TCP/IP Part II

Based on Notes by D. Hollinger

Based on UNIX Network Programming, Stevens, Chapters 7,11,21,22,27

Also Java Network Programming and Distributed Computing, Chapter 6

Also Online Java Tutorial, Sun.

Topics

- Issues in Client/Server Programming
- Advanced TCP/IP Options
- Sample Application-layer Protocols
  - TELNET
  - FTP

Issues in Client Programming

- Identifying the Server.
- Looking up an IP address.
- Looking up a well known port name.
- Specifying a local IP address.
- UDP client design.
- TCP client design.

Identifying the Server

- Options:
  - hard-coded into the client program.
  - require that the user identify the server.
  - read from a configuration file.
  - use a separate protocol/network service to lookup the identity of the server.

Identifying a TCP/IP server.

- Need an IP address, protocol and port.
  - We often use *host names* instead of IP addresses.
  - usually the protocol (UDP vs. TCP) is not specified by the user.
  - often the port is not specified by the user.

Can you name one common exception ?

Services and Ports

- Many services are available via “well known” addresses (names).
- There is a mapping of service names to port numbers.

Specifying a Local Address

- When a client creates and binds a socket it must specify a local port and IP address.
- Typically a client doesn't care what port it is on:

```
mySocket = new DatagramSocket()
```

give me any available port !

Local IP address

- A client can also ask the operating system to take care of specifying the local IP address:

```
myAddress =
  InetAddress.getLocalHost();
```

Give me the appropriate address

UDP Client Design

- Establish server address (IP and port).
- Allocate a socket.
- Specify that any valid local port and IP address can be used.
- Communicate with server (send, receive)
- Close the socket.

Connected mode UDP

- A UDP client can call connect(address, port) to establish the address of the server.
- “connect” is a misnomer:
  - A UDP client using a connected mode socket can only talk to that server (using the connected-mode socket).

TCP Client Design

- Establish server address (IP and port).
- Allocate a socket.
- Specify that any valid local port and IP address can be used.
- Call connect()
- Communicate with server (through given streams).
- Close the connection.

Transparent to Java programmers

Closing a TCP socket

- Many TCP based application protocols support multiple requests and/or variable length requests over a single TCP connection.
- How does the server known when the client is done (and it is OK to close the socket) ?

Partial Close

- One solution is for the client to shut down only it's writing end of the socket.

- The **`shutdownOutput()`** socket call provides this function.

  ```
  mySocket.shutdownOutput();
  ```

  – shutdownOutput() flushes output stream and sends TCP-connection termination sequence.

  – shutdownInput() closes input stream and discards any further information (further read()s will get -1)

TCP sockets programming

- Common problem areas:
  - null termination of strings.
  - reads don't correspond to writes.
  - synchronization (including close()).
  - ambiguous protocol.

Not a problem with Java Strings.

TCP Reads

- Each call to read() on a TCP socket returns any available data (up to a maximum).
- TCP buffers data at both ends of the connection.
- *You must be prepared to accept data 1 byte at a time from a TCP socket.*

Server Design

Iterative Connectionless

Iterative Connection-Oriented

Concurrent Connectionless

Concurrent Connection-Oriented

Concurrent vs. Iterative

Concurrent

Large or variable size requests
Harder to program
Typically uses more system resources

Iterative

Small, fixed size requests
Easy to program

Connectionless vs. Connection-Oriented

Connection-Oriented

EASY TO PROGRAM
transport protocol handles the tough stuff.
requires separate socket for each connection.

Connectionless

less overhead
no limitation on number of clients

Statelessness

- *State:* Information that a server maintains about the status of ongoing client interactions.
- Connectionless servers that keep state information must be designed carefully!

Messages can be duplicated!

The Dangers of Statefullness

- Clients can go down at any time.
- Client hosts can reboot many times.
- The network can lose messages.
- The network can duplicate messages.

Concurrent Server Design Alternatives

One process per client

Spawn one thread per client

Preforking multiple processes

Prethreaded Server

One child process per client

- Traditional Unix server:
  - TCP: after call to **`accept()`**, call **`getRuntime().exec(), returns Process.`**
  - UDP: after **`receive()`**, call **`exec()`**.
  - Each process needs only a few sockets.
  - Small requests can be serviced in a small amount of time.
- Parent process needs to clean up after children!!!! (invoke **`waitFor()`**).

One thread per client

- Use new Thread().start();
- Using threads makes it easier (less overhead) to have sibling processes share information.
- Sharing information must be done carefully (use synchronized)

Watch out for deadlocks!

Pre-forked Server

- Creating a new process for each client is expensive.
- We can create a bunch of processes, each of which can take care of a client.
- Each child process is an iterative server.

Pre-forked TCP Server

- Initial process creates socket and binds to well known address.
- Process now calls **`exec()`** a bunch of times.
- All children call **`accept()`.**
- The next incoming connection will be handed to one child.

Sockets library vs. system call

- A pre-forked TCP server won't usually work the way we want if *sockets* is not part of the kernel:
  - calling accept() is a library call, not an atomic operation.
- We can get around this by making sure only one child calls accept() at a time using some locking scheme.

Pre-forking

- Having too many pre-forked children can be bad.
- Using dynamic process allocation instead of a hard-coded number of children can avoid problems.
- The parent process just manages the children, doesn’t worry about clients.

Pre-threaded Server

- Same benefits as pre-forking.
- Can have the main thread do all the calls to accept() and hand off each client to an existing thread.

What's the best server design for my application?

- Many factors:
  - Expected number of simultaneous clients.
  - Transaction size (time to compute or lookup the answer)
  - Variability in transaction size.
  - Available system resources (perhaps what resources can be required in order to run the service).

Server Design

- It is important to understand the issues and options.
- Knowledge of queuing theory can be a big help.
- You might need to test a few alternatives to determine the best design.

TCP Socket Options

- It's important to know about some of these topics, although it might not be apparent how and when to use them.
- Details are in the book(s) - we are just trying to get some idea of what can be done.

Socket Options

- Various attributes that are used to determine the behavior of sockets.
- Setting options tells the OS/Protocol Stack the behavior we want.
- Support for generic options (apply to all sockets) and protocol specific options.

Option types

- Many socket options are boolean flags indicating whether some feature is enabled (true) or disabled (false).
- Other options are associated with different data types, e.g. int, representing time.

Read-Only Socket Options

- Some options are readable only (we can’t set the value).

Setting and Getting option values

`get{Option}()` gets the current value of a socket option, e.g.

```
getReceiveBufferSize();
```

`set{Option}()` is used to set the value of a socket option, e.g.

```
setReceiveBufferSize(size);
```

Some Generic Options

SO_BROADCAST

SO_DONTROUTE

SO_ERROR

SO_KEEPALIVE

SO_LINGER

SO_RCVBUF,SO_SNDBUF

SO_REUSEADDR

SO_BROADCAST

- Boolean option: enables/disables sending of broadcast messages.
- Underlying DL layer must support broadcasting!
- Applies only to Datagram (UDP) sockets.
- Prevents applications from inadvertently sending broadcasts (OS looks for this flag when broadcast address is specified).

SO_DONTROUTE

- Boolean option: enables bypassing of normal routing.
- Used by routing daemons.

SO_ERROR

- Integer value option.
- The value is an error indicator value (similar to **`errno`**).
- Readable (get'able) only!
- In Java, a SocketException, or IOException is thrown.

SO_KEEPALIVE

- Boolean option: enabled means that STREAM sockets should send a *probe* to peer if no data flow for a “long time”.
- Used by TCP - allows a process to determine whether peer process/host has crashed.
- Consider what would happen to an open telnet connection without keepalive.

SO_LINGER

- Used to control whether and how long a call to close will wait for pending ACKS.
- connection-oriented sockets only.
- setSoLinger(boolean onFlag, int duration);
- getSoLinger(); returns duration (-1 if option is disabled)

SO_LINGER usage

- By default, calling **`close()`** on a TCP socket will return immediately.
- The closing process has no way of knowing whether or not the peer received all data.
- Setting SO_LINGER means the closing process can determine that the peer machine has received the data (but not that the data has been **`read()`** !).

shutdown() vs SO_LINGER

- You can use **shutdown{In|Out}put()** to find out when the peer process has read all the sent data.

SO_RCVBUF
SO_SNDBUF

- Integer values options - change the receive and send buffer sizes.
- Can be used with TCP and UDP sockets.
- With TCP, this option effects the window size used for flow control - must be established before connection is made.
  - {g|s}et{Send|Receive}BufferSize(…);

SO_REUSEADDR

- Boolean option: enables binding to an address (port) that is already in use.
- Used by servers that are transient - allows binding a passive socket to a port currently in use (with active sockets) by other processes.

SO_REUSEADDR

- Can be used to establish separate servers for the same service on different interfaces (or different IP addresses on the same interface).
- Virtual Web Servers can work this way.

SO_TIMEOUT

- Can be used to tell the socket to use non-blocking read.
- getSoTimeout() returns the current setting (by default 0, or disabled, representing a blocking read).
- E.g. to tell socket to interrupt reading if 5 seconds pass by, use:

```
mySocket.setSoTimeout(5000);
```

IP Options (IPv4)

- IP_TOS: allows us to set the “Type-of-service” field in an IP header.
  - setTrafficClass(int);

another TCP socket option

- TCP_NODELAY: can disable TCP's Nagle algorithm that delays sending small packets if there is unACK'd data pending.
- TCP_NODELAY also disables delayed ACKS (TCP ACKs are cumulative).
- Java Sockets:
  - getTcpNoDelay();
  - setTcpNoDelay(flag);

Out-of-Band Date

- Ever been on a date, gone to a dance club and the band doesn't show up?
  - This is becoming a serious problem:
    - The number of Internet dating services is growing exponentially.
    - The number of bands is not growing.
  - RFC 90210 proposes some short term solutions (until the number of bands can be increased).

Out-of-Band *Data*

- TCP (and other transport layers) provide a mechanism for delivery of "high priority" data ahead of "normal data".
- We can almost think of this as 2 streams:

TCP OOB Data

- TCP supports something like OOB data using URGENT MODE (a bit is set in a TCP segment header).

- A TCP segment header field contains an indication of the location of the urgent data in the stream (the byte number).

Sending OOB Data

```
sendUrgentData(int data);
```

Puts a single byte of urgent data in a TCP stream (lowest 8 bits).

The TCP layer adds some segment header info to let the other end know there is some OOB data.

Receiving OOB Data

- Receiver needs to set OOBInline flag:
  - setOOBInline(true);
- Urgent data is inlined with normal data.
- Very limited support in Java.
  - No special notification of urgent data, and no distinction between normal and urgent data, unless provided by higher-level protocol.

Socket Options Summary

- This was just an overview
  - there are many details associated with the options described.
  - There are many options that haven't been described.
  - UNIX Network Programming is one of the best sources of information about socket options.

Not ALL options are (fully) supported by Java.

The TELNET Protocol

Reference: RFC 854

TELNET vs. `telnet`

- TELNET is a *protocol* that provides “a general, bi-directional, eight-bit byte oriented communications facility”.
- **`telnet`** is a *program* that supports the TELNET protocol over TCP.
- Many application protocols are built upon the TELNET protocol.

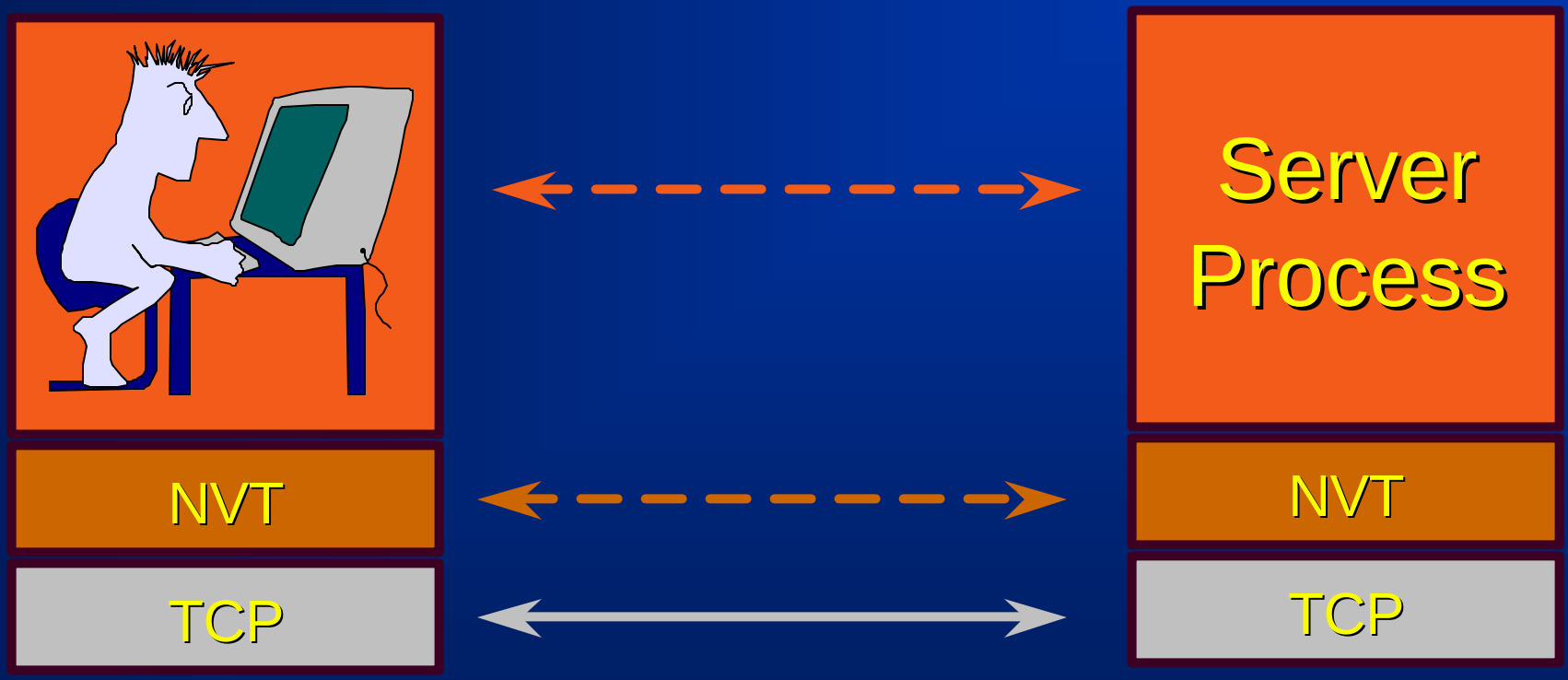
The TELNET Protocol

- TCP connection
- data and control over the same connection.
- Network Virtual Terminal
- negotiated options

Network Virtual Terminal

- intermediate representation of a generic terminal.
- provides a standard language for communication of terminal control functions.

Network Virtual Terminal



Negotiated Options

- All NVTs support a minimal set of capabilities.
- Some terminals have more capabilities than the minimal set.
- The 2 endpoints negotiate a set of mutually acceptable options (character set, echo mode, etc).

Negotiated Options

- The protocol for requesting optional features is well defined and includes rules for eliminating possible negotiation “loops”.
- The set of options is not part of the TELNET protocol, so that new terminal features can be incorporated without changing the TELNET protocol.

Option examples

- Line mode vs. character mode
- echo modes
- character set (EBCDIC vs. ASCII)

Control Functions

- TELNET includes support for a series of control functions commonly supported by servers.
- This provides a uniform mechanism for communication of (the supported) control functions.

Control Functions

- Interrupt Process (IP)
  - suspend/abort process.
- Abort Output (AO)
  - process can complete, but send no more output to user's terminal.
- Are You There (AYT)
  - check to see if system is still running.

More Control Functions

- Erase Character (EC)
  - delete last character sent
  - typically used to edit keyboard input.
- Erase Line (EL)
  - delete all input in current line.

Command Structure

- All TELNET commands and data flow through the same TCP connection.
- Commands start with a special character called the Interpret as Command *escape* character (IAC).
- The IAC code is 255.
- If a 255 is sent as data - it must be followed by another 255.

Looking for Commands

- Each receiver must look at each byte that arrives and look for IAC.
- If IAC is found and the next byte is IAC - a single byte is presented to the application/terminal (a 255).
- If IAC is followed by any other code - the TELNET layer interprets this as a command.

Command Codes

- IP 243
- AO 244
- AYT 245
- EC 246
- EL 247
- WILL 251
- WON’T 252
- DO 253
- DON’T 254
- IAC 255

Playing with TELNET

- You can use the **`telnet`** program to play with the TELNET protocol.
- **`telnet`** is a *generic* TCP client.
  - Sends whatever you type to the TCP socket.
  - Prints whatever comes back through the TCP socket.
  - Useful for testing TCP servers (ASCII based protocols).

Some TCP Servers you can play with

- Many Unix systems have these servers running (by default):
  - **`echo`** port 7
  - **`discard`** port 9
  - **`daytime`** port 13
  - **`chargen`** port 19

telnet hostname port

```
> telnet rcs.rpi.edu 7
Trying 128.113.113.33...
Connected to cortez.sss.rpi.edu
  (128.113.113.33).
Escape character is '^]'.
Hi dave
Hi dave
stop it
stop it
^]
telnet> quit
Connection closed.
```

`telnet` vs. TCP

- Not all TCP servers talk TELNET (most don't)

- You can use the **`telnet`** program to play with these servers, but the fancy commands won't do anything.
  - type ^], then "help" for a list of fancy TELNET stuff you can do in **`telnet`**.

FTP
File Transfer Protocol

Reference:

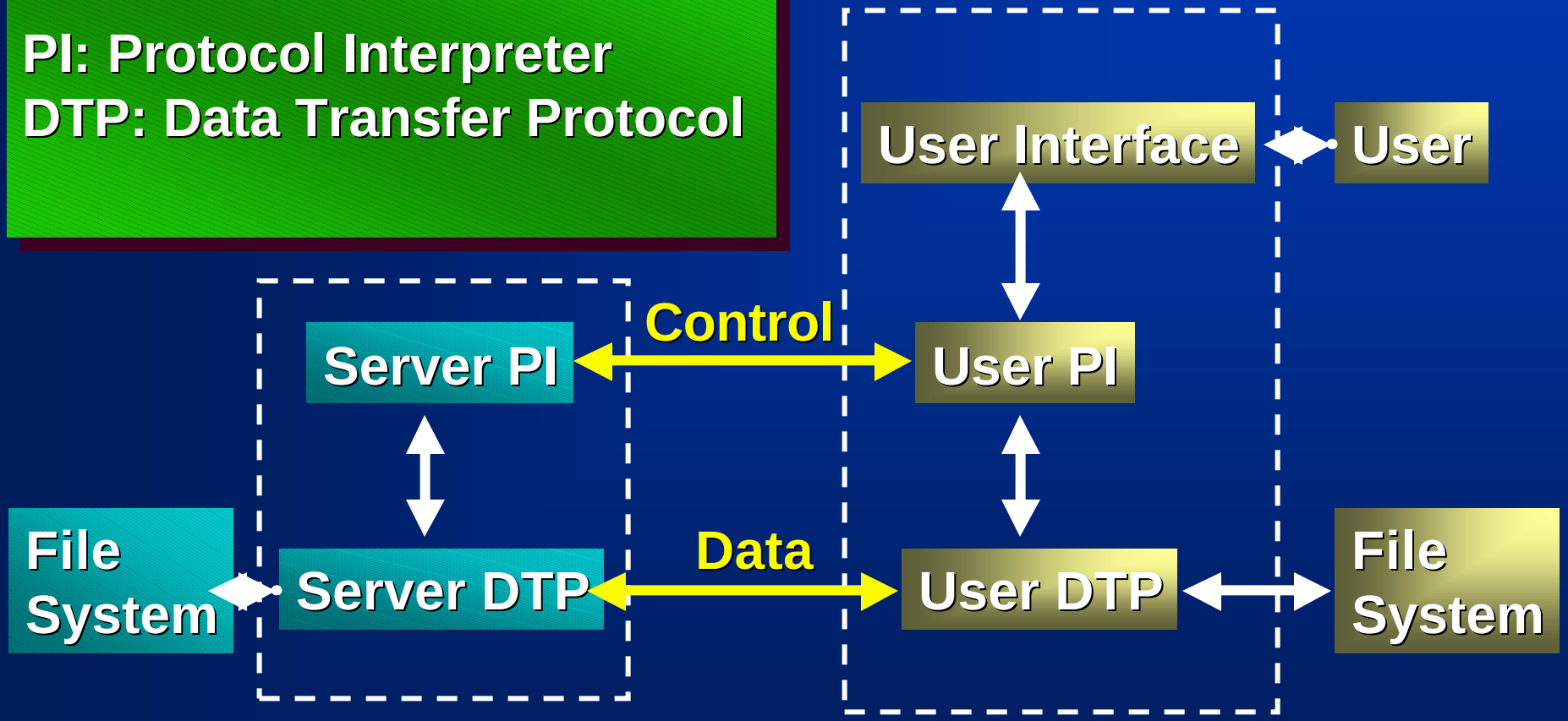
RFC 959

FTP Objectives (from RFC 959)

- promote sharing of files
- encourage indirect use of remote computers
- shield user from variations in file storage
- transfer data reliably and efficiently
- “FTP, although usable directly by a user at a terminal, is designed mainly for use by programs”

The FTP Model

PI: Protocol Interpreter
DTP: Data Transfer Protocol



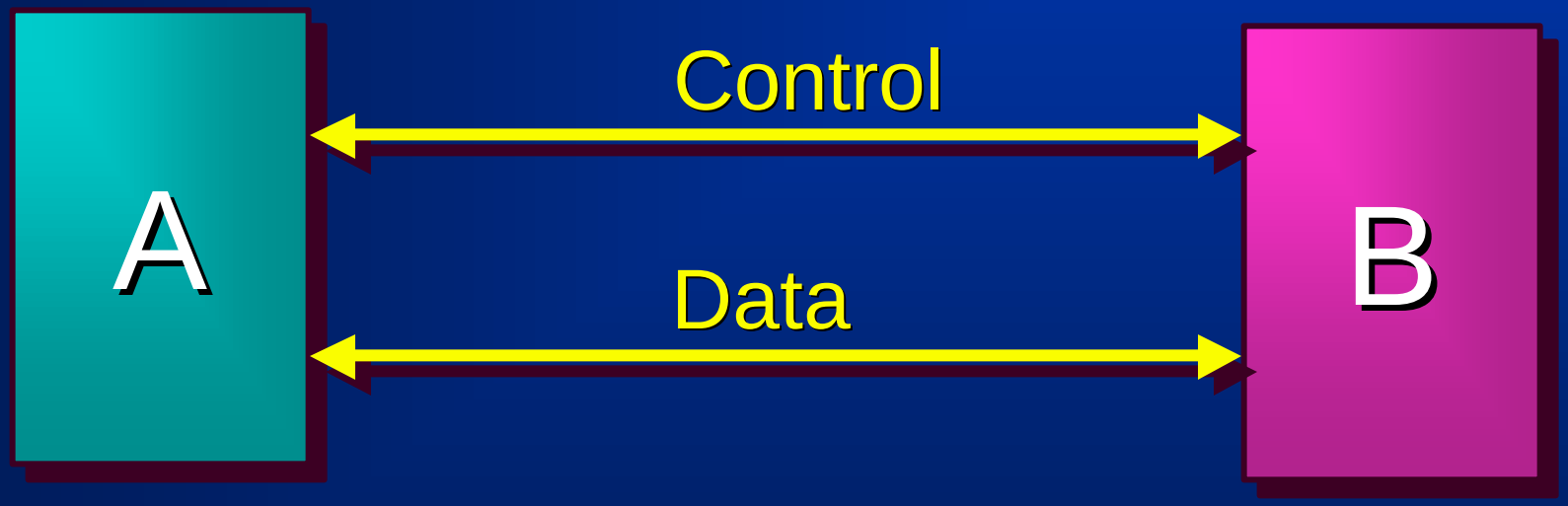
Control and Data Connections

- Control functions (commands) and reply codes are transferred over the control connection.
- All data transfer takes place over the data connection.
- The control connection must be “up” while data transfer takes place.

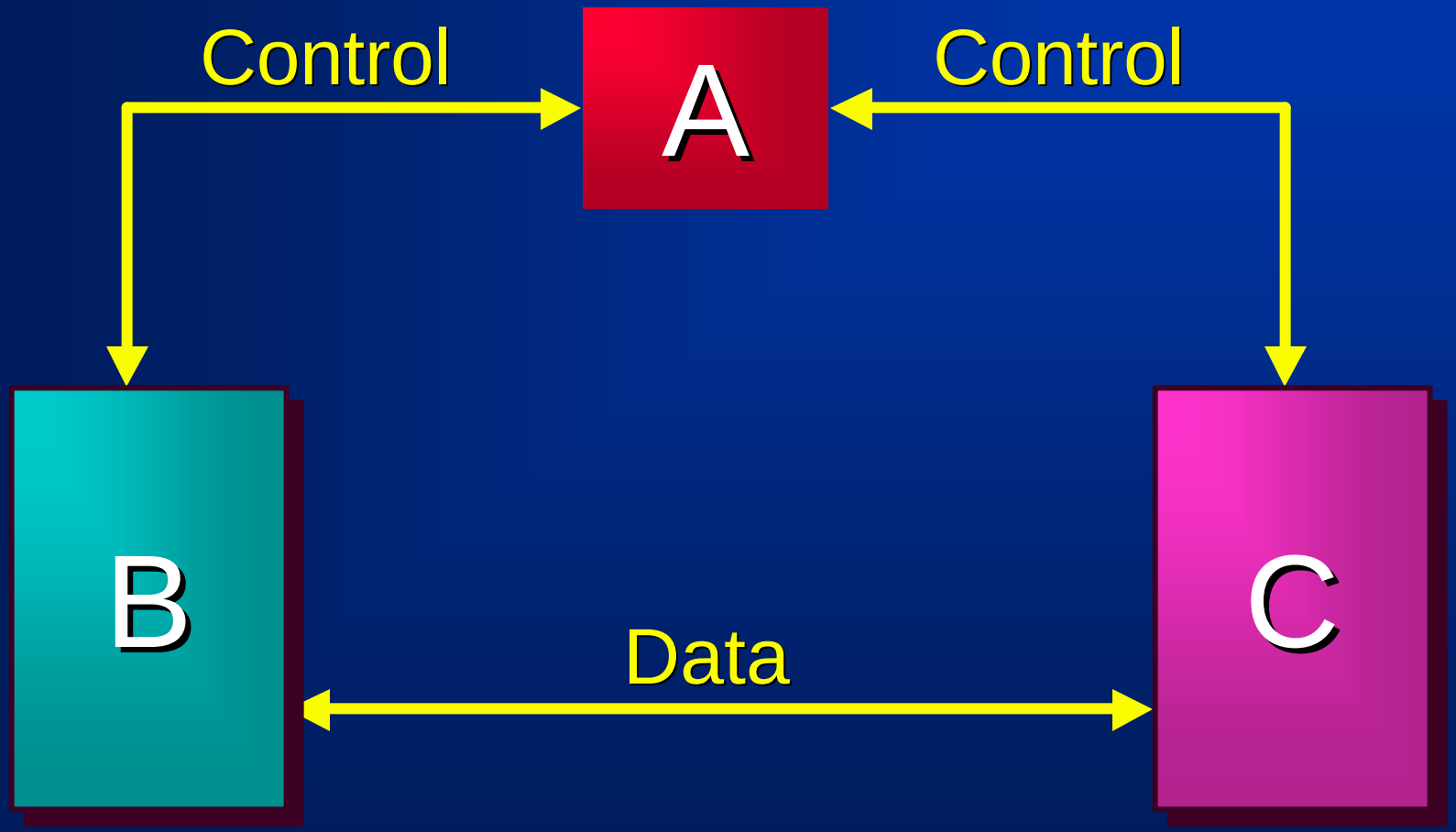
Control Connection

- The control connection is the “well known” service.
- The control connection uses the TELNET protocol.
- Commands and replies are all line oriented text (default is ASCII).

Standard Connection Model



Alternative Connection Model



Access Control Commands

USER	*specify user*
PASS	*specify password*
CWD	*change directory*
CDUP	*change directory to parent*
QUIT	*logout*

Transfer Parameter Commands

PORT	*publish local data port*
PASV	*server should listen*
TYPE	*establish data representation*
MODE	*establish transfer mode*
STRU	*establish file structure*

Service Commands

RETR	*retrieve file*
STOR	*send file*
STOU	*send file and save as unique*
APPE	*send file and append*
ABOR	*abort prev. service command*
PWD	*print working directory*
LIST	*transfer list of files over data link*

FTP Replies

- All replies are sent over control connection.
- Replies are a single line containing
  - 3 digit status code (sent as 3 numeric chars).
  - text message.
- The FTP spec. includes support for multiline text replies.

FTP Reply Status Code

First digit of status code indicates type of reply:

- ‘1’: Positive Preliminary Reply (got it, but wait).
- ‘2’: Positive Completion Reply (success).
- ‘3’: Positive Intermediate Reply (waiting for more information).
- ‘4’: Transient Negative Completion (error - try again).
- ‘5’: Permanent Negative Reply (error - can’t do).

FTP Reply Status Code

- 2nd digit indicates function groupings.
  - ‘0’: Syntax (problem with command syntax).
  - ‘1’: Information (reply to help or status cmds).
  - ‘2’: Connections (problem with a connection).
  - ‘3’: Authentication (problem with login).
  - ‘4’: Unspecified.
  - ‘5’: File system (related to file system).
- 3rd digit indicates specific problem within function group.

Data Transfer Modes

- STREAM: file is transmitted as a stream of bytes.
- BLOCK: file is transmitted as a series of blocks preceded by headers containing count and descriptor code (EOF, EOR, restart marker).
- COMPRESSED: uses a simple compression scheme - compressed blocks are transmitted.

RFC 959

- The RFC includes lots more information and many details including:
  - parameters for commands
  - lists of reply status codes
  - protocol state diagrams
  - support for a variety of file structures
  - sample sessions