CSE 124: INTRODUCTION TO PROTOCOLS, LAYERING, FRAMING AND PARSING

George Porter
Oct 2, 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 now available

For Friday: Reading due: Donahoo and Calvert, Chapter 3
Outline

1. Protocols and layering
2. Encoding data
3. Framing and Parsing
4. Demo (if time)
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, ...)

![Diagram of network protocols](image)
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?
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, it is reusable and 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

3.2. HEADER FIELD DEFINITIONS

These rules show a field meta-syntax, without regard for the particular type or internal syntax. Their purpose is to permit detection of fields; also, they present 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 CTLs, SPACE, and "">  
field-body = field-body-contents  
[ CRLF LWSP-char field-body ]  
field-body-contents =  
< the ASCII characters making up the field-body, as  
defined in the following sections, and consisting  
of combinations of atom, quoted-string, and  
specials tokens, or else consisting of texts >

State transition diagrams

Packet formats

Message Sequence Diagram

By Stefan Birkner, cc-by-sa-2.5,2.0,1.0
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
- More on this Friday.
1. Protocols and layering
2. Encoding data
3. Framing and Parsing
4. Demo (if time)
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, ...
• 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”

int 5 at address 0x1000:

<table>
<thead>
<tr>
<th>Address</th>
<th>Binary</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>
• 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
4. Demo (if time)
• 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

- Binary
- Text (ad-hoc)
- Text (XML)
- Many others...

```c
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;
};
```

Operation: 8 64 16
```
"OP=1, id=428, d=80"

(1,428,80)

XML:
```
<employee>
    <operation>1</operation>
    <id>428</id>
    <department>80</department>
</employee>
```
FRAMING: LENGTH SPECIFICATION VS DELIMITERS

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

Option 1: Explicit length
- But how big should length be?

Option 2: Delimiter
- But what if delimiter is in the message?

```c
struct Employee_t {
    uint8_t operation;
    char * name;
    uint64_t id;
    uint16_t department;
};
typedef struct Employee_t Employee;

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

1. Protocols and layering
2. Encoding data
3. Framing and Parsing
4. Demo
COMPANION CODE

- https://github.com/gmporter/cse124-lec-protocols
Q&A

Homework 1 now available

Project 1 now available

For Friday: Reading due: Donahoo and Calvert, Chapter 3