COP4635: Systems & Networks II

Network Programming:

Sockets & Client Server

Overview

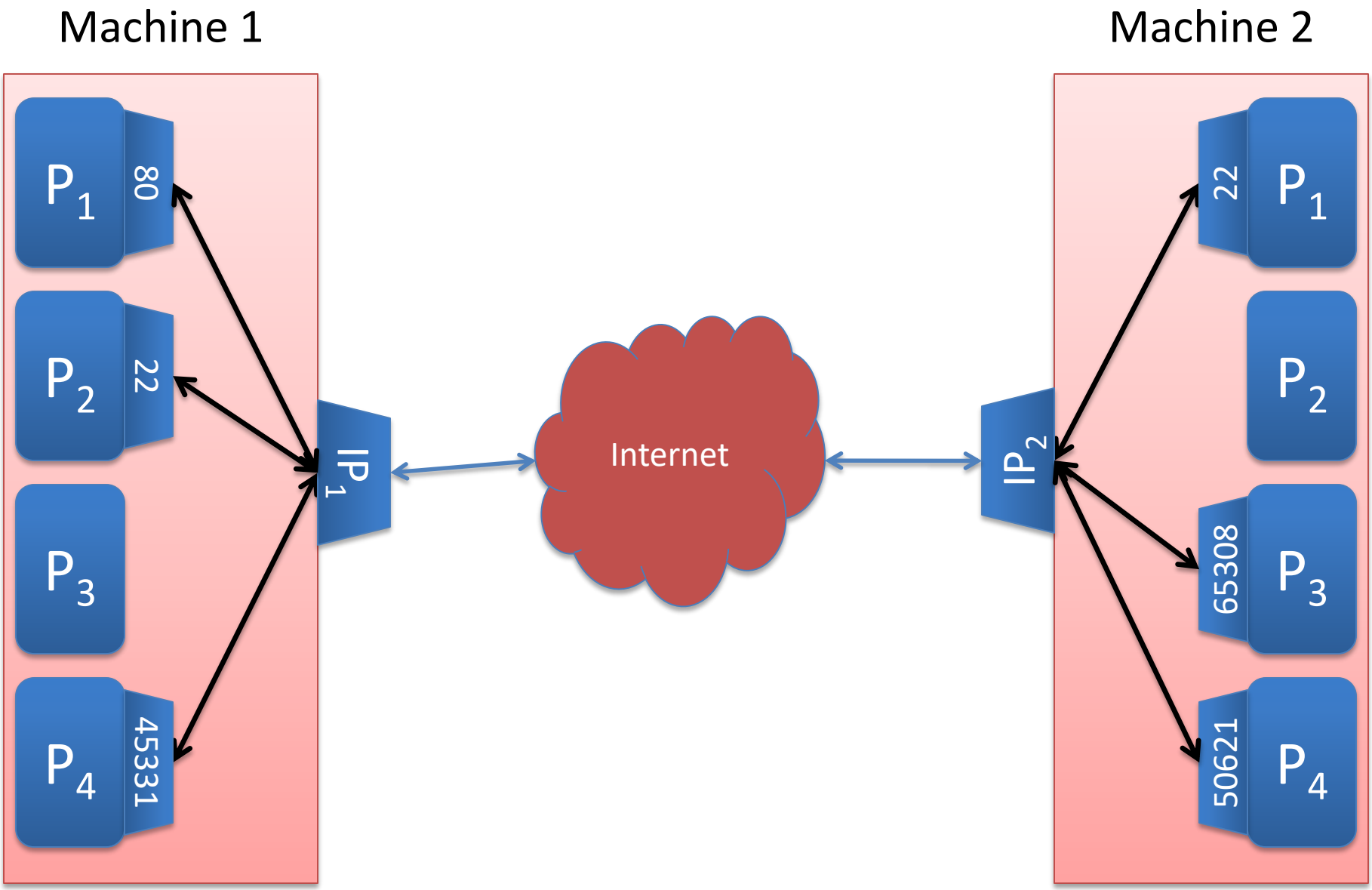
- Sockets
  - communication endpoints for processes
  - can be on different machines
  - can be used over network

- Client - Server
  - client is a process sending requests to a server
  - server is a continuously running process responding to client requests
  - client – server use connection-oriented service

Socket-Based Communication

- Socket
  - Communication endpoint
  - Commonly used for network communication
  - Composed of three parts
    - **IP Address**: Identifies machine
    - **Port**: Identifies process on machine
    - **Protocol**: Specifies communication parameters, order, etc.

Network Communication



Relevant Data Types

The following data types are needed to build sockets:

`u_char`	unsigned char (8-bit)
`u_short`	unsigned short (16-bit)
`u_long`	unsigned long (32-bit)

Internet Address

- Specifies machine on Internet
  - Assume we are using IPv4 (not IPv6)
  - 32-bit integer number
  - Usually written as dotted quad
  - Type is `u_long` (unsigned long)
  - Wrapped in structure

```
struct in_addr
{
    u_long    s_addr;
};
```

Port Numbers

Logical not physical

16-bit integer

(short)

0-1023	Well-known (reserved)
1024-49151	Registered (discouraged)
49152-65535	Dynamic/Private

7	echo		25	smtp		113	ident
11	systat		37	time		119	nntp
21	ftp		79	finger		135	RPC
22	ssh		80	http		143	imap
23	telnet		110	pop3		443	https

Protocol

- Specifies
  - Format of messages
  - Headers
  - Other communication parameters
  - A few examples:

    `PF_INET`

    `PF_INET6`

    `PF_BLUETOOTH`

    `PF_APPLETALK`

SOCKADDR

```
struct sockaddr_in
{
  u_char                  sin_len;
                            // normally
  not used
  u_short                 sin_family;
  u_short                 sin_port;
  struct in_addr          sin_addr;
  char                    sin_zero[8];
                            // normally
  not used
};
```

Socket Types

TCP

- Connection-oriented, reliable
- Also called *stream*

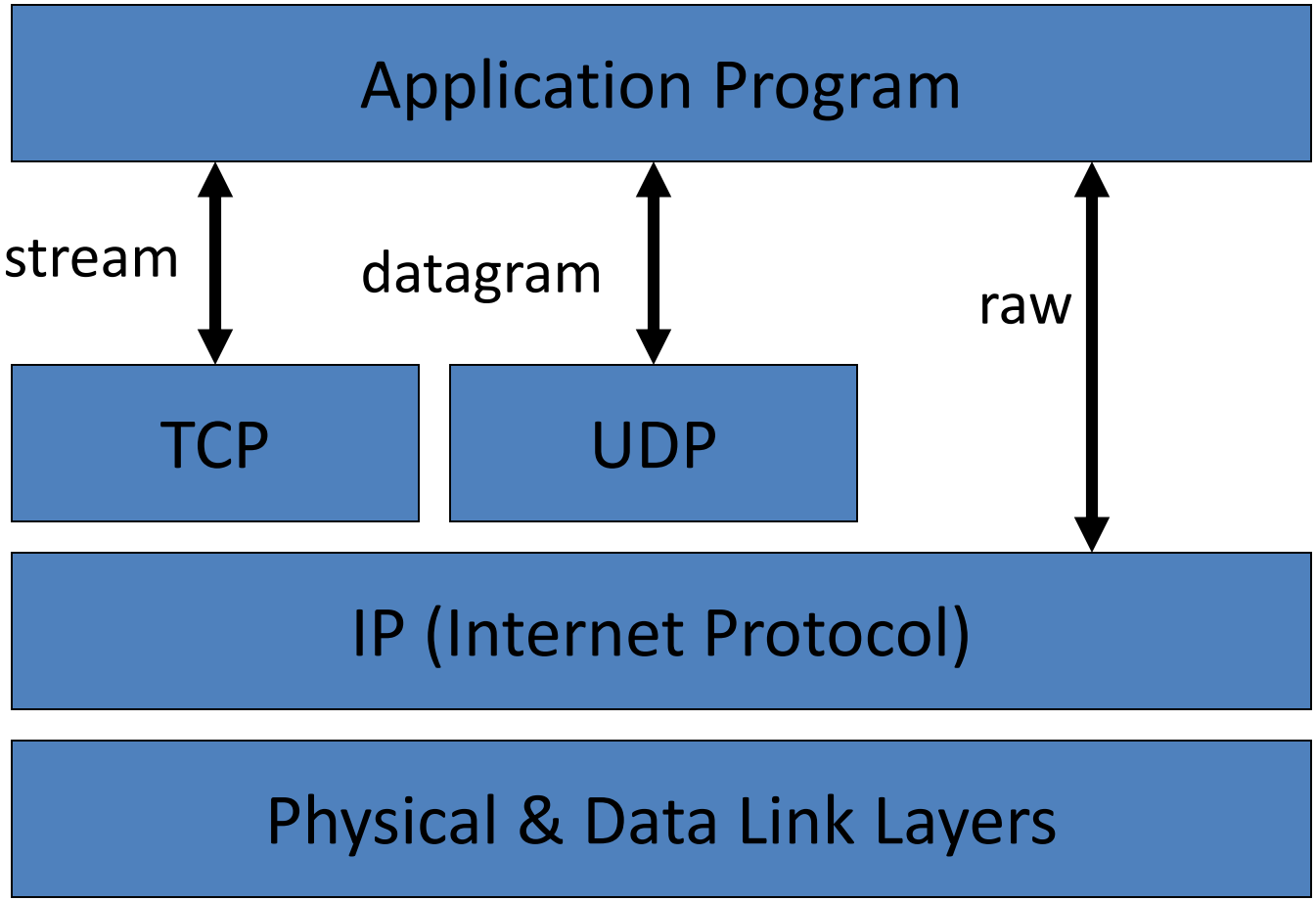
UDP

- Connectionless, unreliable
- Also called *datagram*

RAW

- Bypasses niceties of above
- Functions at IP level

Protocol Layers



Potential Problem

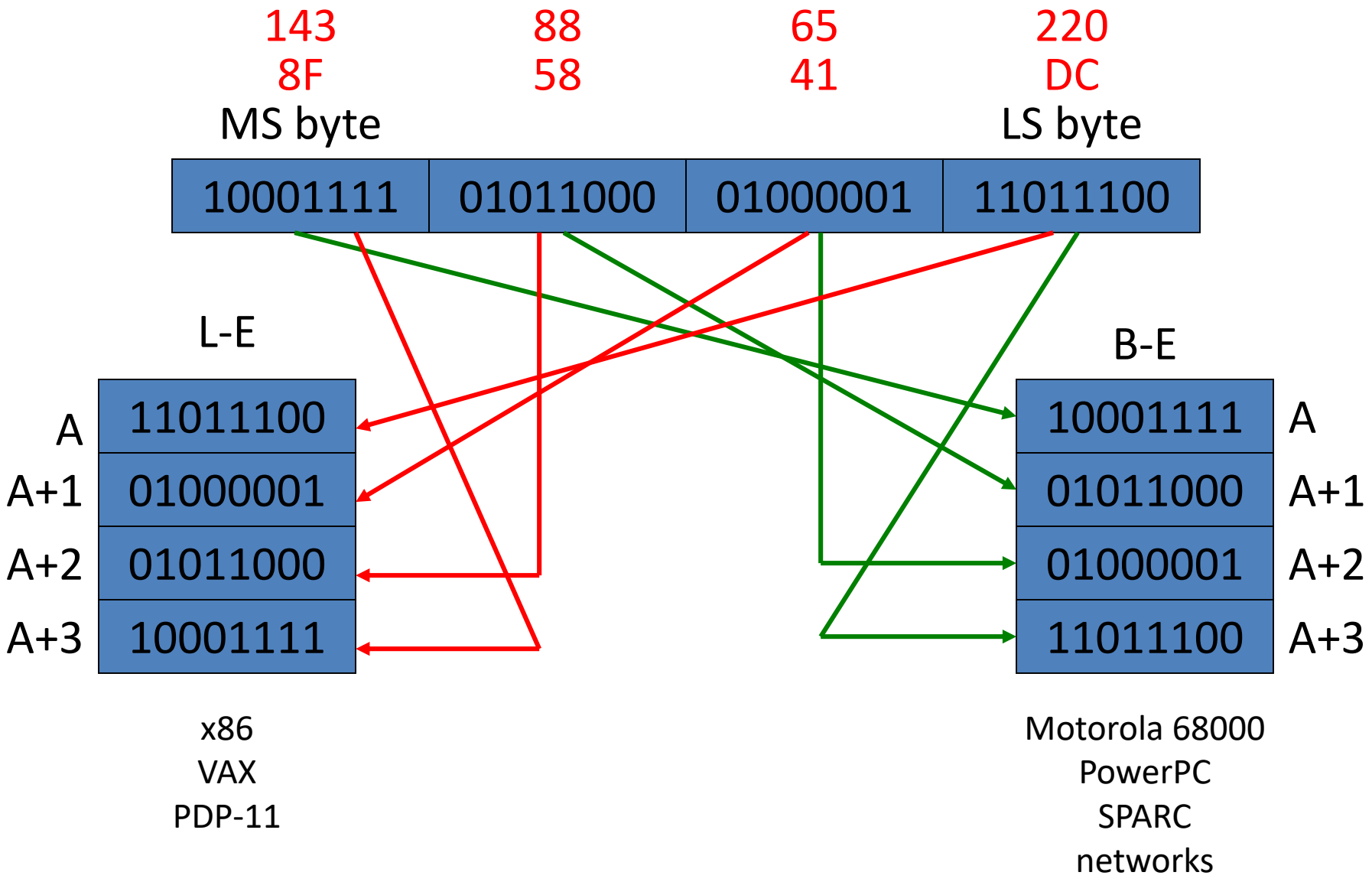
All machines are not created equal.

- 2- & 4-byte values can be stored differently.

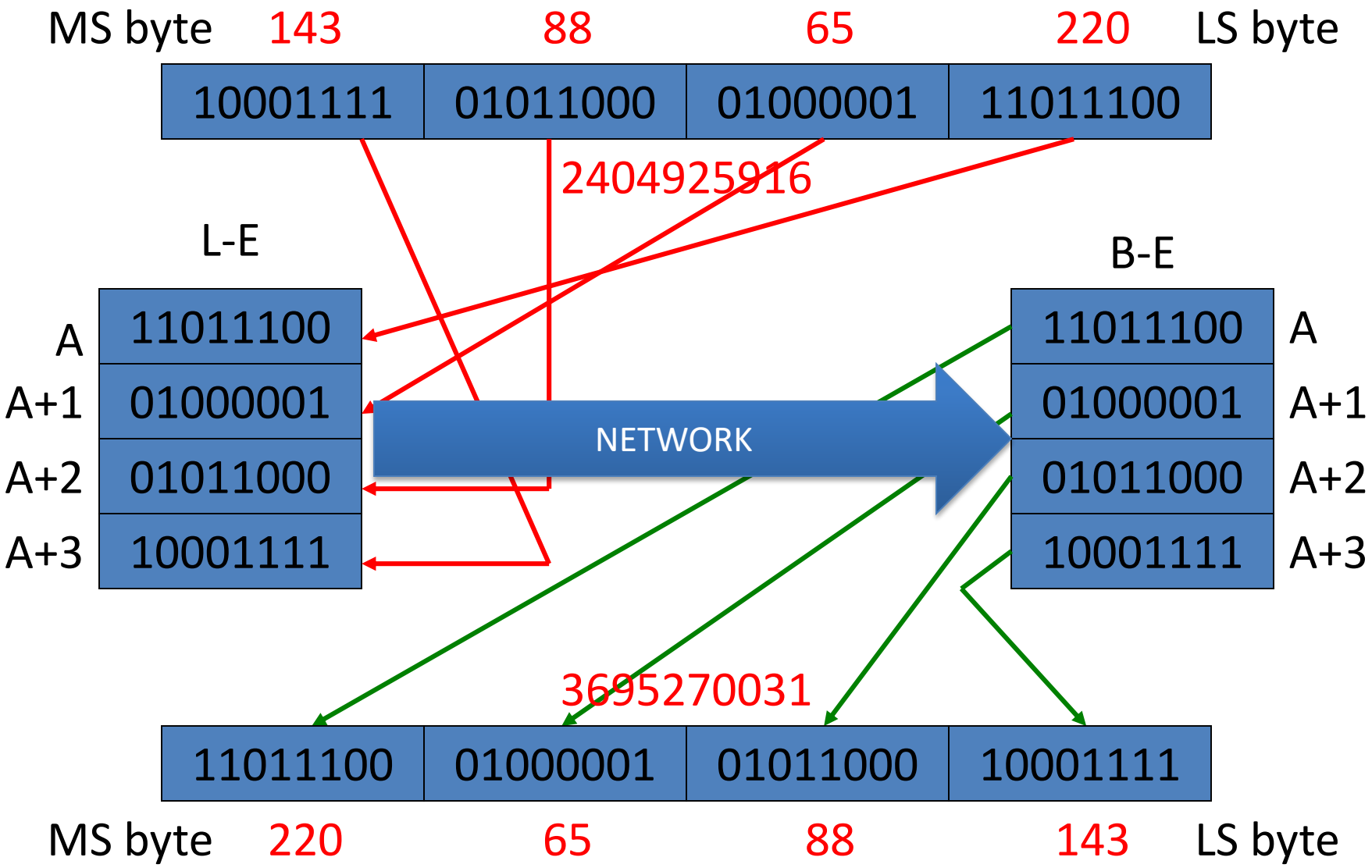
Endianness (byte order) can differ.

The very term *big-endian* comes from Jonathan Swift's satiric novel *Gulliver's Travels*, where tensions are described in Lilliput and Blefuscu: whereas royal edict in Lilliput requires cracking open one's soft-boiled egg at the small end, inhabitants of the rival kingdom of Blefuscu crack theirs at the big end (hence the name *Big-endians*).

Byte Ordering



The Problem


MS byte
143
88
65
220
LS byte
10001111
01011000
01000001
11011100
2404925916
L-E
B-E
A
A+1
A+2
A+3
11011100
01000001
01011000
10001111
NETWORK
11011100
01000001
01011000
10001111
A
A+1
A+2
A+3
3695270031
11011100
01000001
01011000
10001111
MS byte
220
65
88
143
LS byte

Communication

Machine A

- Big Endian
- Send 2-byte decimal value 821
- In binary, that's (`00000011 00110101`)

Machine B

- Little Endian
- Receive binary (`00110101 00000011`)
- In decimal, that's 13571

Network Byte Order

- Network bytes order is big-endian
- Special functions to convert to/from NBO

  `htons` host to network short

  `ntohs` network to host short

  `htonl` host to network long

  `ntohl` network to host long

- Trick
  - B-E machine: functions do nothing
  - L-E machine: functions reverse bytes
  - Always use on data
  - Don’t worry about endianness of machine

Address Transformation

Lots of functions to manipulate IP Addresses.

Refer to man pages for details and include files.

```
int inet_aton( char* cp,
               struct in_addr *inp)
```

Converts dotted-quad (`cp`) to 32-bit address in network byte order (`inp`)

```
char *inet_ntoa( struct in_addr in)
```

Converts 32-bit address in network byte order (`in`) to dotted-quad (return pointer)

Host Information

Given hostname, we need to lookup the IP address:

```
struct hostent * gethostbyname( char* name)

struct hostent
{
  char    *h_name;        /* official name */
  char   **h_aliases;     /* list of aliases */
  int      h_addrtype;    /* PF_INET */
  int      h_length;      /* length, e.g. 4 */
  char   **h_addr_list;   /* list of addr.  */
};
```

Creating a Socket

Access similar to pipe or file

Through file descriptor table

Called *socket descriptor* in this case

```
int socket(
   int domain,    /* PF_INET */
   int type,      /* SOCK_DGRAM, SOCK_STREAM */
   int protocol   /* 0 */
);
```

Return value of `-1` indicates an error

Putting It All Together

1. Create a socket
2. Get information about *dest* host.
3. Fill in *dest addr* structure.
4. Connect to remote host.
5. Send /receive data to *dest.*
6. Close connection.

Create a socket

```
socketFD = socket( AF_INET, SOCK_DGRAM, 0 );
if ( socketFD < 0 ) {
  perror( "sendUDP:socket" );
  return -1;
}
```

Get Dest Host Info

Obtaining information about a remote host:

```
hostptr = gethostbyname( argv[1] );
if ( hostptr == NULL ) {
  perror( "sendUDP:gethostbyname" );
  return -1;
}
```

Function makes a call to DNS and returns a struct called *hostent*.

Fill In Dest Addr

```
memset( &destaddr, 0,
        sizeof(struct sockaddr_in) );
destaddr.sin_family = AF_INET;
memcpy( &destaddr.sin_addr,
        hostptr->h_addr,
        hostptr->h_length );
destaddr.sin_port =
  htons( (u_short)atoi(argv[2]) );
```

Send Message

```
fgets( buffer, 256, stdin );
sendlen = strlen( buffer ) - 1;
bytes = sendto(
   socketFD,
   buffer,
   (size_t)sendlen,
   0,
   (struct sockaddr *)&destaddr,
   (socklen_t)sizeof(destaddr)
);
```

Receiver Extra Step

- Receiver must have port number.
- Delivery:
  - Packet → Machine (host) → OS → Process
- The `bind()` system call provides "name" for socket.
- Can send without a "named socket", but must have "named socket" to receive.
  - Similar to U.S. Postal Service

Using `bind`

```
int bind(
   int sockFD,
   struct sockaddr *my_addr,
   socklen_t addrlen
);
```

What Is `sockaddr` ?

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

So…

`sendto()` → `recvfrom()`

Sender needs to send to a socket

- IP Address
- Protocol Family
- Port number

Receiver needs to establish socket

- Needs to establish port number
- How?

Receiver

If 1st action is receive,

Must establish port number

And announce port number

Why?

So sender knows where to send

Otherwise

Receiver is "*out there*" somewhere

Two Ways - Static

Static binding

- **User** provides port number; some options
- Hardcoded in program

```
DA.sin_port = htons( 51664 );
```

- Provided on command-line

```
DA.sin_port =
  htons((u_short)atoi(argv[1]));
```

- Provided via user input

```
DA.sin_port =
  htons((u_short)atoi(inString));
```

Two Ways - Dynamic

Dynamic binding

- **Kernel** provides port number
- Send special “you pick” to kernel
- Value of 0 (zero) is indicator

```
DA.sin_port = 0;
```

- Kernel picks next available port number
- Stores port number internally

Example

```
int bind(
   int sockFD,
   struct sockaddr *my_addr,
   socklen_t addrlen
);

int getsockname(
   int sockFD,
   struct sockaddr *my_addr,
   socklen_t *addrlen
);
```

The `bind()` selects next available port

The `getsockname()` returns the *name* of the socket

Announcing Socket Name

```
printf("name: %s\n",
  hostptr->h_name);

printf("addr: %s\n",
  inet_ntoa(srcAddr.sin_addr));

printf("port: %d\n",
  ntohs(srcAddr.sin_port));
```

Executions

```
> ./recvUDP
name: cs-ssh1.cs.uwf.edu
addr: 143.88.64.151
port: 54464
> ./recvUDP
name: cs-ssh1.cs.uwf.edu
addr: 143.88.64.151
port: 54208
>
```

What If…

Do you need a port number to send?

Yes

Where do you get it?

Dynamically – from kernel

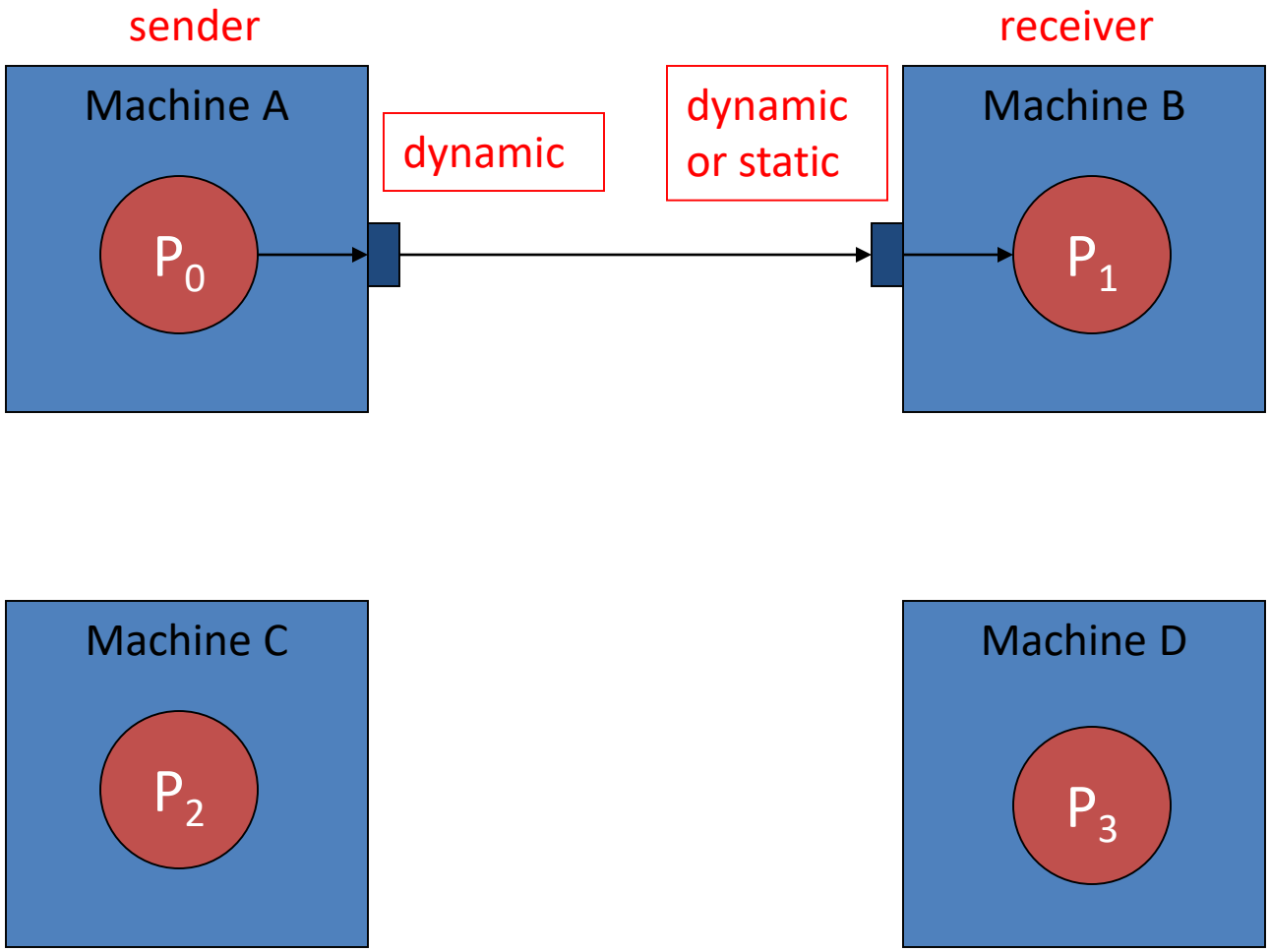
How?

`sendto()` allocates port if needed

This is second form of dynamic binding

Try using `sendto()` then `getsockname()`

Big Picture - A



Logistics for Project Work

Sit at **ONE** machine (machine A)

Open a terminal

- Terminal displays on A
- Commands run on A

Open a second terminal

- Terminal displays on A
- Commands run on A
- Use `ssh` to connect to machine B
- Terminal displays on A
- Commands run on B

What About My Files?

Multiplatform Lab & SSH servers use NFS

- Network File System
- Files stored on “file server”
- Files available on all machines
- Save on A, new file also visible on B

Compile on A, execute on B

On sse-250-061

display

sse-250-061

execute

sse-250-061

```
> ./recvUDP
name: sse-250-061.cs.uwf.edu
addr: 143.88.64.206
port: 13925
got [hello]
```

sse-250-061

sse-250-05e

```
> ssh sse-250-05e.cs.uwf.edu
> ./sendUDP sse-250-061.cs.uwf.edu 13925
Send this> hello
sent 5 bytes
>
```

Network Communication

Requires endpoints (sockets)

- Protocol (AF_INET)
- IP Address (32-bit or dotted quad)
- Port Number (16-bit; static or dynamic)

UDP

- Sends *datagrams*
- May arrive; may be lost
- May arrive in different order than sent

TCP

TCP is a protocol designed for the reliable transfer of bytes from one process to another over a network

How?

- Sequence numbers
- ACK
- Timers (what if small file?)

TCP Overview

Designed for client/server model

Server is at well-known place

Usually only one server

Clients (multiple) are anywhere

Client contacts server for *service*

Client and server communicate to facilitate service

Client and server disconnect

Server continues to wait for next client contact

TCP Notes

Client and Server have “1-to-1” connection

No other process can use connection

Connection is bidirectional

- Client can send to server
- Server can send to client

Requires buffers, segments, handshakes, …

UDP vs TCP

All (sender, receiver, client, server)

- `socket()`
- `gethostbyname( `*who*` )`

TCP must create client/server connection

Client	Server
	`bind()`
	`listen()`
`connect()`	`accept()`

TCP Server

`bind()`

- Makes receipt before send possible

`listen()`

- Expresses willingness to accept incoming connections
- Sets connection parameters (queue limit, etc.) for incoming connections

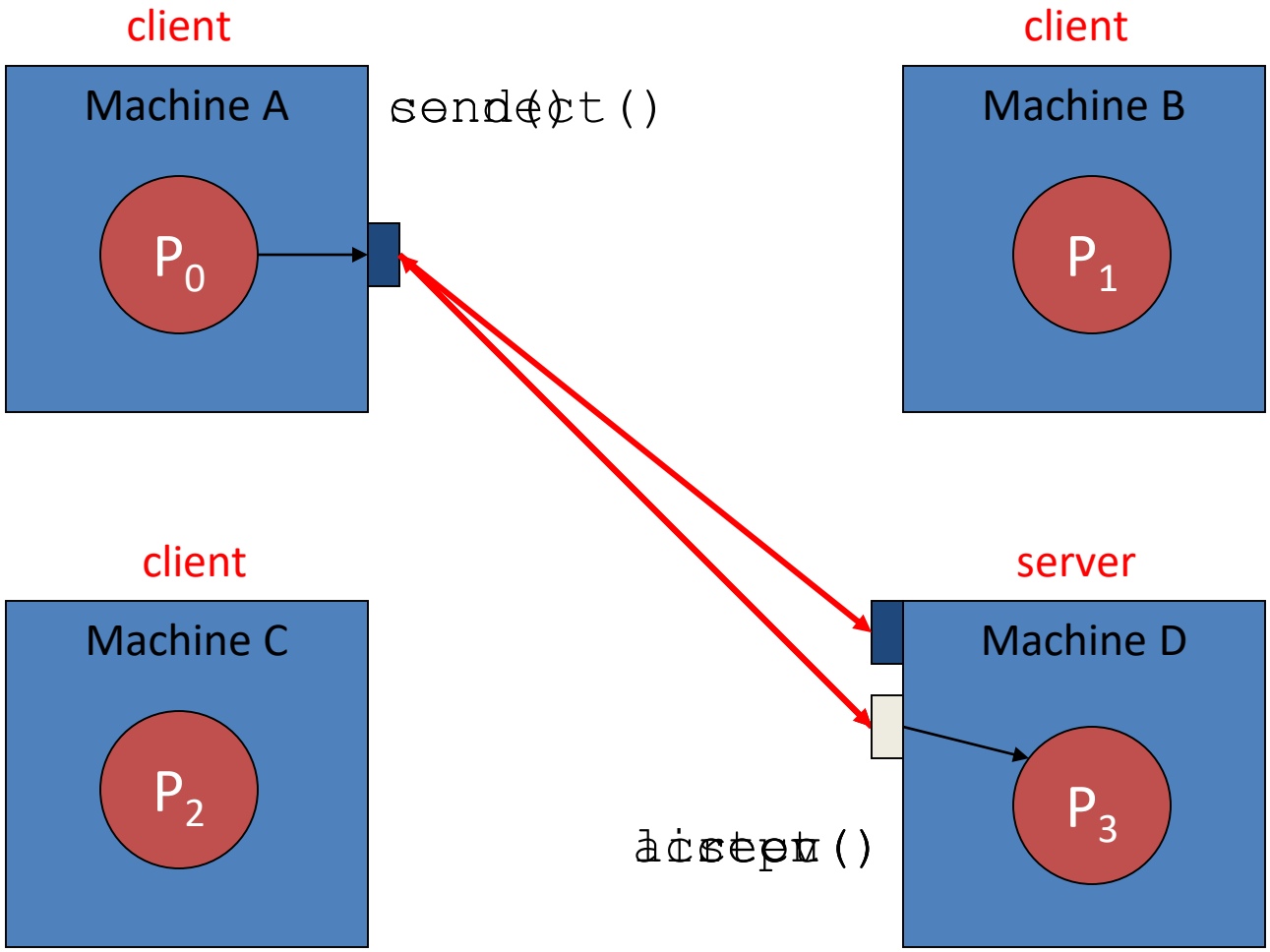
`accept()`

- Accepts a connection on a listening socket
- Returns file descriptor for newly created 1-to-1 socket

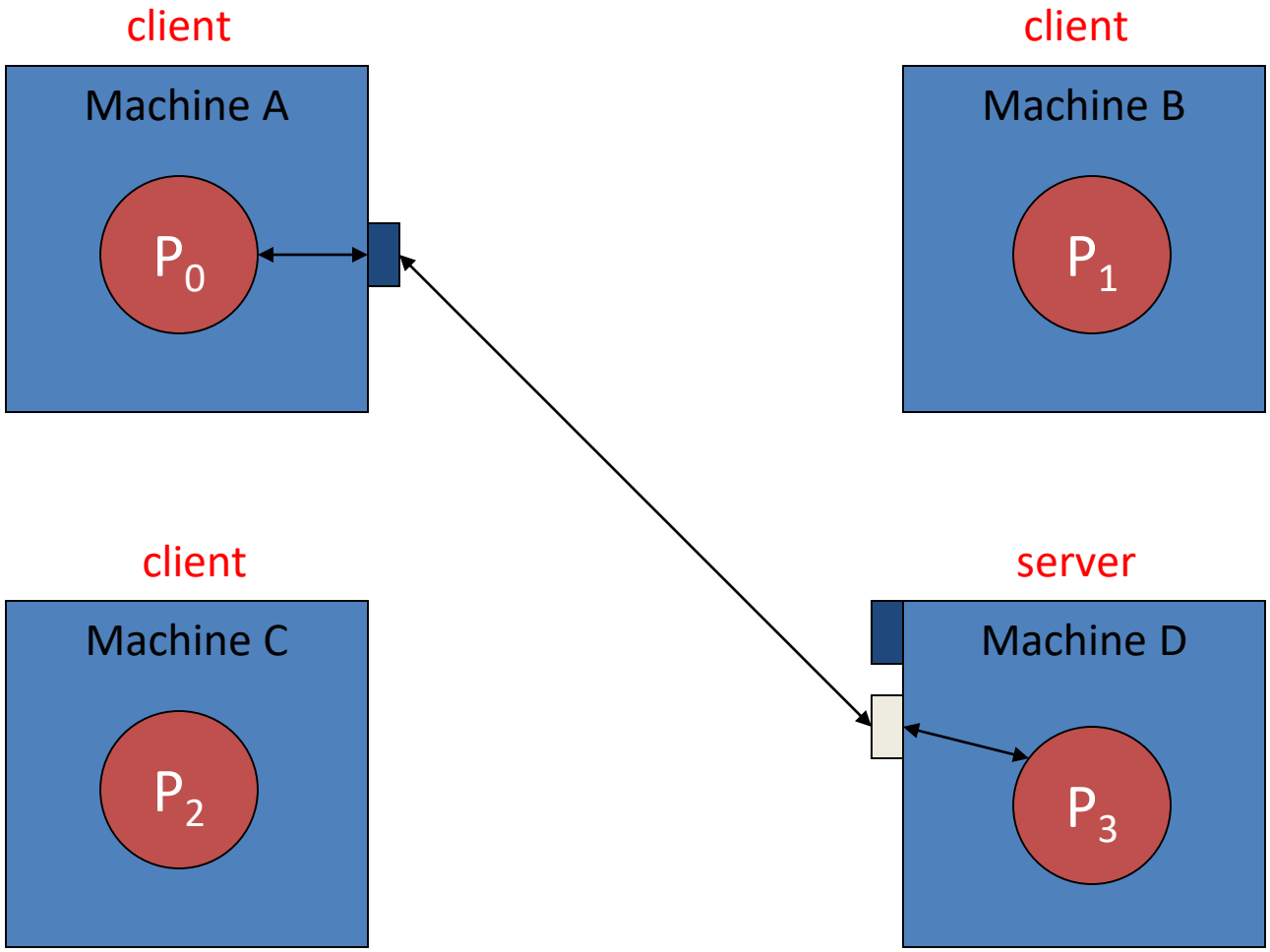
TCP Handshake

- Done by `connect()/accept()`
- Client
  - Sends "`CONN REQ`" to server socket
- Server
  - Accepts request
  - Creates new socket (dynamic port binding)
  - Randomly generates 1st S→C sequence number
  - Sends new socket & 1st S→C sequence number to client ("`CONN ACK`")
- Client
  - Accepts message
  - Randomly generates 1st C→S sequence number
  - Sends 1st C→S sequence number to server ("CONNECT")

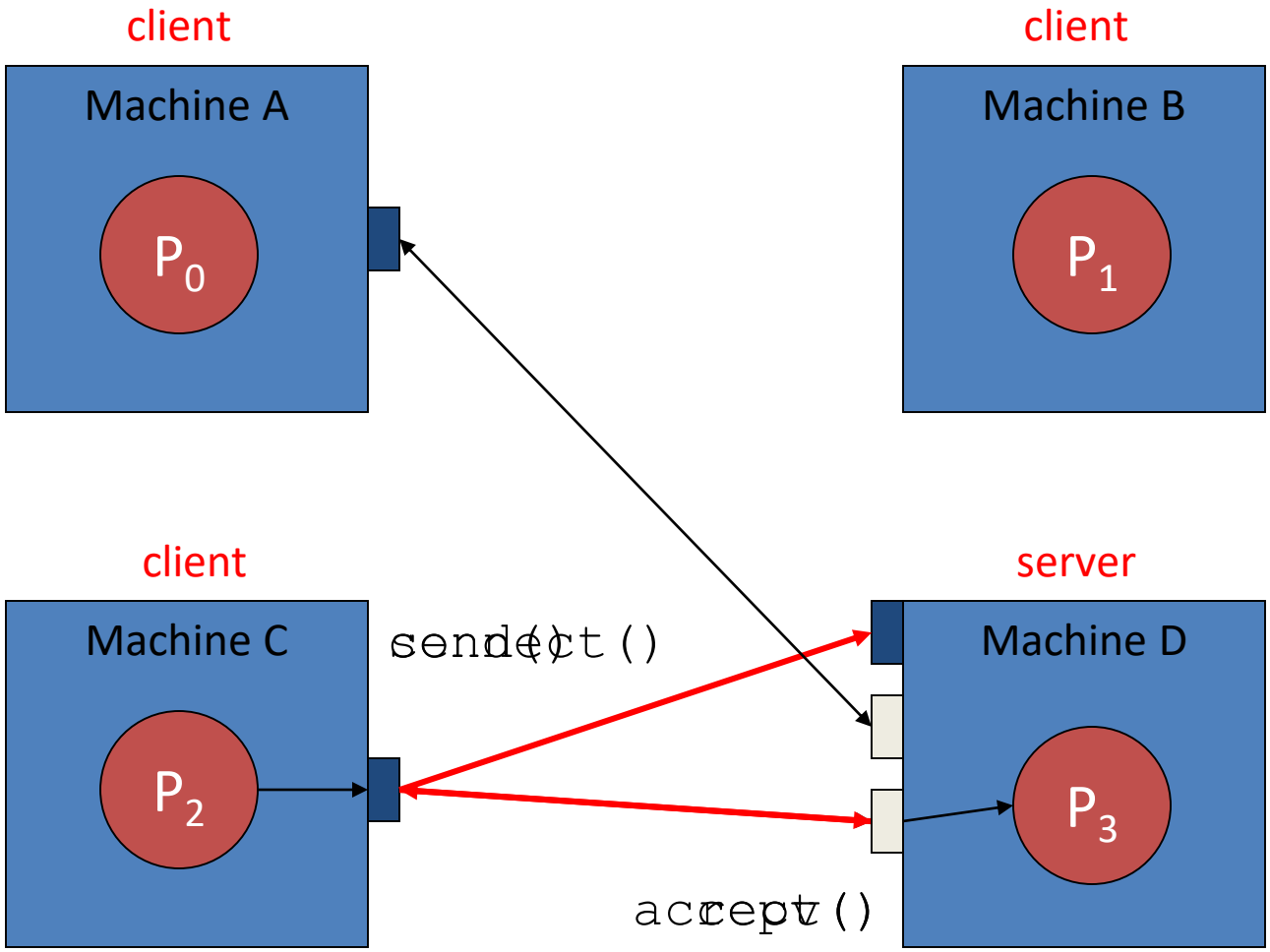
Client to Server



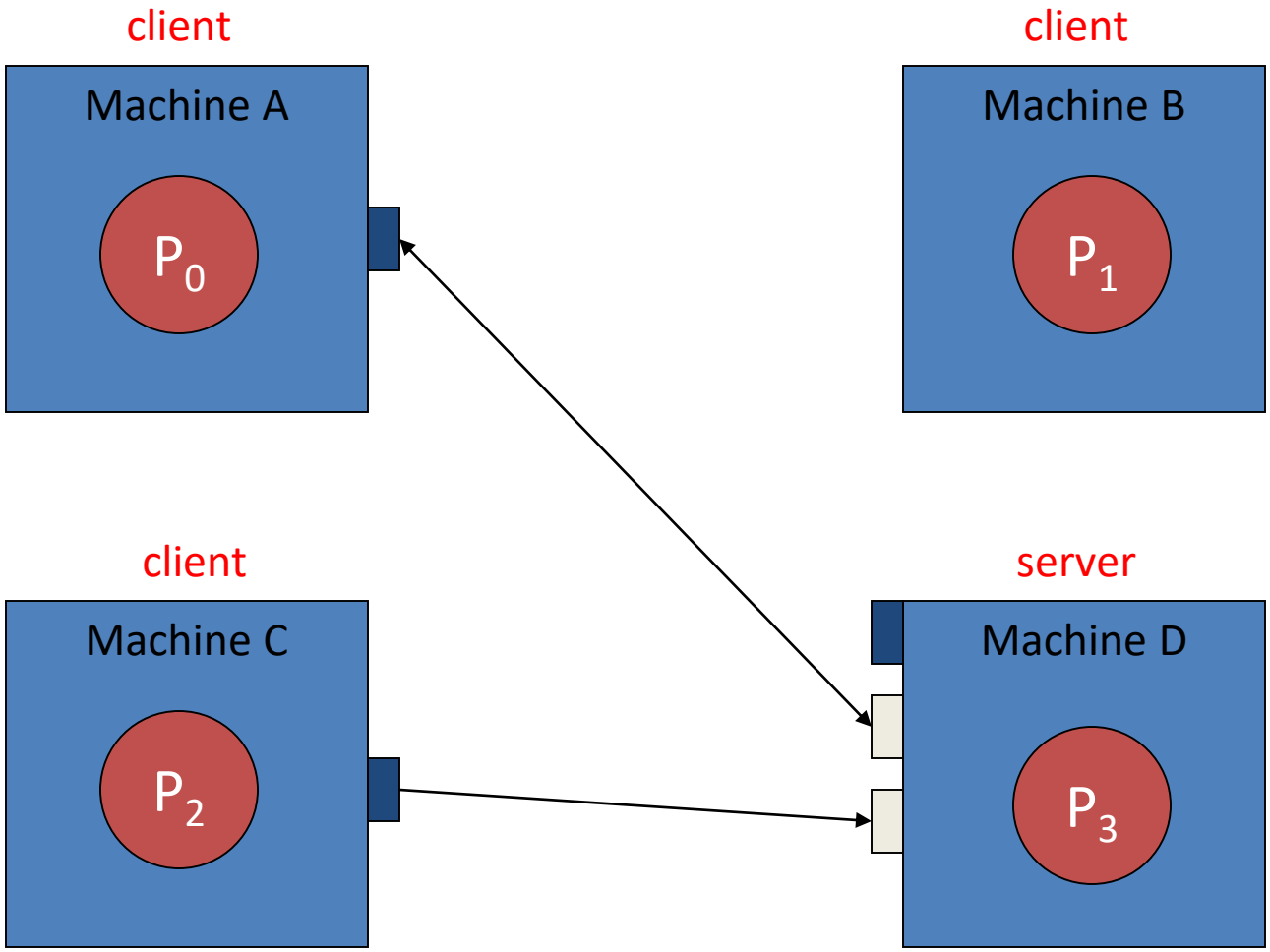
One-to-One Communication



Next Client


client
Machine A
P0
client
Machine B
P1
client
Machine C
P2
connect()
server
Machine D
P3
accept()

Still One-to-One



TCP Client Calls

```
socket()
gethostbyname( server )
connect()
send() / recv()
close()
```

TCP Server Calls

```
socket()
gethostbyname( self )
listen()
accept()
send() / recv()
close()
```

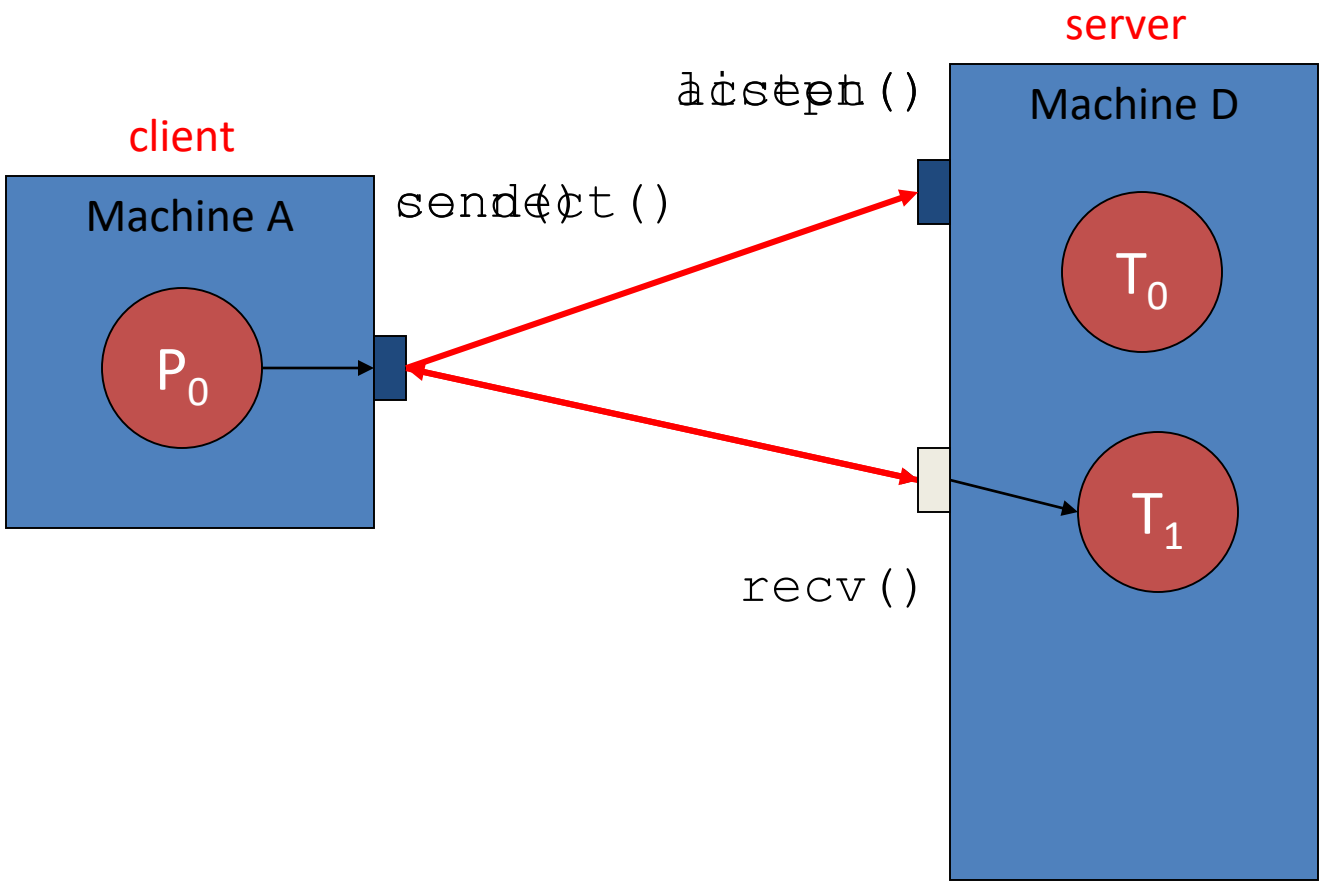
Normal Server Operation

```
listen()
while ( 1 ) {
   newFD = accept()
   pthread_create()
}
```

Child thread

- Handles request
- Communicates with client
- Terminates when service complete
- May block for I/O

Threaded Server



UDP

Sender

```
socket()
gethostbyname( dest )
sendto()
close()
```

Receiver

```
socket()
gethostbyname( dest )
bind ()
recvfrom()
close()
```

FTP via UDP

Datagram is limited in size (i.e., 10K)

How can we transfer large file (i.e., 100K)?

UDP Sender

- Partition file into 10 datagrams
- Send datagram 1, then datagram 2, …

UDP Receiver

- Receive all datagrams
- Reconstruct file from datagrams

FTP/UDP Problem

Some datagrams could be lost

1 2 5 6 8 9 10

Datagrams could arrive out of order

1 2 5 8 10 3 9 7 4 6

Or Both

1 5 8 10 7 4 6

What now?

- We didn't receive file as it was sent

UDP Solution?

Add a number to each datagram

- Sequence number (start at 0)
- Receiver can reorder datagrams
- Receiver can ask for lost datagram to be resent
- Receiver can send acknowledgement (ACK) of receipt for datagrams

Creating Streaming Sockets in Java

- A socket is created using the `java.net.Socket`:

  ```
  Socket socket = new Socket(host, port);
  ```

  `host` is a string specifying a host; can be localhost

  `port` is the port number on which the host is contacted

- Example:

  ```
  Socket socket = new
    java.net.Socket(localhost, 8080);
  ```

Creating Datagram Sockets in Java

- A datagram socket is created using `java.net.DatagramSocket`:

```
DatagramSocket dsocket =
        new DatagramSocket(port);
```

`port` is the UDP port on which packets will be received;

the datagram socket is bound to host's Internet address

- Example:

```
DatagramSocket dsocket =
  java.net.DatagramSocket(8080);
```

Java Streaming Sockets: Reading/Writing

- IO from a socket stream is like file IO:

```
// the buffer that stores read data
byte[] buffer = new byte[1024];

// creating an input stream; an error occurs if the
// socket is not open or not connected
InputStream iStream = socket.getInputStream();
OutputStream oStream = socket.getOutputStream();

// sending data stored in the buffer
oStream.write(buffer)

// reading data stored in the buffer
int numBytesRead = iStream.read(buffer)
```

Java Streaming Sockets: Terminating a Connection

- To end a connection, either side of the end can call `close()`.

  For example:

```
socket.close();
```

- If one end closes the socket and the other end does a
  - `read()`, the function will throw an IO exception,
  - `write()`, the function will throw an IO exception.

Server Socket in Java

- Server sockets in Java (`java.net.ServerSocket`):
  - A server socket is created as follows:

```
ServerSocket ssocket = new
  ServerSocket(port);
```

`port` is the port on which the server listens for connections

  - Example:

```
ServerSocket ssocket = ServerSocket(8080);
```

Designing a Network Protocol

- Server and client respond to messages based on a well-defined protocol.
- Protocol defines actions and data that a client requests from a server.
- Client and server compile and parse message based on protocol.
- Message can be in different formats including text, binary, or mixed.

Network Protocol Examples

- Online banking that supports users perform various banking transactions:

  <action>operation</action><value>aValue</value>

- Browser sends message to a server to retrieve a Web page.

Server-Client: Online Banking Example

Client sending a message to the server:

```
private String sendMessage(String message) throws Exception {
      Socket socket;
      InputStream iStream;
      OutputStream oStream;
      try {
        // creates socket to communicate with server
        socket = new Socket(_host, _port);

        // access data streams from socket
        iStream = socket.getInputStream();
        oStream = socket.getOutputStream();
      }
      catch (Exception xcp) {
         throw new Exception ("connection error: unable to connect to the server");
      }
……
```

Server-Client: Online Banking Example

Client sending a message to the server:

```
    String response;
    try {
       PrintWriter printWriter = new PrintWriter(oStream);
       Scanner scanner = new Scanner(iStream);

       // sending message
       printWriter.println(message);
       printWriter.flush();

       // receiving response
       response = scanner.nextLine();
    }
    catch (Exception xcp) {
       socket.close();
       throw new Exception ("unable to communicate with server");
    }
    // close connection to server
    socket.close();

    return response;
}
```

PrintWriter and Scanner aid in writing and reading text from a socket.

Server-Client: Online Banking Example

Sample protocol for banking transaction:

Client sends a message to the server to create an account:
create|*name*|*deposit*
name: name of the account owner
deposit: initial deposit of the account

Example:
create|joe smith|300

Server sends message confirming account creation:
account_number
account_number: the number of the account

Example:
5

Server-Client: Online Banking Example

Server socket waiting for incoming connection:

```
ServerSocket serverSocket;
try {
   serverSocket = new ServerSocket(_port);
}
catch (Exception xcp) {
   throw new Exception ("unabel to start server on port " + _port);
}
while (true) {
   Socket socket = serverSocket.accept();
   try {
      // get communication channels of socket
      InputStream iStream = socket.getInputStream();
      OutputStream oStream = socket.getOutputStream();

      // start a request handler in a separate thread
      RequestHandler requestHandler = new RequestHandler(iStream, oStream);
      new Thread(requestHandler).start();
   }
   catch (Exception xcp) {
      // do nothing; client has dropped connection
   }
}
```

Socket is accepting a connection to a host.

Handling request from client in a separate thread.

Server-Client: Online Banking Example

Handling a client's request:

```
public void run() {
        // read data from
        String requestMessage = _scanner.nextLine();
        String response = handleRequest(requestMessage);

        // send response back to host
        sendMessage(response);

        // close communication channels
        _printWriter.close();
        _scanner.close();
    }
```

Read request from client and handle it.

Sends computed response.

Closes communication.

Server-Client: Online Banking Example

Creating a bank account using information submitted by client:

```
public int createAccount(String name, float initialBalance) throws Exception {
      int accountNumber = _bankData.getNextAccountNumber();
      BankAccount bankAccount = new BankAccount(accountNumber, name, initialBalance);
      _bankData.write(bankAccount);

      return accountNumber;
}
```

Sends message back to client:

```
public void sendMessage(String message) {
        try {
           _printWriter.println(message);
           _printWriter.flush();
        }
        catch (Exception xcp) {
           System.err.println("error occurred sending message to host: " + message);
        }
     }
```

Server-Client: Online Banking Example

- Server accepts incoming connections continuously.
  - `accept()` blocks server until a new connection is requested
  - requests from clients are handled in separate threads

- Server terminates when program is shutdown.
  - more graceful shutdown could be done through a special server administration console

Summary Questions

- What is different between a UDP and a TCP socket?
- What values have to be mapped between host and network to maintain a correct byte order?
- When is *bind()* needed, when is it not needed to communicate with a host?
- What is the purpose of calling *listen()* on a TCP server socket?
- If a TCP server program communicates with 3 TCP clients how many sockets does the server program have open?