Network programming

Communication paradigm
**Sockets API**

- API:et for communication between applications
  - Applications create sockets and later use them for sending and receiving data
  - Communication peer identified with:
    - IP address
      - A 32-bit binary number (IPv4)
      - Identifies destination node
    - Port number
      - 16-bit binary number
      - Identifies application process at each side
  - A session is defined by the 5-tuple:
    - IP addresses of the endpoints
    - Port numbers of the endpoints
    - Transport protocol
Services provided by the API

- General
  - Adressing an application through IP/port numbers
- TCP
  - Connection oriented
  - Reliable delivery
  - Byte stream
- UDP
  - Connectionless
  - Unreliable delivery
  - Message oriented

TCP (Java)

Server (runs at node)

1. Create socket, port x
   ```
   welcomeSocket = ServerSocket()
   ```
2. Wait for connection
   ```
   connectionSocket = welcomeSocket.accept()
   ```
3. Read query from `connectionSocket`
4. Write answer to `connectionSocket`
5. Close `connectionSocket`

Client

1. Create socket, connect to node, port x
   ```
   clientSocket = Socket()
   ```
2. Send query to `clientSocket`
3. Read answer from `clientSocket`
4. Close `clientSocket`

Data sent and received as a byte stream,
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();
    }
}
```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int sock;
struct sockaddr_in sa;

bzero((char *)&sa, sizeof(sa));
sa.sin_family = AF_INET;
sa.sin_addr.s_addr = inet_addr(...);
sa.sin_port = ...;

if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    error("Can't create TCP socket");

if (connect(sock, (struct sockaddr *)&sa, sizeof(sa)) < 0)
    error("Can't connect to server");

write(sock, ...);
read(sock, ...);
close(sock);
```
UDP (C)

Create socket

\[ s = \text{socket}(\ldots) \]

Bind address to socket

\[ \text{bind}(s,\ldots) \]

Wait for data

\[ \text{recvfrom}(s,\ldots,caddr) \]

Send reply

\[ \text{sendto}(caddr,\ldots) \]

Create socket

\[ c = \text{socket}(\ldots) \]

Bind address to socket

\[ \text{bind}(c,\ldots) \]

Send query

\[ \text{sendto}(c,\ldots) \]

Read reply

\[ \text{recvfrom}(c,\ldots) \]

Some important things

- Always translate byte ordering
  - htons, htonl, ntohs, ntohl
- DNS lookups
  - gethostbyname, or rather: getaddrinfo
- Always capture return values (and handle them!)
  - Some system calls are non-blocking
- Always close a TCP connection
  - Avoid pending connection states from old connections
- Some OS:s does not permit reusing ports
  - Can be resolved with a socket option
Socket Options

• Used to control socket behavior
  – OS/Protocol stack specific
  – Generic options
  – Protocol specific options

• Types
  – Boolean flags
  – Complex types
    • int
    • Timeval
    • in_addr
    • Sockaddr
    • ...

• Some options are read-only

Using socket options

• #include <sys/socket.h>
• getsockopt() and setsockopt()

• Arguments:
  – int sockfd
  – int level
    • Specifies if option is general or protocol-specific
    • SOL_SOCKET or protocol number
  – int optname
    • Predefined constants
  – void *optval
  – socklen_t optlen
(a few) Generic socket options

- **SO_BROADCAST**
  - Defines whether broadcast is possible or not
- **SO_DONTROUTE**
  - Bypass normal routing, used by routing daemons
- **SO_ERROR**
  - Read-only option similar to *errno*
- **SO_KEEPALIVE**
  - Used by TCP to keep connection up in case of low traffic
- **SO_LINGER**
  - Controls ACK waiting time at close
- **SO_OOBINLINE**
  - Enable OOB data to be sent
- **SO_RCVBUF, SO_SNDBUF**
  - Controls advertised window and sending buffer
- **SO_REUSEADDR**
  - Enables bind()'ing to address that are already in use

(a few) Protocol-specific options

- **IP_HDRINCL**
  - Used when building own IP headers
- **IP_TOS, IP_TTL**
  - Used to set the TOS/TTL field of the IP header
- **TCP_KEEPALIVE**
  - Specify TCP idle time when **SO_KEEPALIVE** is enabled
- **TCP_MAXSEG**
  - Specifying the maximum segment size in TCP
- **TCP_NODELAY**
  - Disable the Nagle algorithm and delayed ACK:s in TCP
OOB Data in TCP connection

- Delivery of high priority data using *urgent mode*
  - `send(..., MSG_OOB)`
- At the receiver:
  - Generates a `SIGURG` signal
  - If using `select()`, an exception is thrown
  - `read(..., MSG_OOB)`
- Typical usage
  - Heartbeat mechanism
  - Bypasses congestion control
    - In some OS:es even flow control
- Enabled with `SO_OOBINLINE` option
  - Usage can be disabled by administrator in some OS:es

(BSD-style) Raw IP sockets

- Features
  - Can bypass the transport protocols
  - Can provide packet-oriented data delivery
    - I.e., TCP becomes "packet-oriented" (kind of)
  - Supports custom-made IP headers (read: spoofing)
    - Using `IP_HDRINCL` socket option
  - HUGE security problem, typically disabled
Name/Address Conversion

- `gethostbyname()` and `gethostbyaddr()`
  - Protocol dependent
  - Not part of `sockets` library
- `getaddrinfo()`
  - Protocol independent
  - Part of `sockets` library
  - Better adapted to future with both IPv4 and IPv6
  - Re-entrant function (important for threaded applications)
  - `getaddrinfo()` replaces both `gethostbyname()` and `getservbyname()`

```c
int getaddrinfo(const char *hostname, const char *service, const struct addrinfo *hints, struct addrinfo **result);
```

```c
struct addrinfo {
    int     ai_flags;
    int     ai_family;
    int     ai_socktype;
    int     ai_protocol;
    size_t  ai_addrlen;
    char    *canonname;
    struct sockaddr *ai_addr;
    struct addrinfo *ai_next;
};
```

- `ai_flags`:
  - Used in `socket()`
- `ai_family`:
  - Used in: `bind()`, `connect()`, `sendto()`
- `ai_socktype`:
  - Linked list (if multiple results)
Writing a concurrent server (C)

• Design alternatives
  – One child/client
  – One thread/client
  – Preforking processes
  – Prethreading
• Important to understand the options
• Test before you decide what to use

One child/client

• Traditional solution:
  – After `accept()`/`recvfrom()`, call `fork()`
  – Each process needs only a few sockets.
  – Small requests can be serviced in a small amount of time.
• Parent process needs to clean up!!!
  – call `wait()`
One thread/client

- Almost like using `fork()`
  - Call `pthread_create()` instead
- Less overhead when sharing data with other processes
  - Must be done carefully, using `pthread_mutex`

Preforking and Prethreading

- The initial server
  - calls `socket()` and `bind()`,
  - `fork()` or `pthread_create()` a number of children
- Each process is an iterative server
- All children call `accept()`
  - Next incoming connection handled to a child
- Number of children is a performance tradeoff
- Preforking: Server doesn’t bother about clients
  - Only manages the children
- Prethreading: Server can do all the `accept()`’s
  - Hand over incoming connection to an existing thread
What is the best alternative?

- Consider
  - Number of simultaneous clients
  - Transaction size (incl. variability)
  - Available system resources

I/O Multiplexing

- Alternatives
  - Nonblocking I/O
    - Use `fcntl()` to set `O_NONBLOCK`
  - Alarm and signal handlers
    - Interrupts slow system calls
  - Multiple processes/threads
    - As described
  - Functions that support multi-input checking
### Non-blocking I/O

```c
int flags;
flags = fcntl(sock, F_GETFL, 0);
fcntl(sock, F_SETFL, flags | O_NONBLOCK);

while (! done) {
    if ( (n=read(STDIN_FILENO,...)<0))
        if (errno != EWOULDBLOCK)
            /* ERROR */
        else write(tcpsock,...)
    
    if ( (n=read(tcpsock,...)<0))
        if (errno != EWOULDBLOCK)
            /* ERROR */
        else write(STDOUT_FILENO,...)
}
```

- Harder for OS to put process to sleep
  - Usually happens during blocking I/O requests
  - Process will consume most available CPU time

### Alarms

```c
signal(SIGALRM, sig_alrm);
alarm(MAX_TIME);
read(STDIN_FILENO,...);
...
signal(SIGALRM, sig_alrm);
alarm(MAX_TIME);
read(tcpsock,...);
...
```

- Increases the response time
  - How is `MAX_TIME` chosen properly?
Multi-input checking with `select()`

- Blocking I/O on a set of descriptors
  - Files, Devices, Sockets...
- Create an empty `fd_set`
  - `FD_ZERO(fd_set *fdset);`
- Add descriptors to be monitored
  - `FD_SET(int fd, fd_set *fdset);`
- Call `select()`
- Check the set when `select()` returns
  - `FD_ISSET(int fd, fd_set *fdset);`