Chapter 2a
SOCKET PROGRAMMING

Computer Networks
Timothy Roscoe
Summer 2007

Overview
- Basic socket concepts
- Java socket programming
  - Client & server
  - TCP & UDP
  - Threads
- C socket programming
  - API details
  - TCP client and server
  - Asynchronous I/O and events
- Bonus: EiffelNet API slides

Socket programming
- Learn building client/server applications that communicate using sockets, the standard application programming interface

Socket API
- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by applications
- client/server paradigm
- two types of transport service via socket API
  - unreliable datagram
  - reliable, byte stream-oriented

Goal

Socket programming with TCP
- a door between application process and end-end-transport protocol (UDP or TCP)

TCP service
- reliable transfer of bytes from one process to another
### Socket programming with TCP

Client must contact server
- server process must first be running already
- server must have created socket ("door") that welcomes client's contact

Client contacts server by
- creating client-local TCP socket
- specifying IP address and port number of server process

When client creates socket: client TCP establishes connection to server TCP
When contacted by client, server TCP creates new socket for server process to communicate with client
  - allows server to talk with multiple clients

**application viewpoint**
TCP provides reliable, in-order transfer of bytes ("pipe") between client and server

### Socket programming with UDP

Remember: UDP: no "connection" between client and server

- no handshaking
- sender explicitly attaches IP address and port of destination
- server must extract IP address, port of sender from received datagram

- UDP: transmitted data may be received out of order, or lost

**application viewpoint**
UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server

### Java API vs. C API

- **Java**:
  - High-level, easy to use for common situations
  - Buffered I/O
  - Failure abstracted as exceptions
  - Less code to write

- **C**:
  - Low-level $\Rightarrow$ more code, more flexibility
  - Original interface
  - Maximum control
  - Basis for all other APIs in Unix (and Windows)

### Socket programming with TCP (Java)

Example client-server application
- client reads line from standard input (inFromUser stream), sends to server via socket (outToServer stream)
- server reads line from socket
- server converts line to uppercase, sends back to client
- client reads and prints modified line from socket (inFromServer stream)

**Diagram**
- Input stream: sequence of bytes into process
- Output stream: sequence of bytes out of process
- TCP socket
- client TCP socket
- client process
- server
- server TCP socket
- server process
- keyboard
- monitor
- ...
Client/server socket interaction with TCP (Java)

Server (running on hostid)

- create socket, port=x, for incoming request:
  \[\text{welcomeSocket} = \text{ServerSocket()}\]

- wait for incoming connection request:
  \[\text{connectionSocket} = \text{welcomeSocket.accept()}\]

- read request from \text{connectionSocket}

- write reply to \text{connectionSocket}

- close \text{connectionSocket}

Client

- create socket, connect to hostid, port=x:
  \[\text{welcomeSocket} = \text{ClientSocket}\]

- send request using \text{clientSocket}

- read reply from \text{clientSocket}

- close \text{clientSocket}

Example: Java client (TCP)

```java
import java.io.*;
import java.net.*;

class TCPClient {
    public static void main(String argv[]) throws Exception {
        String sentence;
        String modifiedSentence;
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));
        Socket clientSocket = new Socket("hostname", 6789);
        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());

        BufferedReader inFromServer =
            new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
        sentence = inFromUser.readLine();
        outToServer.writeBytes(sentence + '\n');
        modifiedSentence = inFromServer.readLine();
        System.out.println("FROM SERVER: "+ modifiedSentence);
        clientSocket.close();
    }
}
```

Example: Java server (TCP)

```java
import java.io.*;
import java.net.*;

class TCPServer {
    public static void main(String argv[]) throws Exception {
        String clientSentence;
        String capitalizedSentence;
        ServerSocket welcomeSocket = new ServerSocket(6789);
        while(true) {
            Socket connectionSocket = welcomeSocket.accept();
            BufferedReader inFromClient =
                new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));
            System.out.println("FROM CLIENT: "+ inFromClient.readLine());
            connectionSocket.close();
        }
    }
}
```
Example: Java server (TCP), continued

```java
dataOutputStream outToClient = new DataOutputStream(connectionSocket.getOutputStream());
clientSentence = inFromClient.readLine();
capitalizedSentence = clientSentence.toUpperCase() + '\n';
outToClient.writeBytes(capitalizedSentence);
```

Problem: One client can delay other clients

In fact, one client can block other clients

The Problem: Concurrency

- Networking applications are
  - Inherently concurrent
  - Prone to partial failure
- Hence, “blocking” (waiting for something else) can
  - Slow things down (only one machine running at a time)
  - REALLY slow things down (mostly, no machines running at a time)
  - Stop things (something stops and everything else waits)
- Central problem of distributed systems
  - Not really networking, but probably should be
One solution: Threads

ServerSocket welcomeSocket = new ServerSocket(6789);
while(true) {
    Socket connectionSocket = welcomeSocket.accept();
    ServerThread thread = new ServerThread(connectionSocket);
    thread.start();
}

public class ServerThread extends Thread {
/* ... */
    BufferedReader inFromClient = new BufferedReader(new
        InputStreamReader(connectionSocket.getInputStream()));
    DataOutputStream outToClient = new DataOutputStream(
        connectionSocket.getOutputStream());
    clientSentence = inFromClient.readLine();
    capitalizedSentence = clientSentence.toUpperCase() + '\n';
    outToClient.writeBytes(capitalizedSentence);
/* ... */
}

Does this solve the problem?

Threads

- Threads are programming abstractions of separate activities
- Still need to worry about resources:
  - How many threads?
  - How long should each thread live for?
- Many programming patterns:
  - Thread-per-request
  - Worker pools
  - Etc.
- See distributed systems course for more on these

Client/server socket interaction: UDP (Java)

Server (running on hostid)

- create socket, port=x, for incoming request:
  serverSocket = DatagramSocket()
- read request from serverSocket
- write reply to serverSocket specifying client host address, port number

Client

- create socket, clientSocket = DatagramSocket()
- Create, address (hostid, port=x), send datagram request using clientSocket
- read reply from clientSocket
- close clientSocket

Example: Java client (UDP)

- Input: receives packet (TCP received "byte stream")
- Output: sends packet (TCP sent "byte stream")
**Example: Java client (UDP)**

```java
import java.io.*;
import java.net.*;

public class UDPClient {
    public static void main(String[] args) throws Exception {
        BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress IPAddress = InetAddress.getByName("hostname");
        byte[] sendData = new byte[1024];
        byte[] receiveData = new byte[1024];
        String sentence = inFromUser.readLine();
        sendData = sentence.getBytes();
        DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
        clientSocket.send(sendPacket);
        DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
        clientSocket.receive(receivePacket);
        String modifiedSentence = new String(receivePacket.getData());
        System.out.println("FROM SERVER:" + modifiedSentence);
        clientSocket.close();
    }
}
```

**Example: Java server (UDP), continued**

```java
import java.io.*;
import java.net.*;

public class UDPServer {
    public static void main(String[] args) throws Exception {
        DatagramSocket serverSocket = new DatagramSocket(9876);
        byte[] receiveData = new byte[1024];
        byte[] sendData = new byte[1024];
        while(true) {
            DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
            serverSocket.receive(receivePacket);
            String sentence = new String(receivePacket.getData());
            InetAddress IPAddress = receivePacket.getAddress();
            int port = receivePacket.getPort();
            String capitalizedSentence = sentence.toUpperCase();
            sendData = capitalizedSentence.getBytes();
            DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, port);
            serverSocket.send(sendPacket);
        }
    }
}
```
**TCP Client in C step by step...**

- Create a socket
- Bind the socket
- Resolve the host name
- Connect the socket
- Write some data
- Read some data
- Close
- Exit

*General flavour: much lower level...*

---

**C API: socket()**

```c
#include <sys/types.h>
#include <sys/socket.h>

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

- `domain`: Address family or domain. In this case IPv4.
- `type`: Service type requested, e.g. `SOCK_STREAM` or `SOCK_DGRAM`.
- `protocol`: Protocol within a service type; 0 ⇒ OS chooses: `IPPROTO_TCP` (often only one!)

```c
int s = socket(AF_INET, SOCK_STREAM, 0);
```

---

**C API: Specifying local address**

```c
struct in_addr {
  u_int32_t s_addr;
};
```

IPv4 address: 4 bytes packed in an integer

```c
struct sockaddr_in {
  short sin_family;
  u_short sin_port;
  struct in_addr sin_addr;
  char sin_zero[8];
};
```

- `sin_family`: Set to `PF_INET` (why?)
- `sin_port`: Port number and local address in network endian byte order
- `sin_addr`: Padding (why?)

---

**C API: Usage of bind()**

```c
struct sockaddr_in sa;

memset(&sa, 0, sizeof(sa));
sa.sin_family = PF_INET;
sa.sin_port = htons(0);
sa.sin_addr = htonl(INADDR_ANY);

if (bind(s, (struct sockaddr *)&sa, sizeof(sa)) < 0) {
  perror("binding to local address");
  close(s);
  return -1;
}
```

---

*What is **bind()** actually for?*

---

*Seems like a lot of work...*
C sockets: resolving a host name

```c
cstruct hostent *h;

h = gethostbyname(host)
if ((h || h->h_length != sizeof(struct in_addr)) {
    fprintf(stderr, "%s: no such host\n", host);
    return -1;
}
```
Result: h->h_addr points to the address of the machine we want to talk to.

C API: Connecting (finally!)

```c
struct sockaddr_in sa;
sa.sin_port = htons(port)
sa.sin_addr = *(struct sockaddr *)h->h_addr;

if (connect(s, (struct sockaddr *)&sa, sizeof(sa)) < 0 {
    perror(host);
    close(s);
    return -1;
}
```
Address structure gives all needed info about the remote end-point

Sending and receiving data

```c
ssize_t send(int s, const void *buf, size_t len, int flags);
• With no flags (0), equivalent to write( s, buf, len )

ssize_t recv(int s, void *buf, size_t len, int flags);
• With no flags, equivalent to read( s, buf, len )

ssize_t sendto(int s, const void *buf, size_t len, int flags,
    const struct sockaddr *to, socklen_t tolen);

ssize_t recvfrom(int s, void *buf, size_t len, int flags,
    struct sockaddr *from, socklen_t *fromlen);
• And these two are for… ?
```

Putting it all together – the “W” client.
**TCP server programming in C**

```c
tcp server programming in C

int listen(int sockfd, int backlog);
• Takes a bound (but not connected!) socket
• Turns it into a listener for new connections
• Returns immediately
• backlog: number of outstanding connection attempts
  – See accept() on next slide
  – Traditionally, 5 (not any more...)
• What do you do with a listening socket?
```

```c
tcp server programming in C

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
• Takes a listening socket sockfd
• Waits for a connection request, and accepts it (!)
  – You don’t get to say “no”...
• Returns a new socket for the connection
  – Plus the address of the remote peer
```

**TCP server: example pattern**

1. Create a server socket and bind to a local address
2. Call listen()
3. Loop:
   1. Call accept() and get a new (“connection”) socket back
   2. Read client’s request from the connection socket
   3. Write response back to the connection socket
   4. Close the connection socket
• See real example server...

**Asynchronous programming: O_NONBLOCK**

```c
if ((n = fcntl (s, F_GETFL)) < 0
    || fcntl(s, F_SETFL, n | O_NONBLOCK) < 0) {
    perror("O_NONBLOCK");
}
```

Socket descriptor now behaves differently:
• read/recv: as normal if there is data to read. EOF returns 0.
  Otherwise, returns -1 and errno set to EAGAIN.
• write/send: if data cannot yet be sent, returns -1 and
  errno = EAGAIN
• connect: if no immediate success, returns -1 and
  errno = EINPROGRESS
• accept: if no pending connections, returns -1 and
  errno = EWOULDBLOCK
Asynchronous programming: 

\[
\begin{align*}
\text{int } & \text{select}(\text{int } nfds, \\
& \quad \text{fd_set *readfds,} \\
& \quad \text{fd_set *writefds,} \\
& \quad \text{fd_set *exceptfds,} \\
& \quad \text{struct timeval *timeout});
\end{align*}
\]

- Returns when anything happens on any set file (i.e. socket) descriptor, or the timeout occurs.
- The \text{fd_set}s are modified to indicate fds that are active.

A basic event loop

- Operations to register callbacks for:
  - File (socket) descriptors
  - Timeout events
- Map from socket descriptor $\rightarrow$ callback
- Priority queue of timer events
- Loop:
  - Process timeouts
  - Call select with next timeout
  - Process any active socket descriptors

Event programming:

- Event programming is hard
  - Callbacks $\Rightarrow$ need to maintain state machine for each activity ("stack ripping")
  - Anything that blocks has to be handled with a callback
  - Hard to deal with long-running operations
- But...
  - No need for synchronization (at least, with one processor)
  - Very scalable (only one thread)
  - Model similar to interrupts $\Rightarrow$ close to how one needs to implement a practical networking stack

More information on TCP and C

- Upcoming labs...
- Some of this material is from the excellent: "Using TCP Through Sockets", by David Mazières, Frank Dabek, and Eric Peterson. [http://people.inf.ethz.ch/troscoe/teaching/net2-1.pdf](http://people.inf.ethz.ch/troscoe/teaching/net2-1.pdf)
Finally...

- Backup slides also cover Eiffel networking classes
  - Exercises/labs will be Java and C
  - Eiffel abstracts events into "pollers" and related objects

- Next week:
  - Java development
  - Eclipse tutorial for Java and C

- Then:
  - Transport protocols.

---

**EiffelNet: Sockets and communication modes**

- Two modes of socket communication:
  - stream communication
  - datagram communication

- Stream socket:
  - provided by the STREAM_classes
  - provides sequenced communication without any loss or duplication of data
  - synchronous: the sending system waits until it has established a connection to the receiving system and transmitted the data

- Datagram socket:
  - provided by the DATAGRAM_classes
  - asynchronous: the sending system emits its data and does not wait for an acknowledgment
  - efficient, but it does not guarantee sequencing, reliability or non-duplication

**Example: Eiffel Server (TCP - stream socket)**

```eiffel
class OUR_SERVER
  inherit
  SOCKET_RESOURCES
  STORABLE
create
make
feature
  soc1, soc2: NETWORK_STREAM_SOCKET
  make (argv: ARRAY[STRING] is
    local
      count: INTEGER
    do
      if argv.count /= 2
        then
          io.error.puts ("Usage: ")
          io.error.puts (argv.item(0))
          io.error.puts ("portnumber")
        else
          create soc1.make_server_by_port (argv.item(1).to_integer)
          from
            soc1.listen (5) count := 0
            until
              count := 5
              loop
                process
                  count := count + 1
                end
                soc1.cleanup
              end
          soc1.cleanup
        end
    end
rescue
  soc1.cleanup
end
end

EiffelNet: Sockets and communication modes
```

**Example: Eiffel Server (TCP), contd.**

```
process is
  local
    our_new_list: OUR_MESSAGE
    do
      soc1.accept
      soc2 := soc1.accepted
      our_new_list := retrieved (soc2)
      from
        our_new_list.start
        until
          our_new_list.after
          loop
            io.puts (our_new_list.item)
            our_new_list.forth
            io.new_line
          end
        our_new_list.extend ("Server message. %N")
        our_new_list.general_store (soc2)
        soc2.close
      end
    end
end
```

The message exchanged between server and client is a linked list of strings

- the server obtains access to the server
- accept - ensures synchronization to with the client
- accept - creates a new socket which is accessible through the attribute accepted
- the accepted value is assigned to soc2 - this makes soc2 available to accept connections with other clients

```
extends the message received from the client
```

```
The extended message received from the client
```

```
Sends the extended message back to the client
```

```
Closes the socket
```

```
The message exchanged between server and client is a linked list of strings
```

```
the server obtains access to the server
```

```
accept - ensures synchronization to with the client
```

```
accept - creates a new socket which is accessible through the attribute accepted
```

```
the accepted value is assigned to soc2 - this makes soc2 available to accept connections with other clients
```
Example: Eiffel Client (TCP - stream socket)

1. Creates a socket and setup the communication
2. Builds the list of strings
3. Sends the list of strings to the server
4. Receives the message from the server
5. Prints the content of the received message
6. Closes the open socket and free the corresponding resources

Example: Eiffel Client (TCP), continued

Builds the list of strings 'our_list' for transmission to the server

Example: Eiffel Server (UDP - datagram socket)

1. Creates and write commands
2. Attach them to a poller
3. Set up the poller for execution

Example: Eiffel Server (UDP), continued

Creates network datagram socket bound to a local address with a specific port

Monitors the sockets for the corresponding events and executes the command associated with each event that will be received.
Example: Eiffel Client (UDP - datagram socket)

```eiffel
class OUR_DATAGRAM_CLIENT
create
make
feature
make (argvc: ARRAY [STRING]) is
local
soc: NETWORK_DATAGRAM_SOCKET
readcomm: DATAGRAM_READER
writecomm: CLIENT_DATAGRAM_WRITER
end
if argvc.count > 3 then
io.error.putstring ("Usage: ")
oio.error.putstring (argvc.item(0))
oio.error.putstring ("hostname portnumber")
else
create soc
create readcomm
create writecomm
create readcomm.make (soc)
create writecomm.make (soc)
create soc.make_read_only (readcomm)
create soc.make_write_only (writecomm)
soc.close
end
end

1. Create read and write commands
2. Attach them to a poller
3. Set up the poller for execution

Command executed in case of a read
Command executed by the client when the socket "is ready for writing"

Create a datagram socket connected to 'hostname' and 'port'

Creates poller with multi-event polling

1. Creates a read command which it attaches to the socket
2. Enters the read command into the poller
3. Creates a write command which it attaches to the socket
4. Enters the write command into the poller
```

Example: Eiffel Command class (UDP)

```eiffel
class SERVER_DATAGRAM_WRITER
create
make
feature
active_medium: NETWORK_DATAGRAM_SOCKET
execute (arg: ANY) is
local
sen_pack: DATAGRAM_PACKET
i: INTEGER
end
from i := 0 until i > 9 loop
sen_pack.put_element ("a")
i := i + 1
end
sen_pack.set_packet_number (1)
active_medium.send (sen_pack, 0)
end
end

print all the characters from the packet

Prints the packet number of the packet
```

Example: Eiffel Command class (UDP), contd.

```eiffel
class SERVER_DATAGRAM_WRITER
create
make
feature
active_medium: NETWORK_DATAGRAM_SOCKET
execute (arg: ANY) is
local
sen_pack: DATAGRAM_PACKET
i: INTEGER
end
from i := 0 until i > 9 loop
sen_pack.put_element ("a")
i := i + 1
end
sen_pack.set_packet_number (2)
active_medium.send (sen_pack, 0)
end
end

Command executed by the server when the socket 'is ready for writing'
```

```eiffel
class CLIENT_DATAGRAM_WRITER
create
make
feature
active_medium: NETWORK_DATAGRAM_SOCKET
execute (arg: ANY) is
local
sen_pack: DATAGRAM_PACKET
char: CHARACTER
end
from char := 'a' until char > 'j' loop
sen_pack.put_element (char)
char := char.next
end
sen_pack.set_packet_number (1)
active_medium.send (sen_pack, 0)
end
end

Receive a packet with 10 characters
```

Example: Eiffel Command class (UDP)

```eiffel
class CLIENT_DATAGRAM_WRITER
create
make
feature
active_medium: NETWORK_DATAGRAM_SOCKET
execute (arg: ANY) is
local
sen_pack: DATAGRAM_PACKET
char: CHARACTER
end
from char := 'a' until char > 'j' loop
sen_pack.put_element (char)
char := char.next
end
sen_pack.set_packet_number (1)
active_medium.send (sen_pack, 0)
end
end

Command executed by the client when the socket 'is ready for writing'
```

```eiffel
class OUR_DATAGRAM_READER
inherit
POLL_COMMAND
create
make
feature
active_medium: NETWORK_DATAGRAM_SOCKET
execute (arg: ANY) is
local
rec_pack: DATAGRAM_PACKET
i: INTEGER
end
from i := 0 until i > 9 loop
sen_pack.put_element (rec_pack.packet_number (i))
i := i + 1
end
end
end

Receive a packet with 10 characters
```

Example: Eiffel Command class (UDP)

```eiffel
class OUR_DATAGRAM_READER
inherit
POLL_COMMAND
create
make
feature
active_medium: NETWORK_DATAGRAM_SOCKET
execute (arg: ANY) is
local
rec_pack: DATAGRAM_PACKET
i: INTEGER
end
from i := 0 until i > 9 loop
sen_pack.put_element (rec_pack.packet_number (i))
i := i + 1
end
end
end

Receive a packet with 10 characters
```

Example: Eiffel Command class (UDP)

```eiffel
class OUR_DATAGRAM_READER
inherit
POLL_COMMAND
create
make
feature
active_medium: NETWORK_DATAGRAM_SOCKET
execute (arg: ANY) is
local
rec_pack: DATAGRAM_PACKET
i: INTEGER
end
from i := 0 until i > 9 loop
sen_pack.put_element (rec_pack.packet_number (i))
i := i + 1
end
end
end

Receive a packet with 10 characters
```

Example: Eiffel Command class (UDP)

```eiffel
class OUR_DATAGRAM_READER
inherit
POLL_COMMAND
create
make
feature
active_medium: NETWORK_DATAGRAM_SOCKET
execute (arg: ANY) is
local
rec_pack: DATAGRAM_PACKET
i: INTEGER
end
from i := 0 until i > 9 loop
sen_pack.put_element (rec_pack.packet_number (i))
i := i + 1
end
end
end

Receive a packet with 10 characters
```