Sockets and Parallel Computing

CS439: Principles of Computer Systems
April 11, 2018
Last Time

• Introduction to Networks
• OSI Model (7 layers)
  – Layer 1: hardware
  – Layer 2: Ethernet (frames)
    • SAN, LAN, WAN
  – Layer 3: IP (packets)
  – Layer 4: TCP, UDP (segments)
• Network communication
  – Protocols, naming, routing
• Client/Server transactions
• TCP/IP congestion control mechanisms
Transferring Data Over an Internet

LAN1

1. Data

internet packet

2. Data PH FH1

LAN1 frame

3. Data PH FH1

LAN2

4. Data PH FH1

5. Data PH FH2

Router

6. Data PH FH2

7. Data PH FH1

8. Data

PH: Internet packet header
FH: LAN frame header
Transferring Data Over an internet: Text Description

- Example has 2 LANs connected by a router
- Terminology:
  - PH: Internet packet header
  - FH: LAN frame header
- Steps to transfer:
  1. Data is sent from Host A’s client application to protocol software
  2. The PH and FH1 are appended to the data and sent along to the LAN1 adapter
  3. LAN1 adapter sends Data + PH + FH1 along to the router’s LAN1 adapter
  4. Data + PH + FH1 sent to router’s protocol software, which strips off FH1 and creates FH2
  5. Data + PH + FH2 sent to router’s LAN2 adapter
  6. Data + PH + FH2 sent to LAN2 adapter in LAN2
  7. Data + PH + FH2 sent along to protocol software where the PH and FH2 are stripped off
  8. Data goes along to Host B’s server application
Today’s Agenda

• Accessing the Network
  – Sockets
    • Client-Side Programming
    • Server-Side Programming

• Parallel Computing
  – Description of Parallel Computing
  – Parallel Programming Models
    • Shared Memory
    • Message Passing
Internet Connections

• Clients and servers communicate by sending streams of bytes over connections

• A *socket* is an endpoint of a connection
  – Underlying basis for all Internet applications
  – Created in the early 80s as a part of the original Berkeley distribution of UNIX that contained an early version of the Internet protocols
  – Socket address is an IP address:port pair

• A *port* is a 16-bit integer that identifies a process:
  – *Ephemeral port*: Assigned automatically by client kernel when client makes a connection request.
  – *Well-known port*: Associated with some *service* provided by a server (e.g., port 80 is associated with Web servers)
Anatomy of an Internet Connection

**Client socket address**
128.2.194.242:51213

**Server socket address**
208.216.181.15:80

**Connection socket pair**
(128.2.194.242:51213, 208.216.181.15:80)

- **Client host address**
  128.2.194.242

- **Server host address**
  208.216.181.15

51213 is an ephemeral port allocated by the kernel

80 is a well-known port associated with Web servers
Sockets: Implementation Details

• What is a socket?
  – To the kernel, a socket is an endpoint of communication
  – To an application, a socket is a file descriptor that lets the application read/write from/to the network
    • Remember: All Unix I/O devices, including networks, are modeled as files

• Clients and servers communicate with each other by reading from and writing to socket descriptors

• The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors
Sockets Interface

1. Start server
   - Server
   - getaddrinfo
   - socket
   - bind
   - listen
   - accept

2. Start client
   - Client
   - getaddrinfo
   - socket
   - connect

Connection request

Client / Server Session

3. Exchange data
   - Await connection request from next client
   - Connection
   - recv()
   - send()

4. Disconnect client
   - close

5. Drop client
   - recv()
   - close

open_clientfd

open_listeningfd

EOF
**Sockets Interface**

**Client**
- `getaddrinfo`
- `socket`
- `connect`
- `rio_readlineb`
- `rio_writen`
- `close`

**Server**
- `getaddrinfo`
- `socket`
- `bind`
- `listen`
- `accept`
- `rio_readlineb`
- `rio_writen`
- `close`

- `open_clientfd` → `connect` → `rio_readlineb` → `rio_writen` → `close`
- `open_listenfd` → `bind` → `listen` → `accept` → `rio_readlineb` → `rio_writen` → `close`

**Connection Flow**
- Client initiates the connection with `open_clientfd`.
- Server listens on the `open_listenfd`.
- Client sends a `Connection request` to accept.
- Server accepts the connection and sends `rio_readlineb`.
- Client reads the data and sends `rio_writen`.
- Server reads the data and sends `rio_readlineb`.
- Both parties close the connection on `EOF`.

**Await Connection**
- Server waits for the next client connection.

Interfacing with DNS

Functions for retrieving host entries from DNS:

getaddrinfo(): query key is a DNS domain name, returns:

```c
struct addrinfo {
    int ai_flags;
    int ai_family;  /* desired address family */
    int ai_socktype;  /* desired Layer 4 protocol */
    int ai_protocol;
    socklen_t ai_addrlen;
    struct sockaddr *ai_addr;
    char *ai_canonname;
    struct addrinfo *ai_next;
};
```

getnameinfo(): query key is an IP address, returns a sockaddr struct (more soon)
Using `getaddrinfo()`

```c
int clientfd; /* socket descriptor */

/* setup information*/
struct addrinfo hints;
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_UNSPEC; /* Allow IPv4 or IPv6 */
hints.ai_socktype = SOCK_DGRAM; /* Datagram socket */

/* fill in the server's IP address and port */
struct addrinfo *result = NULL; /* DNS host entry */
int s = 0;
if ((s = getaddrinfo(hostname, port, &hints, &result)) != 0)
  return -1;
```

Note: This is untested code.
**Sockets Interface**

**Client**
- `getaddrinfo`
- `socket`
- `connect`
- `send()`
- `recv()`
- `close`

**Server**
- `getaddrinfo`
- `socket`
- `bind`
- `listen`
- `accept`
- `recv()`
- `send()`
- `close`

**Steps**
1. **open_clientfd**
   - `open_clientfd`

2. **Await connection request**
   - `Connection request`
   - `accept`

3. **Client / Server Session**
   - `send()`
   - `recv()`
   - `close`

4. **Server**
   - `recv()`
   - `close`

**Other Notes**
- `open_listenfd`
- `getaddrinfo`
Client: `socket(2)`

`socket(2)` creates a socket descriptor on the client

- Allocates and initializes some internal data structures
- `AF_UNSPEC`: indicates that the socket is associated with Internet protocols but either IPv4 or IPv6 is fine
- `SOCK_STREAM`: selects a reliable byte stream connection
  - Bi-directional pipes
  - Gives you TCP
- `SOCK_DGRAM` results in UDP

```c
int clientfd; /* socket descriptor */

if ((clientfd = socket(AF_UNSPEC, SOCK_STREAM, 0) <0))
    return -1; /* check errno for cause of error */
...
```
Socket Implementation in the OS

• Each socket fd has associated socket structure with:
  – Send and receive buffers
  – Queues of incoming connections (on listen socket)
  – A *protocol control block* (PCB)
  – A *protocol handle*

• PCB contains protocol-specific information, such as:
  – Pointer to IP Transmission Control Block with source/destination IP address and port
  – Information about received packets and position in stream
  – Information about unacknowledged sent packets
  – Information about timeouts
  – Information about connection state (setup/teardown)
Sockets Interface

Client

getaddrinfo
socket
connect
send()
recv()
close

Server

getaddrinfo
socket
bind
listen
accept
recv()
send()
recv()
close

open_clientfd

Connection request

Await connection request from next client

open_listenfd

Client / Server Session
Finally the client creates a connection with the server
– Client process blocks until the connection is created
– After resuming, the client is ready to begin exchanging messages with the server via Unix I/O calls (typically send/recv) on descriptor clientfd

```c
int clientfd; /* socket descriptor */
struct addrinfo *result; /* DNS host entry initialized in * call to getaddrbyinfo()*/
typedef struct sockaddr SA; /* generic sockaddr */

... /* Establish a connection with the server */
if (connect(clientfd, result->ai_addr, result->ai_addrlen) < 0)
    return -1;
```
**Server: bind(2)**

`bind()` associates the socket descriptor with the socket address (created similarly to that of the client)

```
int listenfd;                   /* listening socket */
struct addrinfo *result;       /* DNS host entry initialized in *
                               * call to getaddrbyinfo()*/
...
/* listenfd will be an endpoint for all requests to port *
on any IP address for this host */
if (bind(listenfd, rp->ai_addr, rp->ai_addrlen) < 0)
    return -1;
```
Server: `setsockopt(2)`

- Avoid problems with `bind()` ... Give the socket some attributes

```c
... /* Eliminates "Address already in use" error from bind(). */
if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
(const void *)&optval, sizeof(int)) < 0)
    return -1;
```

- Handy trick that allows us to rerun the server immediately after we kill it
  - Otherwise we would have to wait about 15 seconds
  - Eliminates “Address already in use” error from `bind()`

- Strongly suggest you do this for all your servers to simplify debugging
Sockets Interface

Client:
- getaddrinfo
- socket
- connect
- send()
- recv()
- close

Server:
- getaddrinfo
- socket
- bind
- listen
- accept
- recv()
- send()
- close

open_clientfd
open_listendid

Connection request from next client

Await connection request from next client

Client / Server Session
Server: `listen(2)`

- `listen()` indicates that this socket will accept connection (`connect`) requests from clients
  - Kernel assumes an active socket on the client end by default
- `LISTENQ` is a constant indicating how many pending requests are allowed

```c
int listenfd; /* listening socket */
...
/* Make it a listening socket ready to accept connection requests */
if (listen(listenfd, LISTENQ) < 0)
    return -1;
```

- We’re finally ready to enter the main server loop that accepts and processes client connection requests.
Sockets Interface

Client
- `getaddrinfo`
- `socket`
- `connect`
- `send()`
- `recv()`
- `close`

Server
- `getaddrinfo`
- `socket`
- `bind`
- `listen`
- `accept`
- `send()`
- `recv()`
- `close`

Connection request

Await connection request from next client
Server: accept(2)

• `accept()` blocks waiting for a connection request

```c
int listenfd; /* listening descriptor */
int connfd; /* connected descriptor */
struct sockaddr_in clientaddr;
int clientlen;

clientlen = sizeof(clientaddr);
connfd = accept(listenfd, (SA *)&clientaddr, &clientlen);
```

• `accept()` returns a connected descriptor (`connfd`) with the same properties as the listening descriptor (`listenfd`)
  – Returns when the connection between client and server is created and ready for I/O transfers
  – All I/O with the client will be done via the connected socket

• `accept()` also fills in client’s IP address
Connected vs. Listening Descriptors

• Listening descriptor
  – End point for client connection requests
  – Created once and exists for lifetime of the server

• Connected descriptor
  – End point of the connection between client and server
  – A new descriptor is created each time the server accepts a connection request from a client
  – Exists only as long as it takes to service client

• Why the distinction?
  – Allows for concurrent servers that can communicate over many client connections simultaneously
    • E.g., Each time we receive a new request, we create a new thread to handle the request
### accept Illustrated

1. **Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`**

2. **Client makes connection request by calling and blocking in `connect`**

3. **Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`**
iClicker Question

The listen() call is listening on a different port than the port the server will eventually use to send data to the client.

A. True
B. False
1. Start server

Server

1. Start server
2. Start client

Client

3. Exchange data

Client / Server Session

open_clientfd

open_listenfd

getaddrinfo

socket

bind

listen

accept

recv()

send()

recv()

send()

close

recv()

close

3. Exchange data

Await connection request from next client

4. Disconnect client
Parallel Computing
Parallel Computing: Description

• Tightly-coupled systems
• Processors share:
  – Clock
  – Memory (a single physical address space)
  – A single OS
• Examples: Multicore systems, Symmetric Multi Processor (SMP) systems (and you can make SMP multicore processors...)
Single CPU
Symmetric Multi Processor

Processor 0

Registers

L1 d-cache

L2 unified cache

L3 unified cache

ALU

Processor 3

Registers

L1 d-cache

L2 unified cache

L3 unified cache

ALU

Main memory

L1 i-cache

L1 i-cache

L3 unified cache

Bus
Multicore Processor

Processor package

Core 0
- Registers
- L1 d-cache
- L1 i-cache
- L2 unified cache
- L3 unified cache (shared by all cores)

Core 3
- Registers
- ALU
- L1 d-cache
- L1 i-cache
- L2 unified cache

Main memory
Parallel Programming
Why a different type of programming?

Sequential programs get no benefit from multiple processors

– Key property is how much communication per unit of computation.
  • The less communication per unit computation the better the scaling properties of the algorithm.

– Many applications can be (re)designed/coded/compiled to generate cooperating, parallel instruction streams

– No automated compiler/language exists to automate this “parallelization” process.
Parallel Programming

• Parallel programming involves:
  – Decomposing an algorithm into parts
  – Distributing the parts as tasks which are worked on by multiple processes simultaneously
  – Coordinating work and communication of those processes
    • Synchronization

• Parallel programming considerations:
  – Type of parallel architecture being used
  – Type of process communication used
Parallel Programming Models

Communication and synchronization based on either:

– **Shared memory**
  • Interprocess communication is implicit
  • Synchronization is explicit
  • Assume processes/threads can read & write a set of shared memory locations
  • Difficult to provide across machine boundaries

– **Message passing**
  • Interprocess communication is explicit
  • Synchronization is implicit
  • Extensible to communication in distributed systems

send(message) receive(message)
Shared Memory Programming Model

• Programs/threads communicate/cooperate via loads/stores to memory locations they share.
• Communication is therefore at memory access speed (very fast) and is implicit.
• Cooperating pieces must all execute on the same system (computer).
• OS services and/or libraries used for creating tasks (processes/threads) and coordination (semaphores/barriers/locks.)
Example Shared Memory Code

fork N processes
each process has a number, p, and computes

\[ \text{istart}[p], \text{iend}[p], \text{jstart}[p], \text{jend}[p] \]

for (s=0; s<STEPS; s++) {
    k = s&1 ; m = k^1 ;
    \text{forall}(i=\text{istart}[p]; i<=\text{iend}[p]; i++) {
        \text{forall}(j=\text{jstart}[p]; j<=\text{jend}[p]; j++) {
            a[k][i][j] = c1*a[m][i][j] + c2*a[m][i-1][j] +
            c3*a[m][i+1][j] + c4*a[m][i][j-1] +
            c5*a[m][i][j+1] ; // implicit comm
        }
    }
    \text{barrier() ;}
}
Message Passing Programming Model

• “Shared” data is communicated using send/receive services (across an external network).
• Shared data must be formatted into message chunks for distribution
  – Unlike shared memory
• Coordination is also via sending/receiving messages.
• Program components can be run on the same or different systems, so can use many of processes.
• Standard libraries exist to encapsulate messages:
  – Parasoft's Express (commercial)
  – PVM (standing for Parallel Virtual Machine, non-commercial)
  – MPI (Message Passing Interface, also non-commercial).
Message Passing Logistics: Synchronization

When do \textit{send/receive} operations terminate?

- **Blocking (aka Synchronous):**
  - Sender waits until its message is received
  - Receiver waits if no message is available

- **Non-blocking (aka Asynchronous):**
  - Send operation “immediately” returns
  - Receive operation returns if message is available or not (polling)

- **Partially blocking/non-blocking:**
  - \texttt{send()/receive()} with \textit{timeout}

- **Diagram:**

  Sender > OS Kernel
  Receiver > OS Kernel
  Sender > Receiver
  Sender > OS Kernel
  Receiver > OS Kernel
  Sender > Receiver
  How many buffers?
Limitations of Message Passing

Easy for OS, hard for programmer

– Programmer must code communication
– Programmer may have to code format conversions, flow control, error control
– No dynamic resource discovery
Summary

• Accessing the network from user code is detailed but often formulaic

• Parallel Computing: tightly-coupled systems

• Parallel Programming Models
  – Shared Memory (OpenMP)
    • Sharing is implicit, synchronization is explicit
  – Message Passing (MPI)
    • Sharing is explicit, synchronization is implicit
Announcements

• Exams are graded, scores are not entered
  – Will be returned in discussion section this week
• Project 3 due Friday
• Project 4 out Friday
  – Group registration is due Monday, 4/16
• Problem Set 9 is due Friday in section