INTRODUCTION TO PROTOCOLS, LAYERING, FRAMING AND PARSING

George Porter
Oct 4, 2017

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:
  • Computer Networks: A Systems Approach, 5e, by Peterson and Davie
  • Michael Freedman and Kyle Jamieson, Princeton University (also under a CC BY-NC-SA 3.0 Creative Commons license)
ANNOUNCEMENTS
Homework 1 now available

Project 1 will be available soon

For Friday: Reading due: “The most expensive one-byte mistake”
and our first In-class exercise [Bring your laptop if you have one!]

Outline

1. Protocols and layering
2. Encoding data
3. Framing and Parsing
**WHAT ARE PROTOCOLS?**

- Explicit and implicit conventions for how to communicate
- Not for what is communicated

**NETWORK PROTOCOLS: PEER INTERFACE**

- Defines operations and format for communicating between peers in a distributed system
- E.g., how a web client communicates with a web server via HTTP protocol
  - Defines format and content of HTTP requests and responses
  - Allows _decoupled_ development of client(s) and server(s)
  - Web servers can support different web browsers (Chrome, Safari, Edge, ...)
THE PROBLEM OF COMMUNICATION

- Re-implement every application for every new underlying transmission medium?
- *Change every application* on any change to an underlying transmission medium?
- No! But how does the Internet design avoid this?

SOLUTION: LAYERING

- Intermediate *layers* provide a set of abstractions for applications and media
- New applications or media need only implement for intermediate layer’s interface
The service interface is presented to a higher layer in a software stack.

This interface enables abstraction by hiding details of the lower layers.

Ideally is reusable, general enough to support multiple higher layers.

Example: TCP offers a “reliable byte stream” abstraction to the network.

Where do protocols come from?

- Standards bodies
  - IETF: Internet Engineering Task Force
  - ISO: International Standards Organization
- Community efforts
  - “Request for comments”
  - Bitcoin
- Corporations/industry
  - RealAudio™, Call of Duty multiplayer, Skype
**How Are Protocols Specified?**

**Prose/BNF**

```plaintext
These rules shows a field syntax, without regard for the particular type or internal syntax. Their purpose is to permit detection of fields, also, they prevent to higher-level parsers an image of each field as fitting on one line.

field = field-name '.' [ field-body ] CRLF
field-name = 1*any CHAR, excluding (\t, \n, \r, and ";")
field-body = field-body-contents
field-body-contents = (the ASCII characters making up the field-body, as defined in the following sections, and consisting of combinations of one, quoted-string, and special tokens, or else consisting of text)
```

**State Transition Diagrams**

**Message Sequence Diagram**

```
Peggy  Victor

x = r^2 mod n

y = rs^n mod r

y^2 = x*y^2 mod r

By Stefan Birkner, cc-by-sa-2.5.2.0.1.0
```

**Packet Formats**

**Role of Layering in Protocols**

- Each layer offers useful semantics to layer above
- IP gets packets to a destination host/server on the Internet (but is unreliable)
- TCP uses IP to offer *reliable, in-order bytestream* abstraction
- TCP useful for file transfer, as well as HTTP/web
Outline

1. Protocols and layering
2. Encoding data
3. Framing and Parsing

ENCODING INTEGER TYPES

- C’s ‘int’, ‘long’, ... not well defined
- Use ‘standard’ int types instead:
  - #include <stdint.h>
  - int32_t, int8_t, uint64_t, uint8_t, ...
**ENDIAN-NESS REVIEW**

- x86-64 is a little endian architecture
  - Least significant byte of multi-byte entity at lowest memory address
  - “Little end goes first”
- Some other systems use big endian
  - Most significant byte of multi-byte entity at lowest memory address
  - “Big end goes first”

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000 :</td>
<td>0000  0101</td>
</tr>
<tr>
<td>0x1001 :</td>
<td>0000  0000</td>
</tr>
<tr>
<td>0x1002 :</td>
<td>0000  0000</td>
</tr>
<tr>
<td>0x1003 :</td>
<td>0000  0000</td>
</tr>
</tbody>
</table>

**NETWORK BYTE ORDER**

- Endian-ness: Use “network byte order”

```c
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```
ENCODING STRINGS

• Strings are arrays of characters
• But what is a character?
• Defined by the choice of Character Set
  • Specified as part of the protocol or negotiated during connection setup
• For this class, we’ll be using 1-byte US-ASCII

ENCODING BINARY STRUCTURES

```c
struct addressInfo {
    uint16_t streetAddress;
    int16_t aptNumber;
    uint32_t postalCode;
} addrInfo;
```

```c
struct integerMessage {
    uint8_t oneByte;
    uint16_t twoBytes;
    uint32_t fourBytes;
    uint64_t eightBytes;
}
```
FILE* STREAMS

- FILE streams can abstract on-disk files, as well as network connections (TCP sockets)
  - FILE * fopen(“foo.dat”, “wb”)
  - FILE * fdopen(int socket, const char * mode)
  - fwrite(), fread(), fflush(), fclose()
- Benefits:
  - They are buffered (minimize context switches)
  - They can read/write complex fixed-length objects

Outline

1. Protocols and layering
2. Encoding data
3. Framing and Parsing
DEFINITIONS

• Operation (e.g., in a voting system)
  • An action you can perform within a protocol’s peer interface
  • E.g., “Submit vote”, “get current vote count”, “reset vote count to zero”
• Message
  • An encoding of an operation according to a protocol’s wire format. Common formats include XML, binary, JSON, ...
• Framing
  • Writing out (and reading in) messages from a stream such that messages can be separated and interpreted correctly
• Parsing/encoding/decoding
  • Converting a message to/from an application data structure

EN/DECODING APP STATE TO/FROM A MESSAGE

```
struct Employee_t {
    uint8_t operation;
    uint64_t id;
    uint16_t department;
};
typedef struct Employee_t Employee;

// list of operations
enum {
    ADD_EMPLOYEE = 1,
    QUERY = 2,
    DELETE_EMPLOYEE = 3;
};
```

- Binary
  - 8 64 16
  - operation id department

- Text (ad-hoc)
  - “OP=1, id=428, d=80”
  - (1,428,80)

- Text (XML)
  - `<employee>`
  - `<operation>1</operation>`
  - `<id>428</id>`
  - `<department>80</department>`
  - `</employee>`

- Many others...
FRAMING: LENGTH SPECIFICATION VS DELIMITERS

- Binary representation of name?
  - Handling variable length
- Consider “Alan” as a name

  ```
  97 108 97 110
  ```

- Option 1: Explicit length

  ```
  4 97 108 97 110
  ```

  - But how big should length be?
- Option 2: Delimiter

  ```
  97 108 97 110 0
  ```

  - But what if delimiter is in the message?

FRAMING: GETNEXTMSG AND PUTMSG

- GetNextMsg()
  - Finds and returns bytes corresponding to single message
  - Even if messages are variable length
- PutMsg()
  - Writes out bytes corresponding to a message with enough context for GetNextMsg to work
FRAMING SCENARIO

• Consider the voting system from the book
• Each message is variable length
  • “Voting v 134” → [Vote for candidate 134]
  • “Voting i 19381”
    • → [Query candidate 19381’s vote count]
• First is 12 characters, second is 14 characters
• Given a stream of vote operations, how to separate them?

FRAMING CHOICES

• Delimiter (in this case ‘$’)

Voting v 134$Voting v 2817$Voting i 9172651$Voting v 2$Voting i 1900$Voting v 32$Voting i 8

• Length + message

12Voting v 13413Voting v 281716Voting i 917265110Voting v 213Voting i 190011Voting v 3210Voting i 8
DYNAMIC ARRAYS REVIEW

• Growable array of bytes
  • Underpins classes like Vector, ArrayList, etc...
• State variables:
  • uint64_t capacity;
  • uint8_t buffer[capacity]
  • uint64_t size
• Appending item ‘myItem’:
  • size = size + 1
  • buffer[size] = myItem;
• Accessing item ‘i’:
  • return buffer[i];
• But what if array is full?
  • I.e., (size == capacity)?
• Doubling algorithm:
  • allocate newbuffer[2 * capacity];
  • copy buffer into newbuffer
  • Replace pointer in the data structure with a pointer to newbuffer
  • Free the original buffer
• Amortized cost to insert?
  • O(1)

READING REQUEST OF UNKNOWN SIZE INTO A DYNAMIC BUFFER

dynamic_buffer requestBuffer;
uint8_t readBuffer[512];
while(...) {
  read up to 512 bytes from client into readBuffer
  requestBuffer.append(readBuffer);
  Does requestBuffer contain a full request?
    If yes,(1) parse it, then (2) remove from requestBuffer
    Otherwise, keep reading from the client
}
HOW TO TELL IF BUFFER CONTAINS A COMPLETE REQUEST?

• This is the framing problem
• For length-based framing:
  • Keep reading until we have 12 bytes of request data
• For delimiter-based framing:
  • OK to simply scan for delimiters using e.g., a for() loop
  
  for(int i = 0; i < requestBuffer.size(); i++)
      if (requestBuffer[i] == '\r') …

FRAMING: SUMMARY

• PutMsg()
• Given an array of bytes representing an application-level operation, writes to stream
  1. Explicit length
      • Writes out the length of the message, then message
  2. Delimiter
      • Ensures delimiter doesn’t appear in message
      • Writes out message
      • Then writes out delimiter

• GetNextMsg()
• Reads from stream until entire message is read, returns to higher layer
  1. Explicit length
      • Reads the length, then reads that many bytes (security?)
  2. Delimiter
      • Reads continuously into a buffer until delimiter is encountered
      • Message then returned to higher layer
HW1: A SIMPLE CALCULATOR INSTRUCTION SET

- Instructions separated by <CR> <LF> two-byte delimiter
- Groups of instructions separated by a pair of <CR><LF> delimiters
- Goal:
  - Working with delimiters in a simple protocol, without all of HTTP

Q&A

Homework 1 now available

Project 1 will be available soon

For Friday: Reading due: “The most expensive one-byte mistake” and our first In-class exercise [Bring your laptop if you have one!]