Internet Exam

Sam Auciello

23 April 2013

Question 1:

(a) Assume that a computer in the Marlboro computer lab has just been initialized. It has its network configured, but has otherwise not been used. Explain what packets and protocols would be exchanged were someone try to visit (a) cs.marlboro.edu or (b) mit.edu with a browser. Be explicit, and use this scenario to explain as many of the fundamental layers and services as you can.

(b) Illustrate your answer to the first part with actual packet captures from wireshark or a similar tool in an analogous situation.

cs.marlboro.edu:

1. The lab computer does a DNS (domain name system) query to one of the local campus DNS servers at 10.1.2.2, 10.1.2.17, or 10.2.0.2. This query uses ethernet protocol to communicate with the router in the lab and the IP (internet protocol) layer to direct the packet to whichever DNS server is being used on port 53. It then uses UDP (user datagram protocol) as the transport layer to send request for the IP address associated with cs.marlboro.edu.

2. Noting that the request came from within campus network, the DNS server responds with a CS's local IP address: 10.1.2.19

3. The lab computer next sends an HTTP (hyper-text transfer protocol) request, again using ethernet to communicate with a router and IP to direct the packet to CS (using the recently discoverred IP address) on port 80. This time it uses TCP (transmission control protocol) to

establish and maintain connection where both parties keep track of what has been sent and recieved so that any data that gets lost or arrives out of sequence can be re-requested. At the application layer, an HTTP request like the following is made:

```
GET / HTTP/1.1
Host: cs.marlboro.edu
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_3) AppleWebKit/536.2
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding gzip, deflate
DNT: 1
Connection: keep-alive
```

4. CS sends an HTTP response like the following:

```
Connection: Keep-Alive
Content-Type: text/html; charset=ISO-8859-1
Date: Mon, 22 Apr 2013 17:04:12 GMT
Keep-Alive: timeout=15, max=50
Server: Apache
Transfer-Encoding: chunked
```

Followed by an HTML (hyper-text markup language) document describing the page. In this case the document contains references to three other resources:

- `/home/home$_{\mathrm{style}}$.css`
- `/images/mc-logo.gif`
- `/images/cs$_m$$_{\mathrm{edu}}$.png`

mit.edu

1. The lab computer does a DNS query to one of the local campus DNS servers at 10.1.2.2, 10.1.2.17, or 10.2.0.2. This query uses ethernet protocol to communicate with the router in the lab and the IP layer to direct the packet to whichever DNS server is being used on port 53. It then uses UDP as the transport layer to send request for the IP address associated with mit.edu.

2. The DNS server responds with its cached mit.edu IP address: 18.9.22.69

3. The lab computer next sends an HTTP request, using IP to direct the packet to mit.edu (using the recently discoverred IP address) on port 80. The packet's path is much more complicated this time as it leaves the campus and goes as far as New York before getting to Cambridge. It uses TCP to establish and maintain connection where both parties keep track of what has been sent and recieved so that any data that gets lost or arrives out of sequence can be re-requested.

4. mit.edu sends an HTTP response not unlike that of CS followed by an HTML (hyper-text markup language) document describing the page. In this case the document contains references to 22 other resources. 15 of these resources are also at mit.edu and can be retrieved using the same connection. 3 of them are at web.mit.edu which is an alias for WWW.MIT.EDU.EDGEKEY.NET which is in turn an alias for e7086.b.akamaiedge.NET which has the IP address 23.6.134.151. A new TCP connection is started to retrieve them. The remaining four resources from ajax.googleapis.com (an alias for googleapis.l.google.com who's IP address is 173.194.73.95), google-analytics.com (which has five ip addresses: 74.125.226.209, 74.125.226.210, 74.125.226.211, 74.125.226.212, and 74.125.226.208), ssl.gstatic.com (who's ip address is 74.125.226.239), and dnn506yrbagrg.cloudfront.net (who's ip address is 54.240.190.146). It's worth noting that the the resource from ajax.googleapis.com is jquery and whenever this computer visits any web page in the future that links to the jquery hosted at ajax.googleapis.com, the lab computer will be able to refer to the cached copy it gets from this request and avoid needing to repeat the request.

wireshark:

My packet captures are included in the packets directory. They were made using the filter `dst host 10.1.2.19 or src host`10.1.2.19| for cs.marlboro.edu and `dst host 18.9.22.69 or src`host 18.9.22.69| for mit.edu.

Question 2:

```
Design a SQL database schema for an online go server,
including games, players, obvservers, and whatever else
seems appropriate. Use this situation to explain the notions
of database normalization, many-to-many, many-to-one,
and one-to-one relations.
```

```
Construct an explicity schema for your design in a SQL system of your
choice, populate it with some sample data, and illustrate how it could
be accessed with a few representative typical interactions.
```

My Schema for the server I constructed for the following example was fairly simple. It consists of two tables connected by two many-to-one relations. Rails automatically generates a schema file `db/schema.rb`. Mine follows:

```
ActiveRecord::Schema.define(:version => 20130417185645) do

  create_table "games", :force => true do |t|
    t.text     "moves"
    t.datetime "created_at",      :null => false
    t.datetime "updated_at",      :null => false
    t.integer  "white_player_id"
    t.integer  "black_player_id"
  end

  create_table "users", :force => true do |t|
    t.string   "username"
    t.datetime "created_at", :null => false
    t.datetime "updated_at", :null => false
  end

end
```

The `white$_{\mathrm{player}}$$_{\mathrm{id}}$` and `black$_{\mathrm{player}}$$_{\mathrm` fields in the `games` table are foreign keys to the `users` table forming two many-to-one relations which in this case means that a given game has only one white player and one black player while a given user can be the black or white player in many games. This information could have been equivalently kept with a schema like the following:

```
ActiveRecord::Schema.define(:version => 20130417185645) do

  create_table "games", :force => true do |t|
    t.text     "moves"
    t.datetime "created_at",      :null => false
```

```
    t.datetime "updated_at",      :null => false
    t.string   "white_player_username"
    t.datetime "white_player_created_at", :null => false
    t.datetime "white_player_updated_at", :null => false
    t.string   "black_player_username"
    t.datetime "black_player_created_at", :null => false
    t.datetime "black_player_updated_at", :null => false
  end

end
```

The reason for an additional table is what is called database normalization. Normalization is process of organizing a database schema so that no information is repeated in multiple places. In the above schema information about individual users would be repeated for each game that user played. The advantage of not repeating this information is two-fold. First it makes the data storage more efficient in the long run. Second, if anything should need to change (even if no reason to change it can be foreseen) only needing to change it in one place is a huge boon.

It would also have been possible implement observers in my database by having a many-to-many relation between these two tables using an additional table like the following:

```
create_table "observations", :force => true do |t|
  t.datetime "created_at",      :null => false
  t.datetime "updated_at",      :null => false
  t.integer  "user_id"
  t.integer  "game_id"
end
```

This would be a many-to-many relation because a given user could observe many games and a given game could be observed by many users. Many-to-many relations always involve a separate lookup table like this.

One-to-one relations are generally less common. They are implemented the same way as many-to-one relations except that a given record from the table without the foreign key is only meant to be associated with a single record from the table with the foreign key. This can be enforced in most database systems by making the foreign key field unique. This is fairly rare

because usually it is simpler (and therefore preferable) to add all of the fields from one of the tables in the one-to-one relation to the other.

The rails interactive console provides a programatic interface to the database. The following creates a pair of players and a pair of games:

```
> sam = User.new :username => "Sam"
=> #<User id: nil, username: "Sam", created_at: nil, updated_at: nil>
> jim = User.new :username => "Jim"
=> #<User id: nil, username: "Jim", created_at: nil, updated_at: nil>
> sam.save
=> true
> jim.save
=> true
> g = Game.new :white_player => sam, :black_player => jim
=> #<Game id: nil, moves: nil, created_at: nil, updated_at: nil,
     white_player_id: 1, black_player_id: 2>
> g.save
=> true
> g = Game.new :white_player => jim, :black_player => sam
=> #<Game id: nil, moves: nil, created_at: nil, updated_at: nil,
     white_player_id: 2, black_player_id: 1>
> g.save
=> true
```

This is equivalent to the sql:

```
INSERT INTO users (username) VALUES ("Sam"), ("Jim");
INSERT INTO games (white_player_id, black_player_id) VALUES (1, 2), (2, 1);
```

The data can later be accessed with something like:

```
> sam = User.find_by_username "Sam"
=> #<User id: 1, username: "Sam", created_at: "2013-04-19 04:43:39",
     updated_at: "2013-04-19 04:43:39">
> g = sam.games
=> [#<Game id: 13, moves: nil, created_at: "2013-04-19 04:44:22",
      updated_at: "2013-04-19 04:44:22", white_player_id: 1,
      black_player_id: 2>,
 #<Game id: 14, moves: nil, created_at: "2013-04-19 04:44:37",
   updated_at: "2013-04-19 04:44:37", white_player_id: 2,
```

```
   black_player_id: 1>]
> jim = sam.games[0].black_player
=> #<User id: 2, username: "Jim", created_at: "2013-04-19 04:43:42",
     updated_at: "2013-04-19 04:43:42">
```

This is equivalent to the sql:

```
sqlite> SELECT * FROM users WHERE username = "Sam";
1|Sam|2013-04-19 04:43:39.489586|2013-04-19 04:43:39.489586

sqlite> SELECT * FROM users, games WHERE
        users.username = "Sam" AND
        (users.id = games.white_player_id OR
         users.id = games.black_player_id);
1|Sam|2013-04-19 04:43:39.489586|2013-04-19 04:43:39.489586|13||
  2013-04-19 04:44:22.647385|2013-04-19 04:44:22.647385|1|2
1|Sam|2013-04-19 04:43:39.489586|2013-04-19 04:43:39.489586|14||
  2013-04-19 04:44:37.718101|2013-04-19 04:44:37.718101|2|1

sqlite> SELECT * FROM users WHERE id = 2;
2|Jim|2013-04-19 04:43:42.240019|2013-04-19 04:43:42.240019
```

Question 3:

> Implement a prototype at least part of the go server from the previous question with a technology stack of your choice. You don't need to do the go game and graphics itself (though you can if you want); instead, consider mainly the fundamental data model, views, controller, and user interactions of the web pages and the database that you'd expect for any standard web app.
>
> Explain your choices of technology, and use this example to demonstrate your familiarity with current options.
>
> Deploy the app in an environment of your choice. (You may use this as a chance to discuss web servers and their configuration, but that isn't the main point.)

I've implemented a go server using Ruby on Rails. I chose the framework mostly because I will be using it for work a lot soon and it seemed like a

good excuse to refamiliarize myself with it. I also find Rails to be an ideal choice for rapid prototyping for two reasons. Firstly, it was designed to fascilitate rapid developement with plenty of helpful command line scripts that automatically generate code and a fully-featured MVC system that makes the most common things as easy as possible. Secondly, the ammount of mementum behind the rails community ensures that things are supported and work most of the time meaning that prototyping is less likely to devolve into debugging or working around an issue with the framework or re-inventing the wheel where features are missing. Rails has a philosophy of convention before configuration. The means that many of the APIs in the framework assume a certain default behavior (like that the bar method of the FooController class will be mapped to the template found at app/views/foo/bar.html.erb). There are generally ways to override these default behaviors but you still need to know the conventions before you can do anything. This can make learning Rails a bit daunting but it really pays off with rapid prototyping. I also considdered using a Node.js framework like Tower.js as that would give me an excuse to use Hot Cocoa Lisp but that would have required me to learn a new framework and seemed outside the scope of the exam. I did get a chance to write a short piece of client side code in Hot Cocoa Lisp.

I've deployed the app using heroku. It can be found at http://go-server.herokuapp.com. I chose Heroku because for a simple demo like this where I don't really need much bandwidth or processing power anyway, their free option is quite reasonable and deployment is as easy as setting up and pushing a git repository to the right place. I ran into a slight issue because I was using sqlite3 in development Heroku doesn't support sqlite and I had to set it up to us pgdb in production.

The app itself is incredible simple. Users "log in" by supplying a name for themself. If the name corresponds to a user in the database then they are authenticated as that user (so pretending to be somebody else is pretty easy). If the user doesn't exist yet then a new one is created. This means that if you forget or misspell your username you might have issues using the app. Anyone can create a game at which point they are assigned a random color. If a game is missing one or more player it will be listed as open and any player will be able to join as the missing color. Players can make moves when it is their turn by clicking on the intersections of the board or clicking 'pass'. There is nothing to stop a player from going on an intersection that has already been played on (the stone will simply be replaced with one of this player's color). There is no logic for removing captured stones and there

is no score estimation or even a concept of winning. A game is considdered finished when both players have passed successively.

Question 4:

```
Using the go server from the previous question, discuss the possible
security issues in such an installation, including at least user
authentication, cookies, sql injection, cross-site scripting, denial
of service attacks, and whatever else strikes your fancy. Be as
specific as you can.
```

The simplest class security issues withmy go server are intentional ommisions. I didn't bother to implement user authentication or the rules of go so anyone could trivially impersonate another player, place a stone on a non-empty square, or violate the Ko rule. Slightly less obvious issues could crop up if, for example I failed to create a server-side check that it was the current user's turn before making a move, or that a game had an opening before somebody tried to join it. Without these checks someone could trivially spoof the relevant HTTP request to make two moves in a row or supplant one of the players in a game they weren't involved in.

Another important class of security issues is called cross-site scripting. This is when a user submits some form of content that intended to be automatically displayed as HTML and includes some JavaScript. Since this JavaScript will be executed by anyone visiting the page, it can easily access their cookies allowing the attacker to impersonate them. Effectively elliminating this sort of attack requires that only a limitted set of html tags and properties be allow in user submitted content to make sure that JavaScript cannot be included. My Installation dodges this but not having user submitted content. A related but importantly different security issue is cross-site request forgery. This is when a user that is authenticated with one website visits a completely different site the second site causes their browser to make a request that requires the users authentication on the first site. For example, if Alice is logged into her bank account looks at a conversation on an open forum, it is possible for Eve to have placed an image tag on the forum page that instead of pointing to an image, points the URL of a request on Alice's bank website. The bank website could protect Alice from this sort of attack using a CSRF token. The basic strategy is to generate a random number with any request for a web form and put that number in a hidden input tag in the form. Then when the form is submitted the server can ignore

the request and give an error message if the token submitted doesn't match the one given out in the first place. Ruby on Rails implements this by default for all forms. This means you can feel secure in knowing that visitting websites besides go-server.herokuapp.com won't allow malicious websites to make moves for you in your go games.

Another very important class of security issues is SQL injection. This is when a user submits data via a form which is intended to be stored in a database but the user submits SQL code instead which allows them to gain control of your database. The archetypal example is entering something like `';DROP TABLE users;--` enterred into a for that is being processed on the server by something like:

```
sql = "SELECT * FROM users WHERE name = '" + user_submitted_data + "';"
result = db_query(sql);
```

The sql request becomes `SELECT * FROM users WHERE name = ‘‘;`DROP TABLE users;– ‘;| and all data in the users table is lost. The approriate solution to this is to sanitize user inputs so that things like quotes and semicolons are escaped and the user submitted text is treated as it was intended to be. An even better approach in most practical is to use a well tested framework or library for all database access that does this sanitization for you. This saves from the danger of forgetting to sanitize inputs in some places. ActiveRecord, the ORM (object relational mapping) system that rails uses to interface to databases does just this.

Denial of service attacks are a bit different in that they don't usually do any sort of long term damage. Rather, they make a website unavailable or less available by bombarding the server with requests. A traditional DOS attack where you simply repeatedly send requests to a server from a sigle machine are relatively easy to prevent simply by having systems that detect unusually high volumes of traffic from a single source and reject requests all requests from those sources. A distributed denial of service attack (or DDOS) is such an attack is made in a way that makes the traffic appear to (or actually) come from different sources. There are several complex ways of doing this and basically no consistent ways of preventing it. I suspect (though I have not looked into it) that Heroku provides some basic protection from traditional DOS attacks. I am in some sence protected from DDOS attacks by the mere fact that nobody would go to the trouble setting up a DDOS for a simple Go server rails demo. I might be vulnerable to attack on other

sites hosted by Heroku but I'm not particularly worried as DDOSes tend to target high profile sites with some sort of political purpose.

Question 5:

> `Finally, discuss your personal opinions and preferences on old and new technologies and trends in web development, including perhaps the core technologies of HTML, XHTML, and HTML5 (whatever you think that means), CSS and its compilers, Javascript and its compilers and popular libraries, the backend languages and frameworks like PHP, Rails etc, popular systems such as Wordpress, as well as options like Node, Backbone, Bootstrap and so on. The specific list is up to you; the point here is to show that you have an overview and understanding of where the field was and is going.`

Anyone can tell you that internet has changed quite rapidly in recent decades. I think the most significant change has also been the most obvious one: more people use it. When Tim Berners-Lee invented the world wide web a seriously doubt he forsaw FaceBook. I that the rise of social networks has had less to do with the progression of underlying technology then with the change in needs of the web's users. This increase in internet use has (and will likely continue to have) positive and negative effects on the field. Specifically, the increased economic demand for new web applications has fueled many innovative projects. Rails for example has grown into a powerful tool that allows developers to quickly get from an idea to a working prototype. Meanwhile, the demand for higher performance servers has spawned Node.js which powerfully shakes the way people think about concurrency.

The increase in internet use also means that multiple significant demographics that don't have a good understanding of computers now have a stake in the direction of the internet. These include lay-people using social networks, business people who want to profit from the internet's success, and amateur web developers who understand the technologies well enough to use them but never bother to go deeper. Trends like the continued popularity of PHP (and it's use in popular applications like FaceBook, WordPress, and Wikipedia) worry me. People who don't know much about computers can easily look PHP, see how successful it is, and conclude it is a good thing.

Of course these forces aren't totally new either. There have always been forces keeping things from turning out perfectly. The history of the inter-

net is full of stories about a technology not doing everything we wanted it to and people finding inellegant work-arounds that later became industry standards. The document object model and libraries like JQuery is a really good example. The DOM didn't have the features that were needed so we built around it. I see this as making lemonade when life gives you lemons. Languages that compile to JavaScript and CSS are like lemonade.

The internet is a complicated thing with many interested parties and many interlocking moving parts. I think that's why I find it so fascinating. I hope whatever contributions I am able to make to it's future in my lifetime help people make sense of and do cool things with it.