TCP/IP

Transmission Control Protocol / Internet Protocol

Based on Notes by D. Hollinger

Topics

- IPv6
- TCP
- Java TCP Programming

IPv6 availability

- Generally available with (new) versions of most operating systems.
  - BSD, Linux 2.2 Solaris 8
- An option with Windows 2000/NT
- Most routers can support IPV6
- Supported in J2SDK/JRE 1.4

IPv6 Design Issues

- Overcome IPv4 scaling problem
  - lack of address space.
- Flexible transition mechanism.
- New routing capabilities.
- Quality of service.
- Security.
- Ability to add features in the future.

IPv6 Headers

- Simpler header - faster processing by routers.
  - No optional fields - fixed size (40 bytes)
  - No fragmentation fields.
  - No checksum
- Support for multiple headers
  - more flexible than simple “protocol” field.

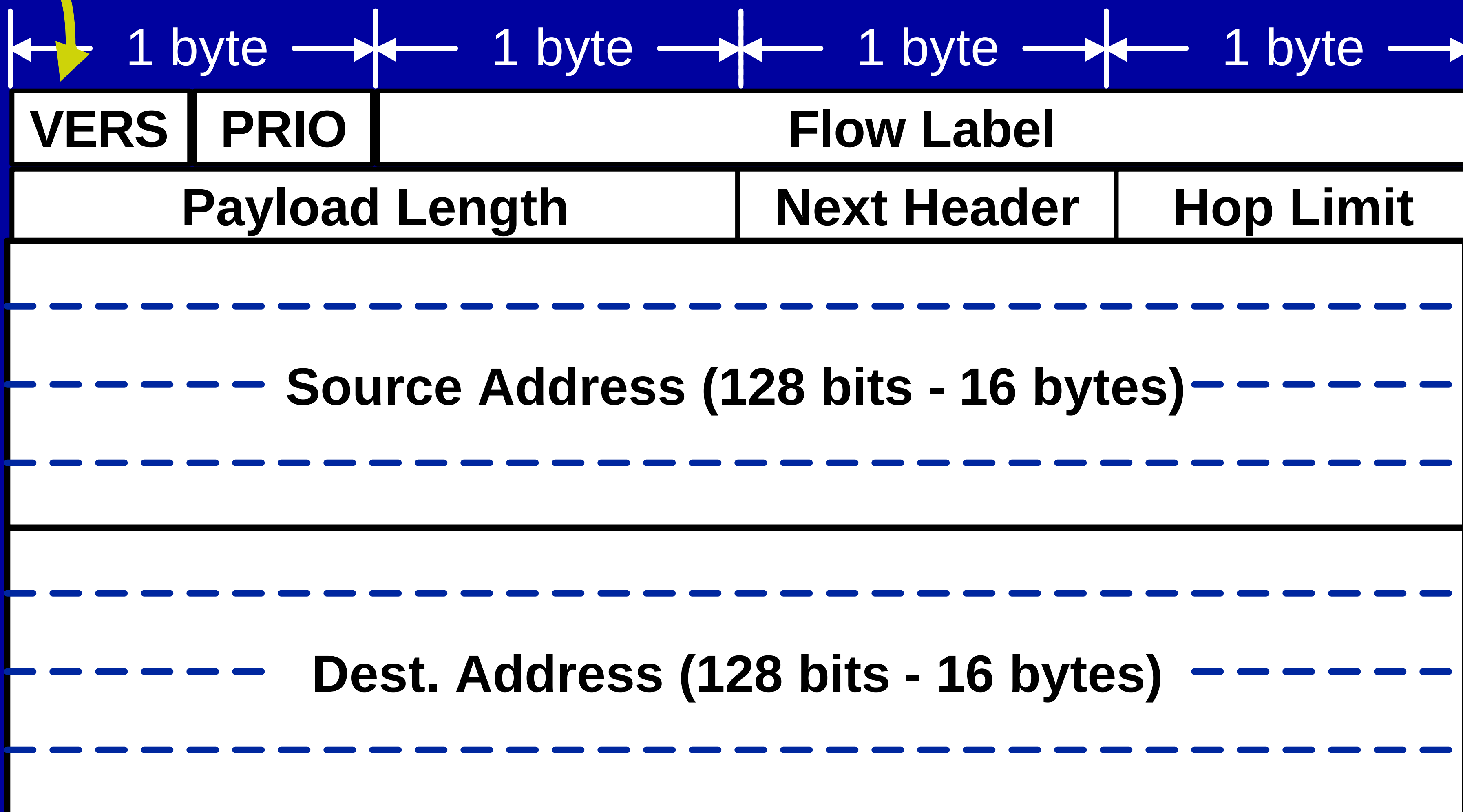
4 for IPv4

IPv4 Header

<table>
<tr><td colspan="2">1 byte</td><td colspan="2">1 byte</td><td colspan="2">1 byte</td><td colspan="2">1 byte</td></tr>
<tr><td>VERS</td><td>HL</td><td colspan="2">Service</td><td colspan="4">Fragment Length</td></tr>
<tr><td colspan="4">Datagram ID</td><td>FLAG</td><td colspan="3">Fragment Offset</td></tr>
<tr><td colspan="2">TTL</td><td colspan="2">Protocol</td><td colspan="4">Header Checksum</td></tr>
<tr><td colspan="8">Source Address</td></tr>
<tr><td colspan="8">Destination Address</td></tr>
<tr><td colspan="8">Options (if any)</td></tr>
<tr><td colspan="8">Data</td></tr>
</table>

6 for IPv6

IPv6 Header



IPv6 Header Fields

- VERS: 6 (IP version number)
- Priority: will be used in congestion control
- Flow Label: experimental - sender can label a sequence of packets as being in the same flow.
- Payload Length: number of bytes in everything following the 40 byte header (up to 64Kb), or 0 for a *Jumbogram (up to 4Gb)*.

IPv6 Header Fields

- Next Header is similar to the IPv4 “protocol” field - indicates what type of header follows the IPv6 header.
- Hop Limit is similar to the IPv4 TTL field (but now it really means hops, not time).

Extension Headers

- Routing Header - source routing
- Fragmentation Header - supports fragmentation of IPv6 datagrams.
- Authentication Header
- Encapsulating Security Payload Header

IPv6 Addresses

- 128 bits - written as eight 16-bit hex numbers.

  `5f1b:df00:ce3e:e200:0020:0800:2078:e3e3`

- High order bits determine the *type* of address.

IPv6
Aggregate Global Unicast Address

3	13	32	16	64
010	**TLA ID**	**NLA ID**	**SLA ID**	**Interface ID**

TLA: top-level aggregation (provider)
NLA: next-level (subscriber)
SLA: site-level (subnet)

Interface ID is (typically) based on hardware MAC address

IPv4-Mapped IPv6 Address

- IPv4-Mapped addresses allow a host that support both IPv4 and IPv6 to communicate with a host that supports only IPv4.

- The IPv6 address is based completely on the IPv4 address.

IPv4-Mapped IPv6 Address

- 80 bits of 0s followed by 16 bits of ones, followed by a 32 bit IPv4 Address:

0000 . . . 0000	FFFF	IPv4 Address
80 bits	16 bits	32 bits

Works with DNS

- An IPv6 application asks DNS for the address of a host, but the host only has an IPv4 address.
- DNS creates the IPv4-Mapped IPv6 address automatically.
- Kernel understands this is a special address and really uses IPv4 communication.

IPv4-Compatible IPv6 Address

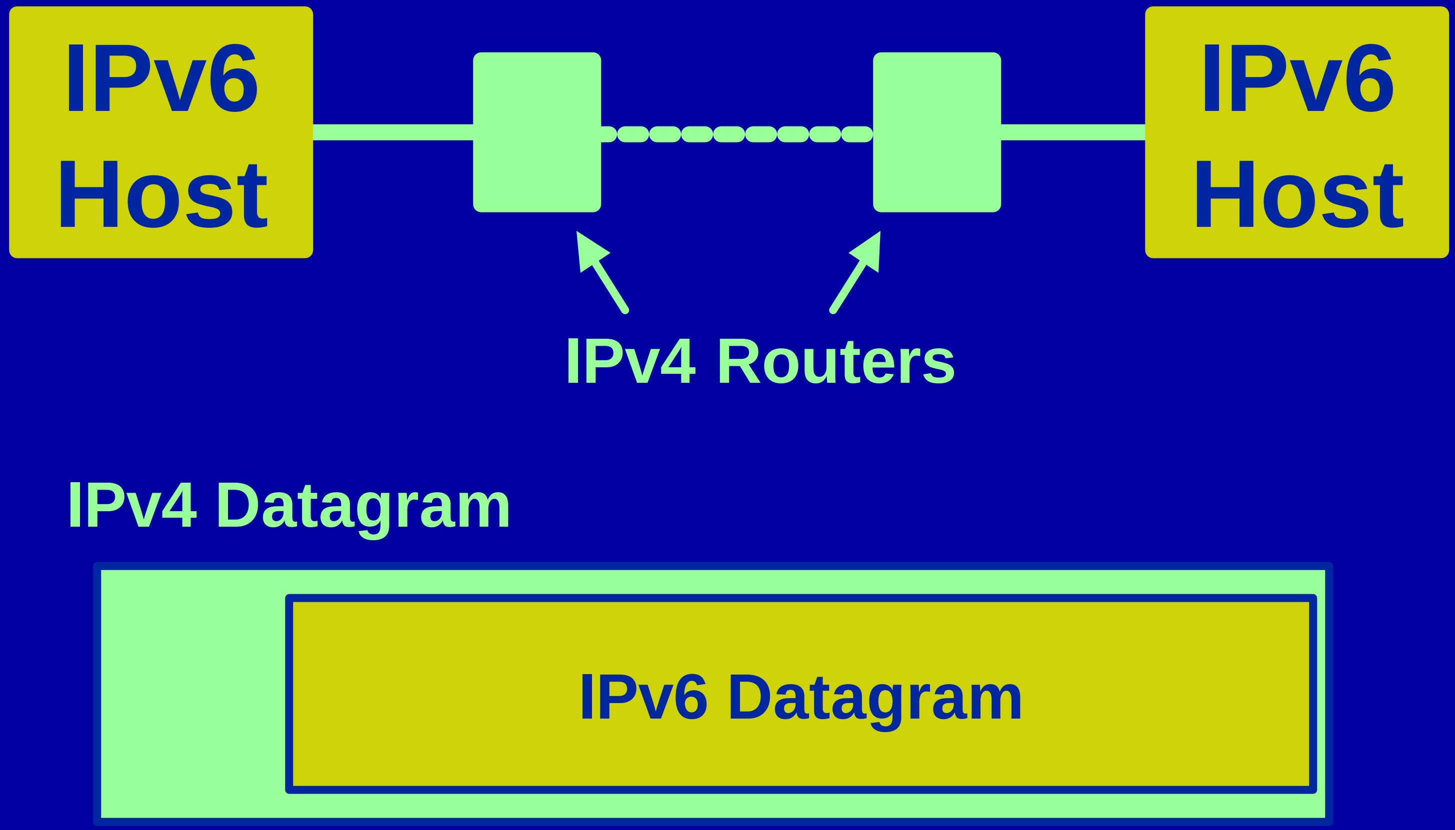
- An IPv4 compatible address allows a host supporting IPv6 to talk IPv6 even if the local router(s) don't talk IPv6.
- IPv4 compatible addresses tell endpoint software to create a tunnel by encapsulating the IPv6 packet in an IPv4 packet.

IPv4-Compatible IPv6 Address

- 80 bits of 0s followed by 16 bits of 0s, followed by a 32 bit IPv4 Address:

0000 . . . 0000	0000	IPv4 Address
80 bits	16 bits	32 bits

Tunneling
(done automatically by kernel when IPv4-Compatible IPv6 addresses used)



IPv6 in Java 1.4

- Inet6Address class
  - :: corresponds to 0.0.0.0 (unspecified) in IPv4
  - ::1 corresponds to 127.0.0.1 (loopback) in IPv4
  - ::ffff:w.x.y.z IPv4-mapped address
  - ::w.x.y.z to tunnel IPv6 packets over IPv4 routing

- For details, see:

http://java.sun.com/j2se/1.4/docs/guide/net/ipv6_guide/

TCP
Transmission Control Protocol

- TCP is an alternative transport layer protocol over IP.
- TCP provides:
  - Connection-oriented
  - Reliable
  - Full-duplex
  - Byte-Stream

Connection-Oriented

- *Connection oriented* means that a virtual connection is established before any user data is transferred.
- If the connection cannot be established - the user program is notified.
- If the connection is ever interrupted - the user program(s) is notified.

Reliable

- *Reliable* means that every transmission of data is acknowledged by the receiver.
- If the sender does not receive acknowledgement within a specified amount of time, the sender retransmits the data.

Byte Stream

- *Stream* means that the connection is treated as a stream of bytes.

- The user application does not need to package data in individual datagrams (as with UDP).

Buffering

- TCP is responsible for buffering data and determining when it is time to send a datagram.

- It is possible for an application to tell TCP to send the data it has buffered without waiting for a buffer to fill up.

Full Duplex

- TCP provides transfer in both directions.
- To the application program these appear as 2 unrelated data streams, although TCP can piggyback control and data communication by providing control information (such as an ACK) along with user data.

TCP Ports

- Interprocess communication via TCP is achieved with the use of ports (just like UDP).

- UDP ports have no relation to TCP ports (different name spaces).

TCP Segments

- The chunk of data that TCP asks IP to deliver is called a *TCP segment*.

- Each segment contains:
  - data bytes from the byte stream
  - control information that identifies the data bytes

TCP Segment Format

<table>
<tr><th colspan="3">1 byte</th><th>1 byte</th><th>1 byte</th><th>1 byte</th></tr>
<tr><td colspan="4">Source Port</td><td colspan="2">Destination Port</td></tr>
<tr><td colspan="6">Sequence Number</td></tr>
<tr><td colspan="6">Acknowledgment Number</td></tr>
<tr><td>offset</td><td>Reser.</td><td colspan="2">Control</td><td colspan="2">Window</td></tr>
<tr><td colspan="4">Checksum</td><td colspan="2">Urgent Pointer</td></tr>
<tr><td colspan="6">Options (if any)</td></tr>
<tr><td colspan="6">Data</td></tr>
</table>

TCP Lingo

- When a client requests a connection it sends a “SYN” segment (a special TCP segment) to the server port.
- SYN stands for synchronize. The SYN message includes the client’s ISN.
- ISN is Initial Sequence Number.

More...

- Every TCP segment includes a *Sequence Number* that refers to the first byte of *data* included in the segment.
- Every TCP segment includes an *Acknowledgement Number* that indicates the byte number of the next data that is expected to be received.
  - All bytes up through this number have already been received.

And more...

- There are a bunch of control flags:
  - URG: urgent data included.
  - ACK: this segment is (among other things) an acknowledgement.
  - RST: error – connection must be reset.
  - SYN: synchronize Sequence Numbers (setup)
  - FIN: polite connection termination.

And more...

- MSS: Maximum segment size (A TCP option)
- Window: Every ACK includes a Window field that tells the sender how many bytes it can send before the receiver will have to toss it away (due to fixed buffer size).

TCP Connection Creation

- Programming details later - for now we are concerned with the actual communication.
- A *server* accepts a connection.
  - Must be looking for new connections!
- A *client* requests a connection.
  - Must *know* where the server is!

Client Starts

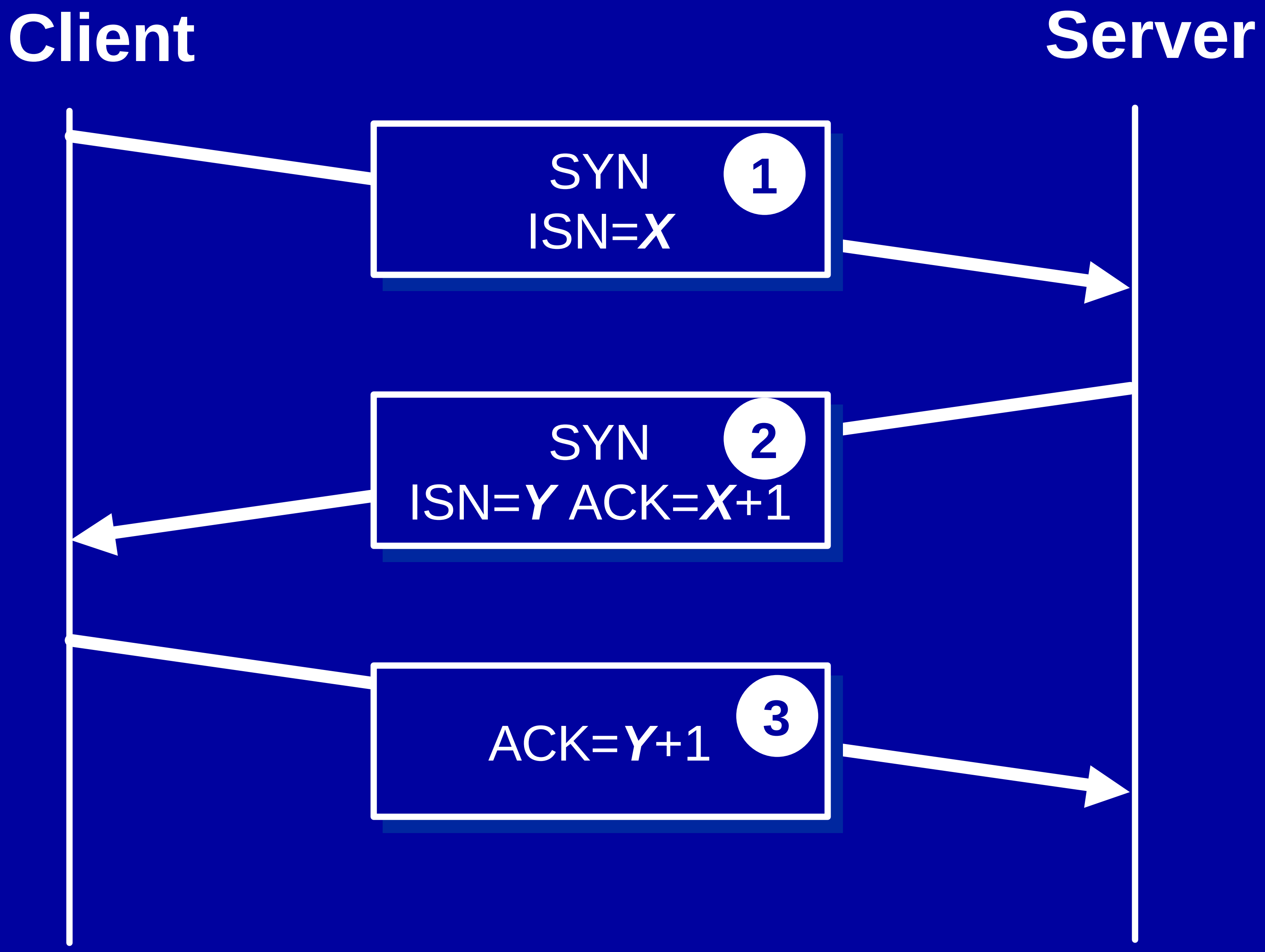
- A client starts by sending a SYN segment with the following information:
  - Client’s ISN (generated pseudo-randomly)
  - Maximum Receive Window for client.
  - Optionally (but usually) MSS (largest datagram accepted).
  - No payload! (Only TCP headers)

Server Response

- When a waiting server sees a new connection request, the server sends back a SYN segment with:
  - Server's ISN (generated pseudo-randomly)
  - Request Number is Client ISN+1
  - Maximum Receive Window for server.
  - Optionally (but usually) MSS
  - No payload! (Only TCP headers)

Finally

- When the Server's SYN is received, the client sends back an ACK with:
  - Acknowledgment Number is Server's ISN+1


Client
Server
SYN
ISN=X
1
SYN
ISN=Y ACK=X+1
2
ACK=Y+1
3

TCP 3-way handshake

1. Client: “I want to talk, and I’m starting with byte number ***X***”.

2. Server: “OK, I’m here and I’ll talk. My first byte will be called number ***Y***, and I know your first byte will be number ***X+1***”.

3. Client: “Got it - you start at byte number ***Y+1***”.

? Bill: “Monica, I’m afraid I’ll syn and byte your ack”

Why 3-Way?

- Why is the third message necessary?

- HINTS:
  - TCP is a reliable service.
  - IP delivers each TCP segment.
  - IP is not reliable.

TCP Data and ACK

- Once the connection is established, data can be sent.
- Each data segment includes a sequence number identifying the first byte in the segment.
- Each segment (data or empty) includes a request number indicating what data has been received.

Buffering

- Keep in mind that TCP is part of the Operating System. The O.S. takes care of all these details asynchronously.
- The TCP layer doesn't know when the application will ask for any received data.
- TCP buffers incoming data so it's ready when we ask for it.

TCP Buffers

- Both the client and server allocate buffers to hold incoming and outgoing data
  - The TCP layer does this.
- Both the client and server announce with every ACK how much buffer space remains (the Window field in a TCP segment).

Send Buffers

- The application gives the TCP layer some data to send.
- The data is put in a send buffer, where it stays until the data is ACK'd.
- The TCP layer won't accept data from the application unless (or until) there is buffer space.

ACKs

- A receiver doesn’t have to ACK every segment (it can ACK many segments with a single ACK segment).
- Each ACK can also contain outgoing data (piggybacking).
- If a sender doesn’t get an ACK after some time limit, it resends the data.

TCP Segment Order

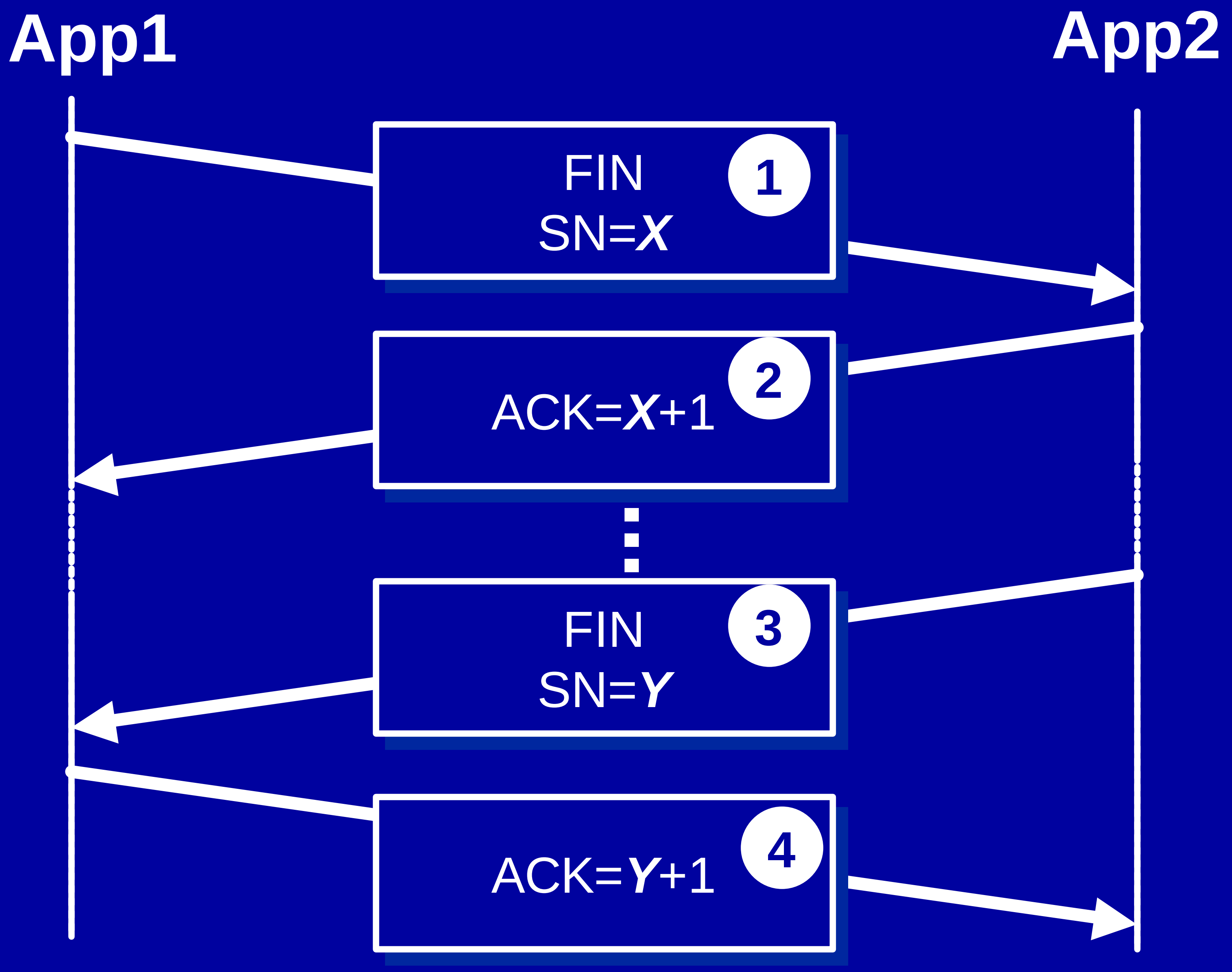
- Most TCP implementations will accept out-of-order segments (if there is room in the buffer).
- Once the missing segments arrive, a single ACK can be sent for the whole thing.
- Remember: IP delivers TCP segments, and IP is not reliable - IP datagrams can be lost or arrive out of order.

Termination

- The TCP layer can send a RST segment that terminates a connection if something is wrong.
- Usually the application tells TCP to terminate the connection politely with a FIN segment.

FIN

- Either end of the connection can initiate termination.
- A FIN is sent, which means the application is done sending data.
- The FIN is ACK'd.
- The other end must now send a FIN.
- That FIN must be ACK'd.


App1
App2
FIN
SN=X
1
ACK=X+1
2
FIN
SN=Y
3
ACK=Y+1
4

TCP Termination

1. App1: “I have no more data for you”.
2. App2: “OK, I understand you are done sending.”

   dramatic pause…

3. App2: “OK - Now I’m also done sending data”.
4. App1: “Roger, Over and Out, Goodbye, Hastalavista Baby, Adios, It’s been real ...”

   camera fades to black ...

TCP TIME_WAIT

- Once a TCP connection has been terminated (the last ACK sent) there is some unfinished business:
  - What if the ACK is lost? The last FIN will be resent and it must be ACK'd.
  - What if there are lost or duplicated segments that finally reach the destination after a long delay?
- TCP hangs out for a while to handle these situations.

TCP Sockets Programming

- Creating a *passive mode* (server) socket.
- Establishing an application-level *connection*.
- Sending/receiving data.
- Terminating a connection.

Establishing a passive mode TCP socket

Passive mode:

- Address already determined.
- Tell the kernel to accept incoming connection requests directed at the socket address.
  - ***3-way handshake***
- Tell the kernel to queue incoming connections for us.

Accepting an incoming connection.

- Once we start listening on a socket, the O.S. will queue incoming connections
  - Handles the 3-way handshake
  - Queues up multiple connections.
- When our application is ready to handle a new connection, we need to ask the O.S. for the next connection.

Terminating a TCP connection

- Either end of the connection can call the **`close()`** system call.
- If the other end has closed the connection, and there is no buffered data, reading from a TCP socket returns 0 to indicate EOF.

Client Code

- TCP clients can connect to a server, which:
  - takes care of establishing an endpoint address for the client socket.
    - don't need to call bind first, the O.S. will take care of assigning the local endpoint address (TCP port number, IP address).
  - Attempts to establish a connection to the specified server.
    - ***3-way handshake***

Reading from a TCP socket

- By default **`read()`** will block until data is available.
- Reading from a TCP socket may return less than max bytes (whatever is available).
- You must be prepared to read data 1 byte at a time!

Writing to a TCP socket

- write might not be able to write all bytes (on a nonblocking socket).

Metaphor for Good Relationships

To succeed in relationships:

bind()

- you need to establish your own identity.
- you need to be open & accepting. ***accept()***
- you need to establish contacts. ***connect()***
- you need to take things as they come, not as you expect them. ***read might return 1 byte***
- you need to handle problems as they arise.

check for errors

Java Sockets Programming

- The package java.net provides support for sockets programming
- Typically you import everything defined in this package with:

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

Classes

`InetAddress`

`Socket`

`ServerSocket`

`DatagramSocket`

`DatagramPacket`

UDP

Socket class

- Corresponds to active TCP sockets only!
  - client sockets
  - socket returned by accept();

- Passive sockets are supported by a different class: ServerSocket

Socket Constructors

- Constructor creates a TCP connection to a named TCP server.
  - There are a number of constructors:

```
Socket(InetAddress server, int port);

Socket(InetAddress server, int port,
  InetAddress local, int localport);

Socket(String hostname, int port);
```

Socket Methods

```
void close();
InetAddress getInetAddress(); getpeername
InetAddress getLocalAddress(); getsockname
InputStream getInputStream();
OutputStream getOutputStream();
```

- Lots more (setting/gettting socket options, partial close, etc.)

Socket I/O

- Socket I/O is based on the Java I/O support (in the package **`java.io`**).
- InputStream and OutputStream are abstract classes
  - common operations defined for all kinds of InputStreams, OutputStreams…

ServerSocket Class (TCP Passive Socket)

- Constructors:

```
ServerSocket(int port);

ServerSocket(int port, int
  backlog);

ServerSocket(int port, int backlog,
  InetAddress bindAddr);
```

ServerSocket Methods

```
Socket accept();

void close();

InetAddress getInetAddress();

int getLocalPort();
```

throw **`IOException, SecurityException`**

Sample Echo Server

TCPEchoServer.java, EchoClient.java, GenericClient.java

Simple TCP Echo server.

Based on code from:

TCP/IP Sockets in Java, Java Online Tutorial