Internet Technology

02. Network Protocol Layers & Sockets

Paul Krzyzanowski
Rutgers University
Spring 2016
Protocols
What’s in the data?

• For effective communication
  – same language, same conventions

• For computers:
  – electrical encoding of data
  – where is the start of the packet?
  – which bits contain the length?
  – is there a checksum? where is it?
    how is it computed?
  – what is the format of an address?
  – byte ordering
These instructions & conventions are known as protocols

Protocols encompass data formats, order of messages, responses
Layering

To ease software development and maximize flexibility:

– Network protocols are generally organized in layers

– Replace one layer without replacing surrounding layers

– Higher-level software does not have to know how to format an Ethernet packet

… or even know that Ethernet is being used
Exist at different levels

understand format of address and how to compute a checksum  
humans vs. whales different wavelengths

versus

request web page  
French vs. Hungarian
Layering

Most popular model of guiding (not specifying) protocol layers is

**OSI reference model**

Adopted and created by ISO

7 layers of protocols

OSI = Open Systems Interconnection
From the ISO = International Organization for Standardization
OSI Reference Model: Layer 1

Transmits and receives raw data to communication medium

Does not care about contents

Media, voltage levels, speed, connectors

1 Physical

Deals with representing bits

Examples: USB, Bluetooth, 802.11
OSI Reference Model: Layer 2

- Detects and corrects errors
- Organizes data into frames before passing it down. Sequences packets (if necessary)
- Accepts acknowledgements from immediate receiver

Examples: Ethernet MAC, PPP
OSI Reference Model: Layer 3

Relay and route information to destination

Manage journey of datagrams and figure out intermediate hops (if needed)

Examples: IP, X.25
OSI Reference Model: Layer 4

Provides an interface for end-to-end (application-to-application) communication: sends & receives segments of data. Manages flow control. May include end-to-end reliability

Network interface is similar to a mailbox

Examples: TCP, UDP
OSI Reference Model: Layer 5

Session

Services to coordinate dialogue and manage data exchange

Transport

Software implemented switch

Network

Manage multiple logical connections

Data Link

Keep track of who is talking: establish & end communications

Physical

Deals with data streams

Examples: HTTP 1.1, SSL
OSI Reference Model: Layer 6

Data representation
Concerned with the meaning of data bits
Convert between machine representations

Examples:
XDR, ASN.1, MIME, XML

Deals with objects
OSI Reference Model: Layer 7

Collection of application-specific protocols

Examples:
- web (HTTP)
- email (SMTP, POP, IMAP)
- file transfer (FTP)
- directory services (LDAP)

Deals with app-specific protocols
### IP vs. OSI stack

<table>
<thead>
<tr>
<th>Layer</th>
<th>Internet protocol stack</th>
<th>OSI protocol stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Application</td>
<td>Application</td>
</tr>
<tr>
<td>6</td>
<td>Transport</td>
<td>Presentation</td>
</tr>
<tr>
<td>5</td>
<td>Network</td>
<td>Session</td>
</tr>
<tr>
<td>4</td>
<td>Data Link</td>
<td>Transport</td>
</tr>
<tr>
<td>3</td>
<td>Physical</td>
<td>Network</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Data Link</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Physical</td>
</tr>
</tbody>
</table>

CS 352 © 2013-2016 Paul Krzyzanowski
A layer communicates with its counterpart
A layer communicates with its counterpart

Logical View

1. Physical
2. Data Link
3. Network
4. Transport
5. Session
6. Presentation
7. Application
A layer communicates with its counterpart
A layer communicates with its counterpart

Logical View

1. Physical
2. Data Link
3. Network
4. Transport
5. Session
6. Presentation
7. Application

CS 352 © 2013-2016 Paul Krzyzanowski
But really traverses the stack

What’s really happening

1. Physical
2. Data Link
3. Network
4. Transport
5. Session
6. Presentation
7. Application
Encapsulation

At any layer
- The higher level protocol headers are just treated like data
- Lower level protocol headers can be ignored
Writing network applications

Network applications communicate with each other over a network

• Regular processes running on computers
  – Any process can access the network

• Use a network API to communicate
  – The app developer does not have to program the lower layers

• Speak a well-defined application-layer protocol
  – If the protocol is well-defined, the implementation language does not matter
    E.g., Java on one side, C on the other
Application Architectures

- Client-server
- Peer-to-peer (P2P)
- Hybrid
Client-Server architecture

- Clients send requests to a server
- The server is always on and processes requests from clients
- Clients do not communicate with other clients
- Examples:
  - FTP, web, email
Peer-to-Peer (P2P) architecture

- Little or no reliance on servers
- One machine talks to another (peers)
- Peers are not owned by the service provider but by end users
- **Self-scalability**
  - System can process more workload as more machines join
- **Examples**
  - BitTorrent, Skype
Hybrid architecture

• Many peer-to-peer architectures still rely on a server
  – Look up, track users
  – Track content
  – Coordinate access

• But traffic-intensive workloads are delegated to peers
It’s always *mostly* client-server!

Even for P2P architectures, we may use client-server terminology

- **Client**: process making a request
- **Server**: process fulfilling the request
Network API

• App developers need access to the network

• A *Network Application Programming Interface (API)* provides this
  
  – Core services provided by the operating system
    • Operating System controls access to resources (the network)

  – Libraries handle the rest
What do we need as programmers?

• Reliable data transfer
  – Reliable delivery of a stream of bytes from one machine to another
  – In-order message delivery
  – Loss-tolerant applications
    • Can handle unreliable data streams

• Throughput
  – Bandwidth sensitive applications: require a particular bitrate
  – Elastic applications: can adapt to available bitrate

• Delay & Jitter Control
  – Jitter = variation in delay

• Security
  – Authentication of endpoints, encryption of content, assured data integrity
What IP gives us

IP give us two transport protocols

- **TCP: Transmission Control Protocol**
  - Connection-oriented service
    - Operating system keeps state
  - Full-duplex connection: both sides can send messages over the same link
  - Reliable data transfer: the protocol handles retransmission
  - In-order data transfer: the protocol keeps track of sequence numbers

- **UDP: User Datagram Protocol**
  - Connectionless service: lightweight transport layer over IP
  - Data may be lost
  - Data may arrive out of sequence
  - Checksum for corrupt data: operating system drops bad packets
What IP does *not* give us

- Throughput (bandwidth) control
- Delay and jitter control
- Security

We’ll see how these were addressed later in the course

Usually addressed at the application with protocols such as SSL. Stay tuned for VPNs…
Addressing machines

(We’ll examine IP addresses in depth later)

Machine addresses

– We identify machines with IP addresses: 32-bit numbers
– Example
  
  cs.rutgers.edu = 128.6.4.2 = 0x8060402

  0x80  0x06  0x04  0x02
Addressing applications

Communication endpoint at the machine

- **Port number**: 16-bit value
  - Port number = transport endpoint
    - Allows application-application communication
    - Identifies a specific data stream

- Some services use well-known port numbers (0 – 1023)
  - IANA: Internet Assigned Numbers Authority (www.iana.org)
  - Also see the file /etc/services

- Ports for proprietary apps: 1024 – 49151
- Dynamic/private ports: 49152 – 65535
The Application Layer: Sockets
Sockets

• Dominant API for transport layer connectivity

• Created at UC Berkeley for 4.2BSD Unix (1983)

• Design goals
  – Communication between processes should not depend on whether they are on the same machine
  – Communication should be efficient
  – Interface should be compatible with files
  – Support different protocols and naming conventions
    • Sockets is not just for the Internet Protocol family
What is a socket?

Abstract object from which messages are sent and received

– Looks like a file descriptor

– Application can select particular style of communication
  • Stream (connection-oriented), datagram (connectionless), message-based, in-order delivery

– Unrelated processes should be able to locate communication endpoints
  • Sockets can have a name
  • Name should be meaningful in the communications domain
    – E.g., Address & port for IP communications
How are sockets used?

Client: web browser

Send HTTP request message to get a page

Receive HTTP request message

Process HTTP request

Send HTTP response message

Receive HTTP response message

Display a page

Server: web server
Connection-Oriented (TCP) socket operations

**Client**
- Create a socket
- Name the socket (assign local address, port)
- Connect to the other side
- read / write byte streams
- close the socket

**Server**
- Create a socket
- Name the socket (assign local address, port)
- Set the socket for listening
- Wait for and accept a connection; get a socket for the connection
- read / write byte streams
- close the socket
- close the listening socket
Connectionless (UDP) socket operations

Client

Create a socket

Name the socket (assign local address, port)

Send a message

Receive a message

close the socket

Server

Create a socket

Name the socket (assign local address, port)

Receive a message

Send a message

close the socket
The sockets system call interface
# POSIX system call interface

<table>
<thead>
<tr>
<th>System call</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket</td>
<td>Create a socket</td>
</tr>
<tr>
<td>bind</td>
<td>Associate an address with a socket</td>
</tr>
<tr>
<td>listen</td>
<td>Set the socket to listen for connections</td>
</tr>
<tr>
<td>accept</td>
<td>Wait for incoming connections</td>
</tr>
<tr>
<td>connect</td>
<td>Connect to a socket on the server</td>
</tr>
<tr>
<td>read/write, sendto/recvfrom, sendmsg/recevmsg</td>
<td>Exchange data</td>
</tr>
<tr>
<td>close/shutdown</td>
<td>Close the connection</td>
</tr>
</tbody>
</table>
Step 1 (client & server)

Create a socket

```c
int s = socket(domain, type, protocol)
```

- **AF_INET**: Address Family: group of protocols for communication. 
  - AF_INET is for IPv4
  - AF_INET6 is IPv6
  - AF_BTH is Bluetooth

- **SOCK_STREAM**: Type of protocol within the family. 
  - SOCK_STREAM: reliable, in-order, 2-way. TCP/IP

- **SOCK_DGRAM**: datagrams (UDP/IP)

- **SOCK_RAW**: “raw” – allows app to modify the network layer header

Conceptually similar to `open` BUT
- `open` creates a new reference to a possibly existing object
- `socket` creates a new instance of an object

useful if some families have more than one protocol to support a given service.
0: unspecified
Step 2 (client & server)

Name the socket (assign *address*, *port*)

\[
\text{int error} = \text{bind}(s, \text{addr}, \text{addrlen})
\]

- **socket**
  - The socket from the *socket* system call.

- **Address structure**
  - `struct sockaddr*`
  - This is a data structure that makes sense for whatever address family you selected.

- **length of address structure**

Naming for an IP socket is the process of assigning our address to the socket. The address is the full transport address: the IP address of the network interface as well as the UDP or TCP port number.
Step 3a (server)

Set socket to be able to accept connections

\[
\text{int error = listen(s, backlog)}
\]

The socket from the `socket` system call.

Number of connections you’ll allow between `accept` system calls

The socket that the server created with `socket` is now configured to accept new connections. This socket will only be used for accepting connections. Data will flow onto another socket.
Step 3b (server)

Wait for a connection from client

```
int snew = accept(s, clntaddr, &clntalen)
```

- `socket` - This is the listening socket
- `pointer to address structure` - This tells you where the socket came from: full transport address.
- `length of address structure` - New socket for this communication session.

Block the process until an incoming connection comes in.
Step 3 (client)

Connect to server

```c
int error = connect(s, svraddr, svraddrlen)
```

- `socket`: The socket from which we’re connecting.
- `address structure`: Full transport address of the destination: address and port number of the service.
- `length of address structure`:

The client can send a connection request to the server once the server did a `listen` and is waiting for `accept`. 
Step 4. Exchange data

- **read/write** system calls (same as for file systems)
  
```
int send(int s, void *msg, int len, uint flags);
int recv(int s, void *buf, int len, uint flags);
```

- **send/recv** system calls
  
```
int send(int s, void *msg, int len, uint flags);
int sendto(int s, void *msg, int len, uint flags, struct sockaddr *to, int tolen);
int recv(int s, void *buf, int len, uint flags);
int recvfrom(int s, void *buf, int len, uint flags, struct sockaddr *from, int *fromlen);
```

- **sendto/recvfrom** system calls
  
```
int sendto(int s, void *msg, int len, uint flags, struct sockaddr *to, int tolen);
int recvfrom(int s, void *buf, int len, uint flags, struct sockaddr *from, int *fromlen);
```

- **sendmsg/recvmsg** system calls
  
```
int sendmsg(int s, struct msghdr *msg, uint flags);
int recvmsg(int s, struct msghdr *msg, uint flags);
```

For connection-oriented service

- **sendto/recvfrom** system calls

  Like **read** and **write** but these support extra flags, such as bypassing routing or processing out of band data. Not all sockets support these.

For connectionless service

- **send/recv** system calls

  If we’re using UDP (connectionless), we don’t need to do **connect**, **listen**, **accept**. These calls allows you to specify the destination address (sendto, sendmsg) to send a message and get the source address (recvfrom, recvmsg) when receiving a message.
Step 5

Close connection

```c
shutdown(s, how)
```

**how:**
- SHUT_RD (0): can send but not receive
- SHUT_WR (1): cannot send more data
- SHUT_RDWR (2): cannot send or receive (=0+1)

You can use the regular `close` system call too, which does a complete shutdown, the same as `shutdown(s, SHUT_RDWR)`. 
Java provides shortcuts that combine calls

Example

Java

Socket s = new Socket(“www.rutgers.edu”, 2211)

C

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

struct sockaddr_in myaddr; /* initialize address structure */
myaddr.sin_family = AF_INET;
myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
myaddr.sin_port = htons(0);
bind(s, (struct sockaddr *)&myaddr, sizeof(myaddr));

/* look up the server's address */
struct hostent *hp; /* host information */
struct sockaddr_in servaddr; /* server address */
memset((char*)&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(2211);
hp = gethostbyname(“www.rutgers.edu”);

if (connect(fd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
    /* connect failed */
}
Using sockets in Java

• **java.net** package
  – **Socket** class
    • Deals with sockets used for TCP/IP communication
  – **ServerSocket** class
    • Deals with sockets used for accepting connections
  – **DatagramSocket** class
    • Deals with datagram packets (UDP/IP)

• Both **Socket** and **ServerSocket** rely on the **SocketImpl** class to actually implement sockets
  – But you don’t have to think about that as a programmer
Create a socket for listening: server

Server:
- `create`, `name`, and `listen` are combined into one method
- `ServerSocket` constructor

```java
ServerSocket svc = new ServerSocket(80, 5);
```

Several other flavors (see API reference)
1. Server: create a socket for listening

Client: web browser

Server: web server

```
Server Socket svc = new ServerSocket(80, 5);
```

Send HTTP request message to get a page

Receive HTTP request message

Process HTTP request

Send HTTP response message

Receive HTTP response message

Display a page
Server: wait for (accept) a connection

- **accept** method of **ServerSocket**
  - block until connection arrives
  - return a **Socket**

```java
ServerSocket svc = new ServerSocket(80, 5);
Socket req = svc.accept();
```

This is a *new* socket for this “connection”
2. Server: wait for a connection (blocking)

Client: web browser

Server: web server

Send HTTP request message
to get a page

Receive HTTP request message
Process HTTP request
Send HTTP response message
Receive HTTP response message
Display a page

Server Socket svc = new ServerSocket(80);
Socket req = svc.accept();

Block until an incoming connection comes in
Create a socket: client

Client:

– *create*, *name*, and *connect* operations are combined into one method

– **Socket** constructor

```
Socket s = new Socket("www.rutgers.edu", 2211);
```

Several other flavors (see API reference)
3. Client: connect to server socket (blocking)

Client: web browser

Server: web server

Socket s = new Socket("pk.org", 80);

blocks until connection is set up

Send HTTP request message to get a page

Receive HTTP request message

Process HTTP request

Send HTTP response message

Receive HTTP response message

Display a page

Server Socket svc = new ServerSocket(80, 5);

Socket req = svc.accept();
3a. Connection accepted

Client: web browser

Server: web server

Socket $s = \text{new Socket} ("pk.org", 80);$  

$\text{Connection is established}$

Send HTTP request message to get a page

$\text{Receive HTTP request message}$

Process HTTP request

Receive HTTP response message

Send HTTP response message

Display a page

Server Socket $svc = \text{new ServerSocket}(80, 5);$  

Socket $\text{req} = svc.\text{accept}();$  

$\text{Connection is accepted}$
Exchange data

• Obtain InputStream and OutputStream from Socket
  – layer whatever you need on top of them
  • e.g. DataInputStream, PrintStream, BufferedReader, …

Example:

client
DataInputStream in = new DataInputStream(s.getInputStream());
PrintStream out = new PrintStream(s.getOutputStream());

server
DataInputStream in = new BufferedReader(
  new InputStreamReader(req.getInputStream()));
String line = in.readLine();
DataOutputStream out = new DataOutputStream(
  req.getOutputStream());
out.writeBytes(mystring + '\n')
4. Perform I/O (read, write)

Client: web browser

Server: web server

Server Socket svc = new ServerSocket(80, 5);

Socket s = new Socket("pk.org", 80);

InputStream s_in = s.getInputStream();
OutputStream s_out = s.getOutputStream();

Send HTTP request message to get a page

Receive HTTP request message

Process HTTP request

Receive HTTP response message

Send HTTP response message

Display a page
Close the sockets

Close input and output streams first, then the socket

**client:**

```java
try {
    out.close();
    in.close();
    s.close();
} catch (IOException e) {} 
```

**server:**

```java
try {
    out.close();
    in.close();
    req.close();    // close connection socket
    svc.close();    // close ServerSocket
} catch (IOException e) {} 
```
TCP vs. UDP sockets

• TCP (“stream sockets”)
  – Requires a connection (connection-oriented)
  – Dedicated socket for accepting connections
  – Communication socket provides a bi-directional link
  – Byte-stream: no message boundaries

• UDP (“datagram sockets”)
  – Connectionless: you can just send a message
  – Data send in discrete packets (messages)
UDP workflow

Client

Send request packet

Wait for request packet

Process request

Send response packet

Receive response packet

Server
/* read a line from the user */
BufferedReader user_input = new BufferedReader(new InputStreamReader(System.in));
String line = user_input.readLine;

/* convert it to an array of bytes */
byte[] out_data = line.getBytes();

/* create a datagram socket */
DatagramSocket s = new DatagramSocket();

InetAddress addr = InetAddress.getByName("test.pk.org"); /* look up IP address */
int port = 1234; /* port number */

/* construct the packet */
DatagramPacket out_packet = new DatagramPacket(data, data.length, addr, port);

/* send it out on the socket */
s.send(out_packet);
Receive a packet

byte in_buf[] new byte[1500];
int port = 4321;    /* port number on which we want to receive data */

/* create a datagram socket */
DatagramSocket s = new DatagramSocket(port);

/* create the packet for receiving the data*/
DatagramPacket in_packet = new DatagramPacket(in_buf, in_buf.length);

/* get the packet from the socket*/
s.receive(in_packet);

System.out.println(
    "received data [" + new String(in_packet.getData(), 0, in_packet.getLength()) + "]" + 
    " from address: " + in_packet.getAddress() + 
    " port: " + in_packet.getPort();
Concurrency & Threads
Threads

• Designed to support multiple flows of execution in one process

• Each thread is scheduled by the operating system’s scheduler

• Each thread has its own stack
  – Local variables are local to each thread

• Shared heap
  – Global and static variables and allocated memory are shared

• Multi-core processors make threading attractive
  – Two or more threads can run at the same time
Appeal of threads

• One process can handle multiple requests at the same time
  – Some threads may be blocked
  – Does not affect the threads that have work to do

• User interactivity possible even if certain events block
  – Examples:
    • disk reads
    • wait for network messages
    • count words
    • justify text
    • check spelling
Java Threads

• Create a class that extends Thread or implements Runnable

• Instantiate this class or a Thread to run this Runnable

• When the run method is invoked, it starts a new thread of execution
  – After the caller returns, the run method is still running … as a separate thread
  – Call join to wait for the run method to terminate (return)
Java Threads example

/* Worker defines the threads that we’ll create */
Class Worker extends Thread {
    Worker(...) {  // constructor
    }
    public void run() {
        /* thread’s work goes here */
        /* thread exits when run() is done */
    }
}

/* other code to start thread */
Worker T = new Worker();  // constructor

T.start();  // start new thread in run method
            // original thread keeps running …

T.join();  // wait for T’s thread to finish.
Java Threads

Main thread

\[ T = \text{new Worker(…)} \]  
\[ T.\text{start()} \]

Work…

\[ T.\text{join()} \]

Wait for the thread to exit

Continue with code after the T.join()

New thread

\[ \text{new object created} \]

run()

Thread work…

return;

Thread terminates
Example of threads in a server

• Main thread
  – Waits for new requests from clients
  – After an *accept*, create a worker thread to handle the socket connection for that client

• Worker thread handles the request for the client
  – Returns when done – thread disappears
Example of threads in a server

for (;;) {
    Socket r = ss.accept(...)
    /* wait for a new connection */
    doWork worker = new doWork(r);
    /* create the object */
    Thread t = new Thread(worker);
    /* create the thread */
    t.start();
    /* start running it */
    /* … and loop back to wait for the next connection */
}

public class doWork implements Runnable {
    private Socket sock;

    doWork(Socket sock) {
        this.sock = sock;
    }

    public void run() {
        /* here’s where the work is done */
        DataInputStream in = new DataInputStream(sock.getInputStream());
        PrintStream out = new PrintStream(server.getOutputStream());
        /* do the work */
        sock.close();
    }
}

This example shows threads with “implements Runnable”
Threads allow concurrent access

- Threads allow shared access to shared data
- If two threads access the same data at the same time, results can be undefined
A race condition is a bug:

– The outcome of concurrent threads is unexpectedly dependent on a specific sequence of events

Example

– Your current bank balance is $1,000
– Withdraw $500 from an ATM machine while a $5,000 direct deposit is coming in

Withdrawal

- Read account balance
- Subtract $500
- Write account balance

Deposit

- Read account balance
- Add $5,000
- Write account balance

Possible outcomes:

Total balance = $5500 (✓), $500 (X), $6000 (X)
Synchronization

• Synchronization: techniques to avoid race conditions
  – Prevent concurrent access

• Operating systems may give us:
  – Semaphores, messages, condition variables, event counters

• Synchronization in Java
  – Add the keyword synchronized to a method
    • JVM ensures that at most one thread can execute that method at a time

```java
Account {
    double total;
    public synchronized void withdraw(double amount) {
        this.total -= amount;
    }
    public synchronized void deposit(double amount) {
        this.total += amount;
    }
}
```

These two methods will never execute concurrently if they’re in the same object
Finer-grain synchronization: blocks

• The `synchronized` keyword provides method-level mutual exclusion
  – Among all methods that are synchronized, only 1 can execute at a time
• Synchronized block: create a mutex for a region

```java
Account {
    double total;
    public void withdraw(double amount) {
        synchronized(this.total) {
            this.total -= amount;
        }
    }
    public void deposit(double amount) {
        synchronized(this.total) {
            this.total += amount;
        }
    }
}
```

These two blocks will never execute concurrently

`this.total` becomes a `monitor object`.

Only one thread can execute in a block synchronized on the same monitor object.
The end