

# **CSCI 553: Networking III**

## **Unix Network Programming**

Spring 2007



---

### **Web Client Programming**



# Today's Agenda

---

- Task for Thursday March 13
  - Update project.txt
  - Begin coding
  - use your student repo project directory
- Examples for today

```
[harry@nisl ~]$ mkdir webprog
[harry@nisl ~]$ cd webprog
[harry@nisl webprog]$ tar xvfz /home/csci553/classfiles/webprog.tgz
./
./spider.py
./urllibex.py
...
```
- Web clients & application programming



# Today's Agenda (3/11)

---

- Reminder: Task for Thursday March 13
  - Update project.txt
  - Begin coding
  - use your student repo project directory
- Examples for today: check your www areas

```
[harry@nisl ~]$ cd www
[harry@nisl www]$ ls
cgi-bin  html
[harry@nisl www]$ cd cgi-bin
[harry@nisl cgi-bin]$ tar xvfz /home/csci553/classfiles/webprog2.tgz .
./
./hello_cgi.py
...
[harry@nisl cgi-bin]$ ls
cookie.py  dynamic.py  formhandling.py  form.py  hello_cgi.py  hello.html
```

- Web clients & web server application programming



# Introduction

---

- The Internet is changing everything
  - Distributed programs are different from unitary ones
  - Distributed teams work differently from collocated ones
- This lecture looks at how to build programs that get data from the web
  - The **next lecture** will discuss simple ways to build programs that supply data
  - And the **one after that** will talk about how to do this securely



# Small Pieces, Loosely Joined

---

- The Unix command line was the world's first *component object model*
  - Programmers build small pieces, then connect them in arbitrary ways
- Key features:
  - Low cost of entry: it's easy to add one more tool to the toolbox
  - Common data format: stream of strings
  - Common communication protocol: stdin, stdout, and zero/nonzero exit codes
- The Web grew so quickly because it replicated these strengths
  - Everything used HTML (data format) over HTTP (communication protocol)



# Distributed is Different

---

- Distributed systems are fundamentally different from unitary systems
  - Small programs (like the ones in this lesson) can ignore these differences...
  - ...but every industrial-strength application eventually has to deal with them
- Difference #1: concurrency
  - As in databases, means “several things happening at once”
  - Can lead to:
    - **Deadlock**: A is waiting for B while B is waiting for A
    - **Race conditions**: final result depends on whether A or B goes last



# Partial Failure

---

- Difference #2: partial failure
  - One component fails while others are still healthy
  - If you've waited five seconds for a web site to respond, should you assume that it's down, or keep waiting?
- Both differences make distributed applications *much* harder to debug than unitary ones
  - Often have *heisenbugs* (which only appear intermittently)
  - And it's usually impossible to get a complete picture of the system's state
- Only way to get a distributed system right is to build it right in the first place



# Under the Hood

---

- These days, the Internet runs on a family of standards called *Internet Protocol* (IP)
- *User Datagram Protocol (UDP)* moves *packets* across the network
  - Fast, but no guarantees of delivery or correct ordering
- *Transmission Control Protocol (TCP)* is much more commonly used
  - Guarantees that everything you send is received, in the right order



# Sockets

---

- Using IP, processes communicate through *sockets*
  - Each socket is one end of a point-to-point communication channel
  - Provides the same kind of read and write operations as files
- The socket's *host address* identifies a machine
  - Consists of four 8-bit numbers, like "24.153.22.195"
  - The *Domain Name System (DNS)* gives these symbolic names like "www.third-bit.com"
  - Use nslookup to talk to DNS directly
- The socket's *port* is just a number in the range 0-65535
  - 0-1023 are reserved for the operating system's use



# Client/Server vs. Peer-to-Peer

---

- A *client/server architecture* is one in which many *clients* communicate with a central *server*
  - Asymmetric: clients ask for things, servers provide them
  - *Web servers* are the best-known examples
  - But *database management systems* are also servers
- A *peer-to-peer architecture* is one in which all processes exchange information equally
  - Symmetric: every participant both provides and receives data
- Client/server architectures are simpler to create
  - But if the server fails, the whole system fails



# Socket Client

---

```
import sys, socket

buffer_size = 1024      # bytes
host = '127.0.0.1'      # local machine
port = 19073            # hope nobody else is using it...
message = 'ping!'       # what to send

# AF_INET means 'Internet socket'.
# SOCK_STREAM means 'TCP'.
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((host, port))

# Send the message.
sock.send(message)

# Receive and display the reply.
data = sock.recv(buffer_size)
print 'client received', `data`

# Tidy up.
sock.close()
```



# Socket Server

---

```
import sys, socket

buffer_size = 1024      # bytes
host = ''               # empty string means 'this machine'
port = 19073            # must agree with client

# Create and bind a socket.
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host, port))

# Wait for a connection request.
s.listen(True)
sock, addr = s.accept()
print 'Connected by', addr

# Receive and display a message.
data = sock.recv(buffer_size)
print 'server saw', str(data)

# Replace vowels in reply.
data = data.replace('i', 'o')
sock.send(data)

# Tidy up.
sock.close()
```

# The Hypertext Transfer Protocol

- the *Hypertext Transfer Protocol (HTTP)* specifies how programs exchange documents over the web

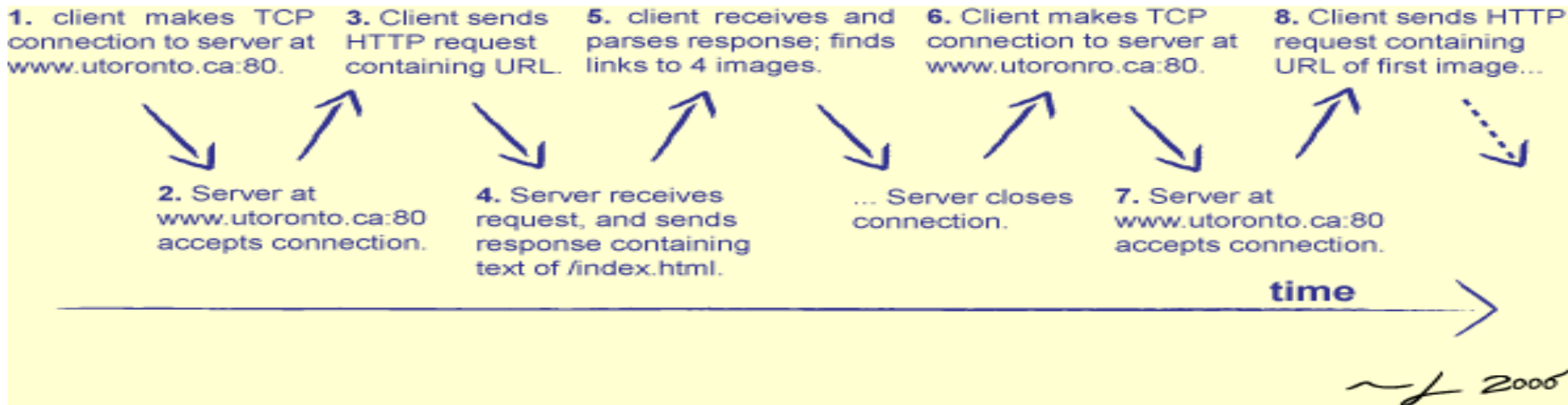


Figure 13.2: HTTP Request Cycle



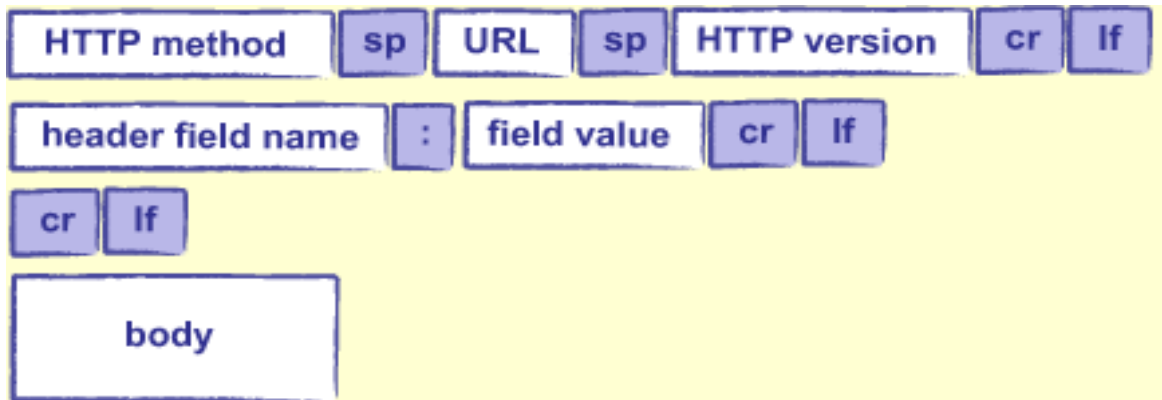
# The Hypertext Protocol

---

- the *Hypertext Transfer Protocol (HTTP)* specifies how programs exchange documents over the web
  - Clients are typically browsers, such as **Firefox** and **Internet Explorer**
  - **Apache** is the most widely used server, but many others exist
- The client sends a request specifying what it wants
- The server sends the contents of the file in reply
  - Or an error message
- HTTP is a *stateless* protocol
  - Server doesn't remember anything between requests
  - Every image in a web page must be requested and downloaded separately

# HTTP Request Line

- An HTTP request has three parts



*~f 2005*

- HTTP method is almost always one of:
  - "GET": to fetch information
  - "POST": to submit form data or upload files
- URL identifies the thing the request wants
  - Typically a path to a file, such as /index.html
  - But it's entirely up to the server how to interpret the URL
- HTTP version is usually "HTTP/1.0"
  - Occasionally see "HTTP/1.1"



# Headers

---

- An *HTTP header* is a key/value pair
  - "Accept: text/html"
  - "Accept-Language: en, fr"
  - "If-Modified-Since: 16-May-2005"
- Unlike a dictionary, a key may appear any number of times
  - So a request can specify that it's willing to accept several types of content

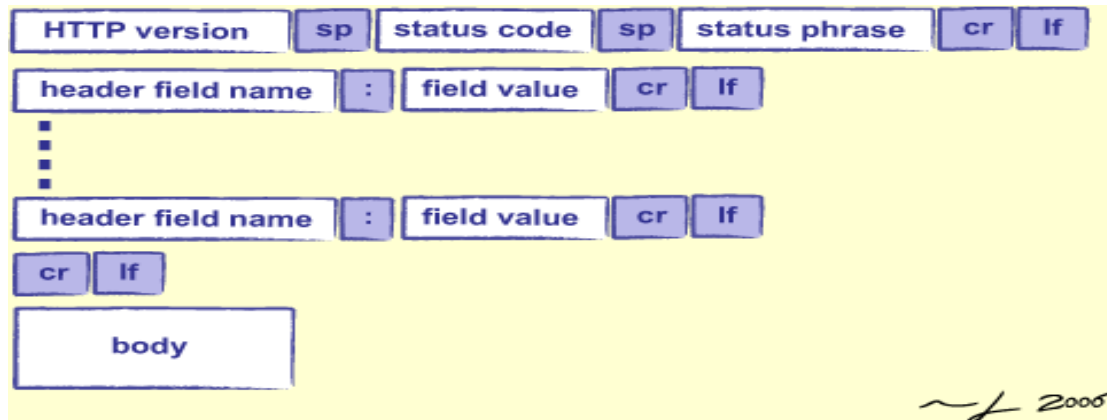


# Body

---

- The body is any extra data associated with the request
  - Used with web forms, to upload files, etc.
- Must be a blank line between the last header and the start of the body
  - Signals the end of the headers
  - Forgetting it is a common mistake
- The "Content-Length" header tells the server how many bytes to read
- Note: there's no magic in any of this
  - An HTTP request is just text—any program that wants to can create them or parse them

# HTTP Response



- HTTP version, headers, and body have the same form, and mean the same thing
- Status code is a number indicating what happened
  - 200: everything worked
  - 404: page not found
- Status phrase repeats that information in a human-readable phrase (like "OK" or "not found")



# HTTP Response Codes

---

<b>Code</b>	<b>Name</b>	<b>Meaning</b>
100	Continue	Client should continue sending data
200	OK	The request has succeeded
204	No Content	The server has completed the request, but doesn't need to return any data
301	Moved Permanently	The requested resource has moved to a new permanent location
307	Temporary Redirect	The requested resource is temporarily at a different location
400	Bad Request	The request is badly formatted
401	Unauthorized	The request requires authentication
404	Not Found	The requested resource could not be found
408	Timeout	The server gave up waiting for the client
500	Internal Server Error	An error occurred in the server that prevented it fulfilling the request
601	Connection Timed Out	The server did not respond before the connection timed out



# HTTP Example

---

- Fetch a page from the course site
  - Request has no headers, so the blank line that signals “end of headers” is right after the request line

```
import sys, socket

buffer_size = 1024

HttpRequest = '''GET /greeting.html HTTP/1.0

'''

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('www.third-bit.com', 80))

sock.send(HttpRequest)

response = ''
while True:
    data = sock.recv(buffer_size)
    if not data:
        break
    response += data
sock.close()

print response
```

- Note: the double parentheses in the call to `sock.connect` are deliberate
  - Method's argument is a (host, port) tuple



# Fetching Pages

---

- Opening sockets, constructing HTTP requests, and parsing responses is tedious
  - So most languages provide libraries to do the work for you
  - In Python, that library is called urllib
- `urllib.urlopen(URL)` does what your browser would do if you gave it the URL
  - Parse it to figure out what server to connect to
  - Connect to that server
  - Send an HTTP request
  - Returns an object that looks like a file, from which to read response data



# urllib Example

---

- Read a page the easy way

```
import urllib
```

```
instream = urllib.urlopen("http://www.third-bit.com/greeting.html")  
lines = instream.readlines()  
instream.close()  
for line in lines:  
    print line,
```

- Note: `readlines` wouldn't do the right thing if the thing being read was an image
  - Might try to convert "line endings"
  - Use `read` to grab the bytes in that case



# Building a Spider

---

- A *web spider* is a program that can explore the web on its own
  - Fetch a page, extract all the external links, visit those pages...
  - That, a search engine, and a few billion dollars, and you're Google



# Building a Spider

---

```
import sys, urllib, re

url = sys.argv[1]
instream = urllib.urlopen(url)
page = instream.read()
instream.close()

links = re.findall(r'href=\"[^\"]+\"', page)
temp = set()
for x in links:
    x = x[6:-1] # strip off 'href=' and '"'
    if x.startswith('http://'):
        temp.add(x)
links = list(temp)
links.sort()
for x in links:
    print x
```



# Passing Parameters

---

- Sometimes want to provide extra information as part of a URL
  - Example: when searching on Google, have to specify what the search terms are
- Could do this as part of the URL
  - Amazon puts ISBNs in URLs
- More flexible to add parameters to the URL
  - `http://www.google.ca?q=Python` searches for pages related to Python
  - "?" separates the parameters from the rest of the URL
  - If there are multiple parameters, they are separated from each other by "&"
    - E.g., `http://www.google.ca/search?q=Python&client=firefox`



# Special Characters

---

- What if you want to include "?" or "&" in a parameter?
  - Same problem (and solution) as including a quote in a string, or <> in XML
- *URL encode* special characters using "%" followed by a 2-digit hexadecimal code
  - And replace spaces with "+"



# Special Characters

---

Character	Encoding
"#"	%23
"\$"	%24
"%"	%25
"&"	%26
"+"	%2B
","	%2C
"/"	%2F
":"	%3A
","	%3B
"="	%3D
"?"	%3F
"@"	%40



# Encoding Example

---

- To search Google for "grade = A+", use `http://www.google.ca/search?q=grade+%3D+A%2B`
- `urllib` has functions to make this easy
  - `urllib.quote(str)` replaces special characters in `str` with escape sequences
  - `urllib.unquote(str)` replaces escape sequences with characters
  - `urllib.urlencode(params)` takes a dictionary and constructs the entire query parameter string

```
import urllib
print urllib.urlencode({'surname' : 'Von Neumann',
                       'forename' : 'John'})
```

```
surname=Von+Neumann&forename=John
```

# Screen Scraping (And Why Not)



---

- Suppose you want to write a script that actually *does* search Google
  - Construct a URL: easy
  - Send it and read the response: no problem
  - Parse the response: there's a lot of junk on the page...
- Many first-generation web applications relied on *screen scraping*
  - “Parse” the HTML with regular expressions
- Hard to get right if the page layout is complex
  - And whenever the layout changes, the application breaks



# Web Services

---

- Modern *web services* separate data from presentation
  - When a client sends a request, it indicates that it wants machine-readable XML, rather than human-readable HTML
    - Much easier to parse
    - Much less likely to change over time

# Web Services

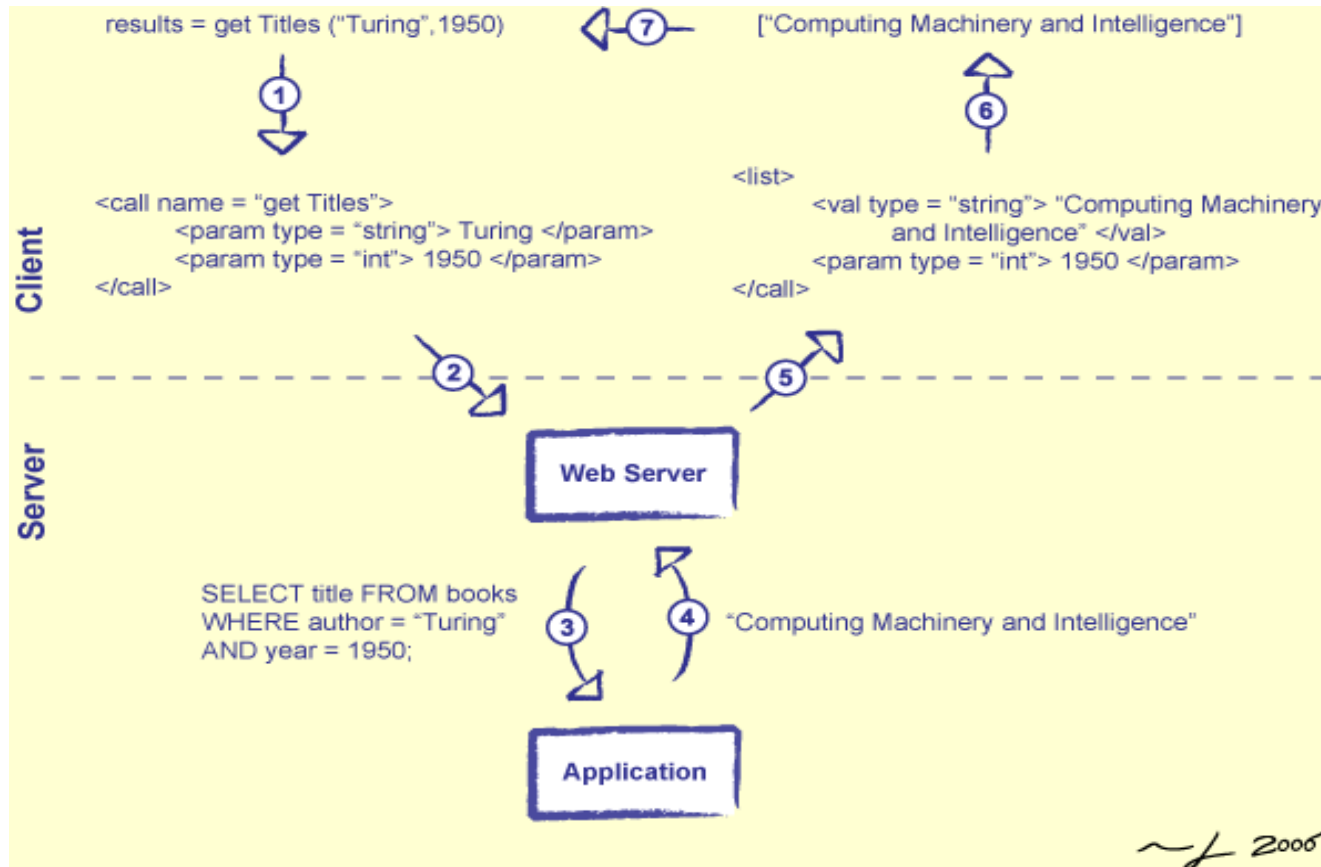


Figure 13.5: Web Services



# Web Services

---

- Many web services use the *Simple Object Access Protocol (SOAP)* standard
  - Despite its name, it's anything but simple
  - Luckily, there are libraries to hide the details for most widely-used web services



# Example: Amazon

---

- Amazon has defined an API for web services
  - You need to get a license key in order to use it
    - They're free
    - But they allow Amazon to throttle requests to one per second per client
- **PyAmazon** turns parameters into URL, and converts the XML reply into Python objects



# Example: Amazon

---

```
import sys, amazon

# Format multiple authors' names nicely.
def prettyName(arg):
    if type(arg) in (list, tuple):
        arg = ', '.join(arg[:-1]) + ' and ' + arg[-1]
    return arg

if __name__ == '__main__':

    # Get information.
    key, asin = sys.argv[1], sys.argv[2]
    amazon.setLicense(key)
    items = amazon.searchByASIN(asin)

    # Handle errors.
    if not items:
        print 'Nothing found for', asin
    if len(items) > 1:
        print len(items), 'items found for', asin

    # Display information.
    item = items[0]
    productName = item.ProductName
    ourPrice = item.OurPrice
    authors = prettyName(item.Authors.Author)
    print '%s: %s (%s)' % (authors, productName, ourPrice)
```



# Summary

---

- Most computers now spend more time communicating than they do calculating
- Every few years, we put another layer on top of the pile of protocols to make communication easier
  - TCP to HTTP to web services to...?
- Getting information from the web is now (almost) as easy as getting it from a file
- See in the next lecture how to *provide* information to others