Project 1: A Web Server Called Liso

15-441/641 Computer Networks

Kenneth Yang
Viswesh Narayanan

"What happens when you type google.com into your browser's address box and press enter?"

- ……
- Establish a TCP connection - Checkpoint 1
- TLS handshake (HTTPS) - Checkpoint 3
- Send HTTP request
- Server generate HTTP response - Checkpoint 2 & 3
- ……

Project 1 Overview

Checkpoint 1

- Socket programming
- I/O multiplexing with select
- Request parsing with Lex/Yacc

Checkpoint 2

- Persistent connection
- Generate response for GET, HEAD, POST

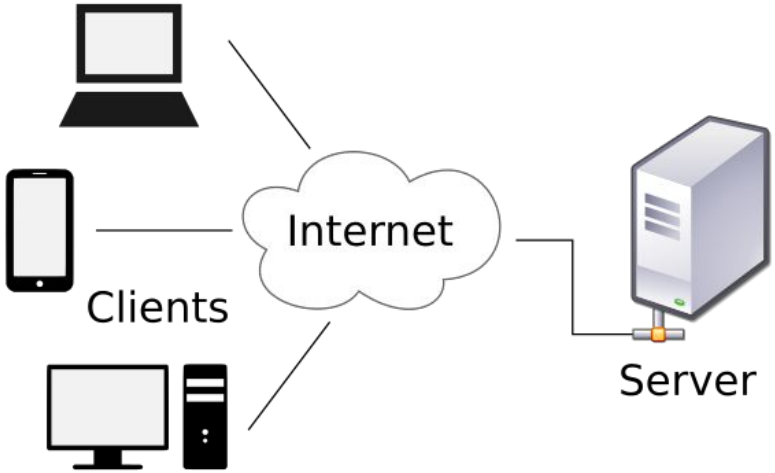
Final submission

- HTTPS
- CGI

Extras

- Design documentation
- Your own tests

Basic concept: Client–server model



Basic concept: Socket

Socket: endpoint <ip, port>

Connection: Source endpoint <ip, port> + Destination endpoint <ip, port>

Establish a TCP connection (client)

- Create a socket via `socket()`
- Connect to an endpoint via `connect()`
- Write to socket via `send()`
- Read from socket via `recv()`

Establish a TCP connection (server)

- Create a socket via `socket()`
- Bind to an endpoint via `bind()`
- Listen for connections via `listen()`
- Accept connections via `accept()`
- Read from socket via `recv()`
- Write to socket via `send()`

Establish a TCP connection

Review 15-213/18-213/15-513 proxy lab.

Check starter code for example usage.

Use man page.

Gotchas

Endianness Matters: Network Byte Order

- `htons()` - host to network short
- `ntohs()` -network to host short

Cleanup state—avoid memory leaks

- `freeaddrinfo()`
- `close()`
- `free()` - check memory leak with valgrind

Error Handling

- Tedious, but worth it (and required!)

Gotchas

Timeouts

- Implement for robust networking behavior

Prepare your code for random failures

- We introduce random faults when grading
- Test too - ctrl+c server and client randomly

Concurrency

Multithreading

- Server gives each client its own thread
- Not in this class!

I/O Multiplexing

- Watch a set of sockets (in main thread)
- Use `select()` to find sockets ready for I/O
- Server-side only - clients are agnostic

“Everything is a file”

File descriptor (fd)

- a non-negative integer
- text file, pipe, socket etc.
- stdin-0, stdout-1, stderr-2

`select()`

- manage file descriptors and examine their status

select()

```
#include <sys/select.h>
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
void FD_CLR(int fd, fd_set *set);     // removes fd from set
int FD_ISSET(int fd, fd_set *set);    // is fd in set?
void FD_SET(int fd, fd_set *set);     // adds fd to set
void FD_ZERO(fd_set *set);            // clears set
```

Useful reference for select():

- Beej's Guide to Network Programming: http://beej.us/guide/bgnet/output/html/multipage/index.html
- `man select`

Advice from past TAs

Check return value of `recv()`

- Error – handle the error and clear up state
- If peer shutdown connection, clear up state

Maintain state

- Maintain a read buffer
- Keep track of number of bytes to read
- May need multiple reads to get all data (Don't worry about this in checkpoint 1)
- Only one read per socket when select() returns

Advice from past TAs

Check return value of `send()`

- Error – handle the error and clear up state
- If peer shutdown connection, clear up state

Maintain state

- Maintain a write buffer
- Keep track of number of bytes to write
- May need multiple writes to send all data (Don't worry about this in checkpoint 1)

Remember

Code quality

- Clean code with appropriate comments will help you debug

Robustness

- Handle errors
- Be careful with buffer overflow
- Connection reset by peer

Efficiency

- Avoid extra or useless buffer copy

Common mistakes

Make sure your executable is named correctly.

tar your complete repo and submit. Autolab expects to find a .git folder within your submission

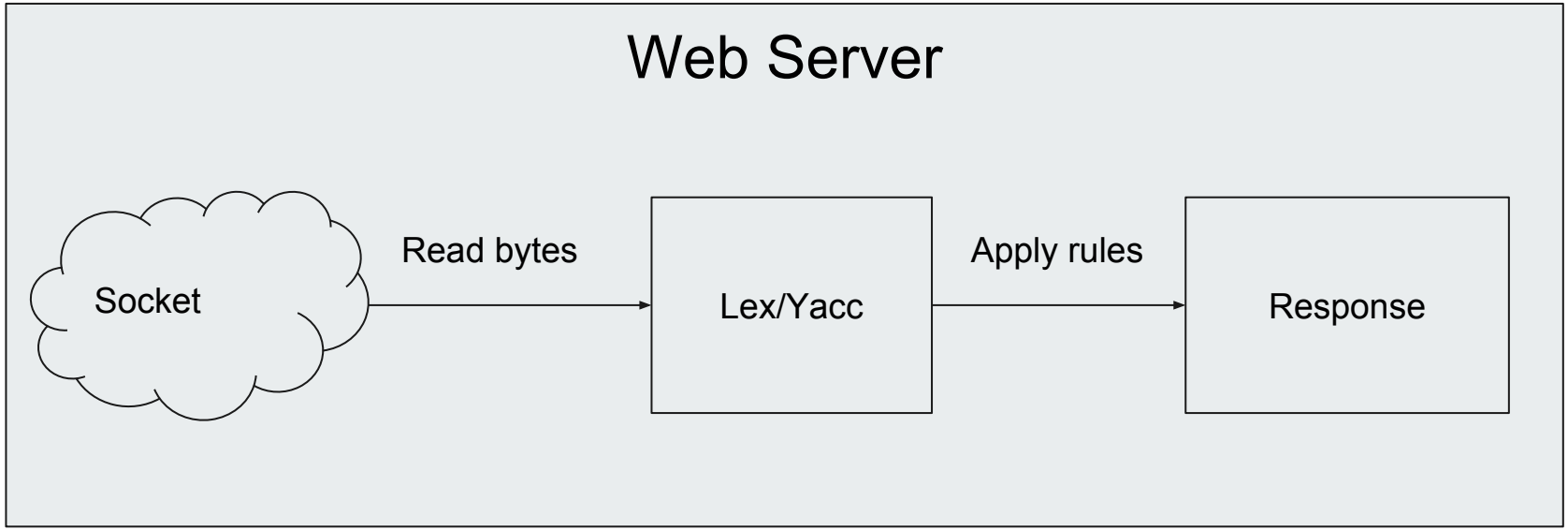
Don't forget to update the tag when you make changes to your code. We run a checkout with the tag name, not your last commit.

Why need HTTP Parsing?

- Socket programming lets you read and write data.
- We need to interpret the data in the form of requests.

```
GET /tutorials/other/top-20-mysql-best-practices/ HTTP/1.1
Host: net.tutsplus.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.1.5)
Gecko/20091102 Firefox/3.5.5 (.NET CLR 3.5.30729)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Keep-Alive: 300
Connection: keep-alive
Cookie: PHPSESSID=r2t5uvjq435r4q7ib3vtdjq120
Pragma: no-cache
Cache-Control: no-cache
```

Solution: Implement rules



Lex/Yacc Overview

... definitions ...

%%

... rules ...

%%

... subroutines ...

- Lex =>Lexical Analyzer
- YACC=>Yet Another Compiler compiler

- Define tokens. (Already done by us)
- Apply rules on tokens.
- You now have HTTP parser !

Starter Code Explanation (Rule 1)

RULE for request line

`GET /some/example/URI/ HTTP/1.1`

Snippet from parser.y:

```
request_line: token t_sp text t_sp text t_crlf {
      YPRINTF("request_Line:\n%s\n%s\n%s\n",$1, $3,$5);
      strcpy(parsing_request->http_method, $1);
      strcpy(parsing_request->http_uri, $3);
      strcpy(parsing_request->http_version, $5);
  }
```

Starter Code Explanation (Rule 2)

RULE for request header

```
Host: cs.cmu.com
```

Snippet from parser.y:

```
request_header: token ows t_colon ows text ows t_crlf {
    YPRINTF("request_Header:\n%s\n%s\n",$1,$5);
    strcpy(parsing_request->headers[parsing_request->header_count].header_name, $1);
    strcpy(parsing_request->headers[parsing_request->header_count].header_value, $5);
    parsing_request->header_count++;
};
```

Design Document

- High-level overview of your code.
- Intuitions, design choices/trade-offs and flow-diagrams.
- This works as your insurance -- if your code our fails test cases.
- Treat this seriously as this the first thing we will look out for.

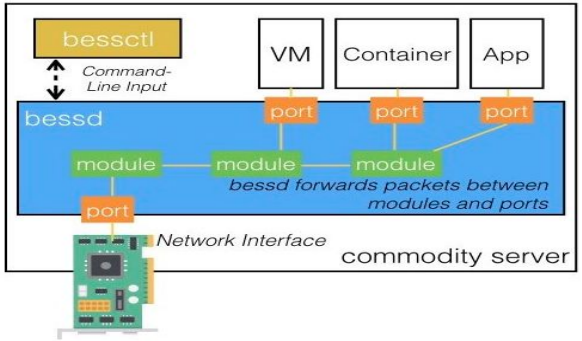
Don’t paste your code and explain

```
// Handler to resize the content container when the browser is resized.
var _resizeTimeOutID;
_window.resize(function () {

    // This technique coalesces the calls to 'adjustContentContainer'
    // so it only fires once the browser is truly done resizing and
    // not fired multiple times during resizing.
    window.clearTimeout(_resizeTimeOutID);
    _resizeTimeOutID = window.setTimeout(adjustContentContainerFromResizeEventHandler, 10);
});

// Only valid in browsers that implement the HTML5 navigation
// pushState and accompanied APIs. When change to the navigation
// state is recognized then the page attempts to load the requested page.
_window.on('popstate', function (event) {

    this._popStateEventCount++;

    // Only process if the event fires as the result of the
    // back or forward button being clicked - skip if
    // the event is being raised on the page load.

    // This skips processing for WebKit browsers the first
    // time the event is raised.
    if ($.browser.webkit && this._popStateEventCount == 1) {
        return;
    }
```

Some good examples

- https://github.com/NetSys/bess/wiki/BESS-Overview

To get started with BESS, you should know about four key components of BESS:



- **bessd**: the "BESS daemon" is the core software switch. The daemon itself carries packets between ports and modules.
- **ports**: ports are places that packets may enter or exit bessd. A port can connect to a network interface, to a virtual machine, to a containerized app, or a normal process running in user space.
- **modules**: modules are chunks of code that allow bessd to inspect or modify packets.

Some good examples

- http://www.aosabook.org/en/bash.html

The basic data structure the shell uses to pass information from one stage to the next, and to operate on data units within each processing stage, is the `WORD_DESC`:

```
typedef struct word_desc {
  char *word;           /* Zero terminated string. */
  int flags;            /* Flags associated with this word. */
} WORD_DESC;
```

Words are combined into, for example, argument lists, using simple linked lists:

```
typedef struct word_list {
  struct word_list *next;
  WORD_DESC *word;
} WORD_LIST;
```

`WORD_LIST` s are pervasive throughout the shell. A simple command is a word list, the result of expansion is a word list, and the built-in commands each take a word list of

3.3. Input Processing

The first stage of the bash processing pipeline is input processing: taking characters from the terminal or a file, breaking them into lines, and passing the lines to the shell pa
commands. As you would expect, the lines are sequences of characters terminated by newlines.

3.3.1. Readline and Command Line Editing

Bash reads input from the terminal when interactive, and from the script file specified as an argument otherwise. When interactive, bash allows the user to edit command lin
familiar key sequences and editing commands similar to the Unix emacs and vi editors.

Bash uses the readline library to implement command line editing. This provides a set of functions allowing users to edit command lines, functions to save command lines a

Questions ?

Good Luck and Start Early!