of computers interconnected for the purpose of sharing data and information
  - 1970s: defense researchers around the country (ARPANET)
  - 1980s: civilian researchers at major universities (NSFNet)
  - 1990s: anyone with a computer (commercial Internet)
  - 2000s: everyone with a computer...

Network infrastructure

- Hosts
  - individual computers
- Switches
  - ties many hosts together
  - defines a 'local area network'
- Routers
  - control and route traffic between networks
- Links
  - connect different components together

How do hosts work?

- 1 host = 1 computer (usually)
- Each host has one or more network interfaces
  - Ethernet adapters, wireless adapters, etc.
- Each network interface has its own network address
  - IPv4: 32-bit address (131.215.254.254)
  - IPv6: 128-bit address

Host problems

- Hosts can run multiple applications. How does each one get the correct traffic?
- Ports!
  - Every protocol gets a port number from 0 to 65535
  - Ports numbered < 1024 are "system" ports
    - require system-level access to assign
    - almost all associated with common services (HTTP, FTP, SSH, ...)
  - Ports numbered >= 1024 are basically free-for-all
    - some are associated with specific services (IRC, Eve Online, ...)
    - any application can use these ports

Host problems

- Few people know how to talk directly to hardware... how do we let applications use the network adapter?
- Sockets!
  - "file-like" constructs provided by the operating system
  - can be connected to remote hosts
  - while connected, can read/write almost at will
  - applications can use with no knowledge of the underlying hardware
  - two main types: TCP and UDP.

TCP sockets

- TCP = Transmission Control Protocol
  - stateful connection-based protocol
    - client has to open a connection to server before communication is possible
  - stream-based protocol
    - applications can send arbitrarily long data without having to packetize it (the OS networking stack takes care of this)
  - guaranteed reliable in-order delivery
    - applications receive data in the order in which it was sent
  - automatic flow control
    - adjusts transmission rate in response to

TCP sockets

- TCP's inherent reliability makes it great for some applications
  - file transfer
  - data transfer
    - where you can't afford to lose a packet...
- Unfortunately, reliability induces delay
  - lost packets must be retransmitted
  - can take a long time to determine that a packet is actually lost
  - "lost" packets may end up actually arriving (and then must be filtered out)
  - not ideal for low-latency applications

UDP sockets

- UDP = User Datagram Protocol
  - stateless, connectionless protocol
    - servers just listen
    - clients just send
  - datagram-based protocol
    - clients must packetize data themselves
  - unreliable protocol
    - packets may arrive out of order
    - packets may not arrive at all (!!)
  - no flow control
    - easy to oversaturate a link if you're not careful

UDP sockets

- UDP is great for applications where low latency trumps reliability
  - audio/video streaming (few packets lost is not a big deal)
  - computer games (packets must arrive ASAP)
  - High Frequency Trading
- UDP is not an ideal choice where ordering or reliability is required
  - transferring encrypted data (though it can be done)

The POSIX Sockets API

- POSIX provides an API exposing socket functionality to applications
  - can create sockets
  - can connect (as a client) to a remote host
  - can send/receive data
  - can bind (as a server) to a local port and listen for connections
  - can accept incoming connections
- The details of the POSIX API are potentially confusing
- We provide our own "CS2Net" object-oriented wrapper for this assignment

The CS2Net Sockets API

We can create a CS2Net::Socket object for ourselves:

```
CS2Net::Socket sock;
```

This object will take care of all the underlying operations for us.

The CS2Net Sockets API

Right now, we have an undifferentiated socket. Suppose we're a client. We can connect to a remote host:

```
std::string hostname("host1.example.net");
uint16_t port = 9001;
int con_ret = sock.Connect(&hostname, port);
```

Wait, what's a hostname?

- Recall that hosts have numerical addresses (10.20.30.40)
- Numbers are inconvenient to remember
- Hostnames are human-readable aliases for particular addresses
  - "caltech.edu" -> 131.215.239.141
- The OS is responsible for translating hostnames to IP addresses
  - uses a service called DNS (Domain Name System) to accomplish this task

The CS2Net Sockets API

- There are a number of reasons why a connection could fail
  - network interface is down
  - remote host is down
  - remote address is wrong
  - remote host is not accepting connections
  - remote host is firewalled...
- Connect() returns a return value
  - if this is less than 0, a bad thing happened
  - if this is equal to 0, we're fine
  - you will need to handle this in your code (check the assignment documentation)

The CS2Net Sockets API

Now that we have an open connection, we can try to send some data to the other end:

```
std::string data("Hello, world!\n");
int send_ret = sock.Send(&data);
```

The CS2Net Sockets API

We can also try to receive some data from the other end:

```
std::string * got_data;
got_data = sock.Recv(1024, false);
```

Blocking functions

- The sockets created by our API are blocking
  - That is, send and recv calls hang until completed
- "Completed" is usually defined loosely here
  - Send() and Recv() usually return when any amount of data is successfully sent or received
- Our Recv() function can also wait for all the requested data to arrive
  - e.g. `Recv(1024, true)` blocks until all 1024 bytes come in (or there is an error)

How to avoid the blocking problem?

- Can't wait on Recv() all day; other things have to happen in the meantime!
- How to avoid this problem?
- Non-blocking sockets
  - Send() and Recv() error instead of blocking
  - Beyond the scope of this assignment
- Multiple threads
  - Put blocking operations on separate threads
  - Not necessary for what we need to do
- Polling
  - Ask the socket periodically if data is available
  - We use this mechanism in our assignment

The CS2Net Sockets API

The poll() system call is not easy to set up. Unfortunately, that means the abstraction isn't so easy either.

```
std::vector<CS2Net::PollFD> to_poll(1);
to_poll[0].sock = &sock;
to_poll[0].SetRead(true);
// poll with 10ms timeout
int poll_err = CS2Net::Poll(&to_poll, 10);
// ... check for various errors ...
if(to_poll[0].CanRead())
{
    // ... do stuff ...
}
```

What did that just do?

- We set up a vector of CS2Net::PollFD objects
  - each PollFD has a pointer to a CS2Net::Socket
  - we SetRead() each one to say we want to read some data
- We call CS2Net::Poll() on this vector
- CS2Net::Poll() updates each PollFD based on the requested events
  - now we can call CanRead() on each one to check if we can read
  - or call HasHangup() or HasError() for error checking

What if the remote host disconnects during a Poll()?

Empirically, we observe the following:

- When a remote host disconnects, the corresponding socket always has data available to read, but Recv() invariably returns 0 bytes.

We can detect and handle this condition.

Cleaning up CS2Net Sockets

- CS2Net::Socket objects autodisconnect when destroyed
  - statically allocated Sockets: going out of scope
  - dynamically allocated Sockets: when deleted
- We can disconnect explicitly:

```
sock.Disconnect();
```

Featureful client messages

Each client2 / server2 message looks like this:

Type (1)	Payload Len (2)	Data (n)

For example, a chat message might look like this (in hex):

20	0D 00	48 65 6C 6C 6F 2C 20 77 6F 72 6C 64 21

This is the chat message "Hello, world!"

(Message types in CS2ChatProtocol.h)

How do we put binary numbers in a string?

Typecasting is your friend.

```
std::string foo = ...;
unsigned short int bar = 1000;
const char * __bar_bits;
__bar_bits = (const char *) (&bar);

foo.append(__bar_bits,
           sizeof(unsigned short int));
```

Endianness

- Little-endian: least significant bytes first
  - 1000 = E8 03
- Big-endian: most significant bytes first
  - 1000 = 03 E8
- "Network byte order": standard byte order chosen for platform interoperability
  - the standard is big-endian
- For this assignment, payload size is little-endian
  - slightly less work on your part

For next time...

- Listener sockets
- Accepting connections
- Tracking and polling multiple connections
- Some light GUI work
- Special topics (time permitting)