PROTOCOL FRAMING AND PARSING

George Porter
Jan 24, 2023
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)
WHAT ARE PROTOCOLS?

- Explicit and implicit conventions for how to communicate
  - Not for what is communicated
- Enables heterogeneous architectures, languages, OSes, byte ordering, …
SERVICE AND PROTOCOL INTERFACES

Party A

Layer N

Layer N-1

Interface

Service

Protocol

Party B

Layer N

Layer N-1
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**

By Stefan Birkner, cc-by-sa-2.5,2.0,1.0
EXAMPLE: A SIMPLE VOTE COUNTING SYSTEM
• Operation (e.g., in a voting system)
  • An action you can perform within a protocol’s service 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
PARSING: CONverting IN-MEMLORY REPRESENTATION WITH A “WIRE” REPRESENTATION

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

```go
type Employee struct {
  operation uint8
  id uint64
  department uint16
}
```

```
<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?

```go
type Employee struct {
    operation uint8
    name string
    id uint64
    department uint16
}
```
• Framing

  • Finds and returns bytes corresponding to single message

  • Even if messages are variable length

  • Writes out bytes corresponding to a message with enough context for the other side to determine the message boundaries
FRAMING SCENARIO

• Consider a voting scenario

• 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
THE MAIN LOOP OF YOUR SERVER

Remaining := ""

buf := make([]byte, 1024)

for {
    for "Does remaining contain a full request?" {
        If yes,(1) parse it, then (2) remove from remaining
    }
    size, err := c.Read(buf)
    data := buf[:size]
    remaining = remaining + string(data)
}

How do you know when a request is completed?
HOW TO TELL IF BUFFER CONTAINS A COMPLETE REQUEST?

• This is the framing problem

• For length-based framing:
  
  12 97 108 97 ...

  • 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
FRAMING: SUMMARY

• Writing data
  • 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

• Reading data
  • 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
• [see turing-printer.go and turing-sender.go demo]