SOCKETS PROGRAMMING AND THE HTTP PROTOCOL

George Porter
Oct 11, 2018

ATTRIBUTION
• These slides are released under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Creative Commons license
• These slides incorporate material from:
  • Practical TCP/IP Sockets in C, 2nd ed., by Donahoo and Calvert
  • Computer Networks: A Systems Approach, 5e, by Peterson and Davie
  • CMU’s 15-441 Computer Networks, Xi Liu
ANNOUNCEMENTS
HW 2 deadline pushed to next week

HW 1 solutions available

Variety of Python networking resources posted to the website

Outline

1. TCP and sockets overview
2. Socket internals
3. HTTP protocol
4. Demo: interacting with web servers
NAMING PROCESSES/SERVICES

- Process here is an abstract term for your Web browser (HTTP), Email servers (SMTP), hostname translation (DNS)

- How do we identify for remote communication?
  - Process id or memory address are OS-specific and transient

- So TCP and UDP use ports
  - 16-bit integers representing mailboxes
  - Identify process uniquely as (IP address, protocol, port)

PICKING PORT NUMBERS

- We still have the problem of allocating port numbers
  - What port should a Web server use on host X?
  - To what port should you send to contact that Web server?

- Servers typically bind to well-known port numbers
  - e.g., HTTP 80, SMTP 25, DNS 53, ... look in /etc/services
  - Ports below 1024 traditionally reserved for well-known services

- Clients use OS-assigned temporary (ephemeral) ports
  - Above 1024, recycled by OS when client finished
USER DATAGRAM PROTOCOL (UDP)

- Provides **unreliable** *message delivery* between processes
  - Source port filled in by OS as message is sent
  - Destination port identifies UDP delivery queue at endpoint
- Connectionless (no state about who talks to whom)

**UDP DELIVERY**
TRANSMISSION CONTROL PROTOCOL

- Reliable bi-directional bytestream between processes
  - Uses a sliding window protocol for efficient/reliable transfer
- Connection-oriented
  - Conversation between two endpoints with beginning and end
- Flow control
  - Prevents sender from over-running receiver buffers
    - (tell sender how much buffer is left at receiver)
- Congestion control
  - Prevents sender from over-running network capacity

TCP DELIVERY

- Application process
  - Write bytes
  - TCP Send buffer
  - Transmit segments
    - Segment | Segment | ... | Segment
  - Application process
  - Read bytes
  - TCP Receive buffer
STARTUP: THREE-WAY HANDSHAKE

• Opens both directions for transfer

TCP TEARDOWN
CLIENT AND SERVER OPERATIONS

Outline

1. TCP and sockets overview
2. Socket internals
3. HTTP protocol
4. Demo: interacting with web servers
DIGGING INTO SEND() A BIT MORE

```c
rv = connect(s,...);
...
rv = send(s,buffer0,1000,0);
...
rv = send(s,buffer1,2000,0);
...
rv = send(s,buffer2,5000,0);
...
close(s);
```

AFTER 3 SEND() CALLS

![Diagram showing the flow of data between sending and receiving sockets layers and the receiving program](image_url)
AFTER FIRST RECV()

Sending sockets layer

<table>
<thead>
<tr>
<th>SendQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

500 bytes

Receiving sockets layer

<table>
<thead>
<tr>
<th>RecvQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

6000 bytes

Receiving program

<table>
<thead>
<tr>
<th>Delivered</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

1500 bytes

1. First send call (1000 bytes)
2. Second send call (2000 bytes)
3. Third send call (5000 bytes)

AFTER ANOTHER RECV()

Sending sockets layer

<table>
<thead>
<tr>
<th>SendQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

500 bytes

Receiving sockets layer

<table>
<thead>
<tr>
<th>RecvQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

2000 bytes

Receiving program

<table>
<thead>
<tr>
<th>Delivered</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

5500 bytes

1. First send call (1000 bytes)
2. Second send call (2000 bytes)
3. Third send call (5000 bytes)
WHEN DOES BLOCKING OCCUR?

- SendQ size: SQS
- RecvQ size: RQS
- `s.send("n bytes");`
  - `n>SQS`: blocks until `(n – SQS)` bytes xferred to RecvQ
  - If `n>(SQS+RQS)`, blocks until receiver calls `recv()` enough to read in `n-(SQS+RQS)` bytes
- How does this lead to deadlock?
  - Trivial cause: both sides call `recv()` w/o sending data

MORE SUBTLE REASON FOR DEADLOCK

- SendQ size = 500; RecvQ size = 500
## Outline

1. TCP and sockets overview
2. Socket internals
3. HTTP protocol
4. Demo: interacting with web servers

## HTTP AS AN EMERGING TRANSPORT LAYER

- HTTP: HyperText Transfer Protocol
  - Tim Berners-Lee at CERN in 1989

- In addition to web browsing:
  - Video streaming via DASH on YouTube.com
  - REST (Representational state transfer)
  - Chat apps like Slack
  - Many others
WEB/HTTP OVERVIEW

- Documents link to other documents
  - Specified in HTML files
- HTTP is the protocol for retrieving HTML files from servers
  - and images, sounds, video, ...
- Implemented in servers
  - Apache, nginx, MSFT IIS
- and clients
  - Chrome
  - MSFT Edge
  - Apple Safari...

SAMPLE HTML FILE

```
<html>
<head>
<title>It works!</title>
</head>
<body>
<h1>It works</h1>
nice!
</body>
</html>
```
HTTP OVERVIEW

• HTTP is a text oriented protocol.
• HTTP is a request/response protocol
• Requests and responses both look like:

  START_LINE <CRLF>
  MESSAGE_HEADER <CRLF>
  MESSAGE_BODY <CRLF>

• The first line (START LINE) indicates whether this is a request message or a response message.

HTTP REQUESTS

• Request Messages define
  • The operation (called method) to be performed
  • The web page the operation should be performed on
  • The version of HTTP being used.
• Examples:
  • GET /index.html HTTP/1.0
  • GET /images/catimg23.jpg HTTP/1.1
  • GET /contracts/contract3.txt HTTP/1.1
OPTIONAL HTTP REQUEST HEADERS

• After the start line are request headers:
  • Text-based, key and value separated by a colon
  • Example 1:
    GET /index.html HTTP/1.0
    User-Agent: Firefox 23.3.1

• Example 2:
  GET /images/cat2.jpg HTTP/1.1
  Host: www.cs.ucsd.edu
  User-Agent: Chrome 12.1

HTTP RESPONSES

• Also begins with a single START LINE.
  • The version of HTTP being used
  • A three-digit status code
  • Text string giving the reason for the response.
  • Example:
    HTTP/1.1 200 OK
    Content-Type: text/html
    Content-Length: 291
```
borabora:~ gmporter$ telnet oec-vmweb09.ucsd.edu 80
Trying 132.239.8.67...
Connected to oec-vmweb09.ucsd.edu.
Escape character is '^]'.
```

```
GET /index.html HTTP/1.0
```

```
HTTP/1.1 200 OK
Date: Mon, 12 Jan 2015 19:36:37 GMT
Server: Apache/2.2.22 (Ubuntu)
Last-Modified: Thu, 28 Feb 2013 17:35:36 GMT
ETag: "fc7b21-a-4d6cc51858aec"
Accept-Ranges: bytes
Content-Length: 10
Vary: Accept-Encoding
Connection: close
Content-Type: text/html

It works!
Connection closed by foreign host.
borabora:~ gmporter$
```

### HTTP RESPONSE CODES

<table>
<thead>
<tr>
<th>Code</th>
<th>Type</th>
<th>Example Reasons</th>
</tr>
</thead>
<tbody>
<tr>
<td>1xx</td>
<td>Informational</td>
<td>request received, continuing process</td>
</tr>
<tr>
<td>2xx</td>
<td>Success</td>
<td>action successfully received, understood, and accepted</td>
</tr>
<tr>
<td>3xx</td>
<td>Redirection</td>
<td>further action must be taken to complete the request</td>
</tr>
<tr>
<td>4xx</td>
<td>Client Error</td>
<td>request contains bad syntax or cannot be fulfilled</td>
</tr>
<tr>
<td>5xx</td>
<td>Server Error</td>
<td>server failed to fulfill an apparently valid request</td>
</tr>
</tbody>
</table>

- For project 1:
  - 200: OK
  - 400: Client Error
  - 403: Forbidden
  - 404: Not Found
HTTP PIPELINING (VERSION HTTP/1.1)

- HTTP/1.0 opened a new connection for every data item it retrieved
- Overhead in establishing a new connection to the same server over and over again
- HTTP/1.1 Persistent Connections
  - Reuse connection over many requests/responses
  - But more complex in terms of framing/parsing
    - How to know when one request ends and the next begins?
    - This is part of the 1.1 spec

REQUIRED REQUEST HEADERS (AT LEAST FOR US)

- Host:
  - Indicates the name of the server you are accessing
  - Used to implement virtual hosts
- User-Agent:
  - Identifies what software is issuing the request
  - E.g.:
    - User-Agent: Opera/9.25 (Windows NT 6.0; U; en)
    - User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en)
      AppleWebKit/125.2 (KHTML, like Gecko) Safari/125.8
REQUIRED RESPONSE HEADERS (AT LEAST FOR US)

- **Server:**
  - Identifies the server
  - *Server: Apache/2*

- **Content-Length:**
  - How many octets (byte) in the response

- **Content-Type:**
  - *text/html*
  - *image/jpeg*
  - *image/png*

Outline

1. TCP and sockets overview
2. Socket internals
3. HTTP protocol
4. Demo: interacting with web servers
DEMO: INTERACTING WITH WEB SERVERS

• Usage:
  • curl -v -o /dev/null http://<URL>
  • curl -v -o /dev/null https://<URL>