Internet Connectivity

1 Introduction

With the explosion of computing power and connectivity, the Internet has become a primary means of storing, processing, and delivering data. Cloud computing, social networks, search engines, the "Internet of Things" – each of these require sharing and storage of information, some of it in large quantities (so-called "big data"). This document lays out the fundamentals of using a remote server to store and display information.

The flow of data will happen in several steps, and tutorials and sample code will be provided to help you understand each piece of the puzzle. You'll start with simple, "Hello, world!" applications and work up to a website that displays key statistics of your system (eg, temperatures) and allows you to *control* your greenhouses from the Internet.

Objectives

Upon successful completion of this lab, the student will be able to:

- Create a database and multiple, linked tables within it,
- Implement scripts for retrieving data from a database,
- Implement scripts for storing data in a database,
- Transfer data (bi-directionally) between an Arduino and a web server, and
- Implement a simple web interface for setting control parameters.

Preparation

Before coming to lab, the student should review the following:

- **Basic LAMP model.** See the overview below as well as Mark Rossi's notes on collab.
- **SQL and PHP.** There are links to resources below as well as a good primer by Mark Rossi on collab.
- **Visit the entire Internet.**

Be sure to read through the entire lab before coming to lab.

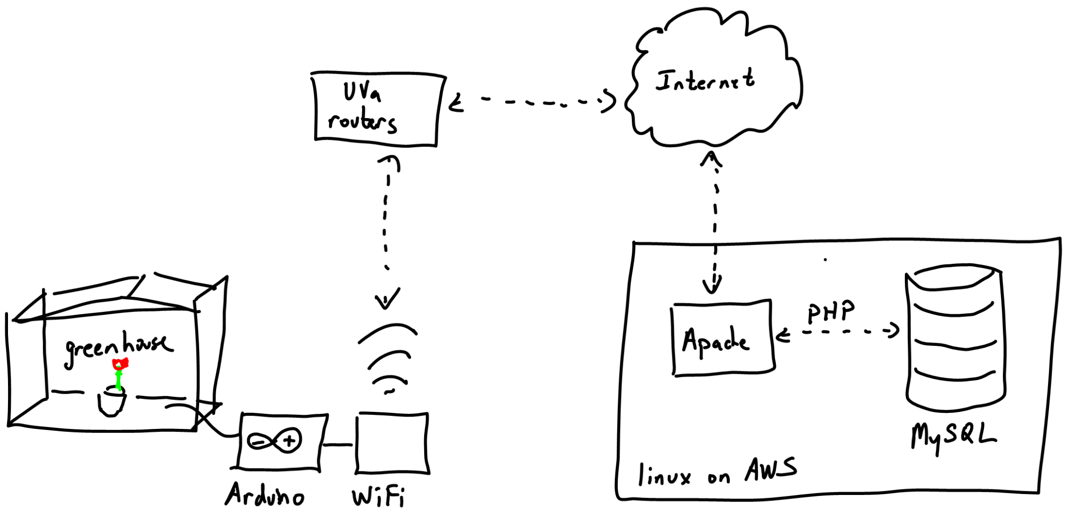

Figure 1: High-level steps needed to send data to a server. See text for description of each step.

2 Overview of information transfer

The goal over the next few weeks is to connect your greenhouse to the Internet. Doing so will allow you to both monitor and control the greenhouse remotely, key tasks in creating systems in the "Internet of Things". For this project, the database will live on a virtual machine at Amazon Web Services, and several steps must be accomplished to both store and retrieve information, both from your Arduino and a web browser. Your system will have to connect to the Internet, contact the web server, and send commands to store or retrieve information. You will have to write programs on the server to process incoming requests and build a database to hold all of the data. You'll need to make a web page to change parameters and visualize information. Each of these steps has its own complications – there is a lot to do!

Figures 1 shows a high level representation of how your system interacts with a database. The steps that are taken include:

1. The microprocessor (Arduino) that controls your system decides that it needs to send information to or retrieve information from the database,
2. The microprocessor opens a connection to the web server via a WiFi chip and sends a formatted request, which could be either a command to store sensor readings or a request parameters,
3. The server receives the request and invokes the appropriate scripts, which you will write,
4. Those scripts access a database to store/retrieve information,
5. The same scripts send responses back to the web server, which relays the results to your system, and
6. The microcontroller processes data received through the WiFi chip.

The LAMP model

The overview above only lists the high-level steps needed to connect your system to the Internet – the details for each step are much more complicated and the focus of the rest of this document. On the server side, however, much of the process has been taken care of for you by programs designed specifically to handle these tasks. Specifically, the server is running something called a *LAMP stack*. LAMP is an acronym that stands for Linux, Apache, MySQL, PHP/Perl/Python. **Linux** is the operating system that runs on the server.[1] **Apache** is the program that serves web pages. Basically, it listens for requests on *ports* on the server (eg, `http` requests on port 80) and responds by generating web pages, which may require gathering necessary information for those pages. **MySQL** is a popular database that uses **S**tructured **Q**uery **L**anguage to insert and extract data from the database. **PHP**, **Perl**, and **Python** are all scripting languages that are used to interface with databases (among other things). They are all object-oriented (like Java or C++), but differ in their implementations. This handout focuses on PHP, which is the most common language for scripting web pages, though many of the cool kids are using Python these days. You may use python if you wish, but the examples herein are PHP and your instructor might not be of much help.

Each of the components in the LAMP stack are courses unto themselves. In particular, there are database classes in both Systems Engineering and CS that will give you a deeper understanding of databases and SQL, and there are programming classes where you can learn more about PHP and Python. If you want to get deeper into linux, I'm not going to stop you.

A good reference for SQL can be found at w3schools.com. It is unrealistic for you to understand everything on those pages, but you will be responsible for the basics (syntax and the different kinds of statements you can use). **For each of the SQL statements in this lab, be sure to look up the syntax of the statement to see how it works.**

A decent, beginner's tutorial for PHP can be found here, and a quick web search will find many others. There is also a short primer on HTML + PHP for this project written by former student Mark Rossi ('15), found on collab. Again, learning PHP is a course unto itself, but digging into his tutorial to see how things work will help you tremendously.

You will be provided with an account on a virtual linux machine on Amazon Web Services, where you will create a database and PHP scripts. Credentials will be provided separately. As noted, the server is running linux, which has implications for how you can access it:

- If you're running linux or OSX, you can connect using `ssh`, which stand for "secure shell". If you're running linux, you presumably know how to do this. If you're running OSX, you may not be familiar with `ssh`, but help will be provided in class.
- If you're running Windows 10, my understanding is that it finally has a native `ssh` client – welcome to the 21st century. It'll be up to you to figure out how to run it (it's hidden by default, according to my sources).
- If you're running an older version of Windows, or simply want to use a tried and true client, you'll can connect using `putty` or any program that will allow you to login to the server. In the past, students have tried to edit local copies and copy them to the server when needed. Debugging with such a setup is painful. Trust me.

[1]Analogous systems that run Windows are frequently termed WAMP. Mac OS, MAMP.

3 Getting started on the cloud

Your instructor has set up a virtual LAMP stack on AWS. It's 'virtual' because it is not simply a single computer sitting in a closet. AWS runs an *instance* of a machine on whatever resources it has available. A single machine, or set of machines, at AWS can run multiple instances, all of which appear to be a single machine to us.

Since this is the first year this class has used AWS, there will likely be some unforeseen issues that arise (in particular, access to AWS machines defaults to "very limited", so your instructor has to specifically open things up for you). As always, your patience is appreciated.

Your account

Your instructor has what is known as *root privileges* on the server, which means he can create and delete accounts, install useful software like MySQL, adjust security settings, and so forth. With this awesome power, he has created an account for you and set you up with access to MySQL. To log in, you will need an `ssh` client, as noted above.

Logging into your account is not done with a password, but with a public/private key pair. The private key will live on your computer and allow you to login to the machine without having to enter a password. This may sound less secure than a password, but it's not only more convenient – it's also more secure, **so long as the private key remains private!** If someone gets hold of your private key, she or he would have unlimited access to your account, which would be detrimental to your system's performance, to say the least, and likely have cascading effects on your grade.

Your instructor will provide you with a private key, which you will store on your computer.

Procedure

1. Assuming your key is in a file called `fyn.pem`, change to the directory where `fyn.pem` lives, and enter:

   ```
   ssh -i "fyn.pem" <comp_id>@ec2-34-209-142-24.us-west-2.compute.amazonaws.com
   ```

   where `comp_id` is your UVa computing id. You should be immediately logged in to the server, where, you can issue a number of linux commands, including:

   - `ls` will list the files in the current directory. `ls -al` will list all the files (including hidden files) and their permissions.
   - `cd <dir>` changes the current directory to `<dir>`. You can used `cd ..` to go up one directory.
   - `pwd` outputs your present working directory.

   When you type `ls`, you should see a directory called `public_html`. This directory has been especially set up for serving your particular web files. Any file you put in `public_html` can be found by the `apache` web server and will therefore be publicly available.

2. To demonstrate, `cd` to the `public_html` directory and type:

```
emacs test.txt
```

This will open the `emacs` text editor, which is both very powerful and rather archaic. Commands in `emacs` are executed using key combinations, some of which make sense and some of which don't. If you prefer other text editors (`vi`, `nano`, etc.), feel free to use something else.

3. In your new file, type "hello".
4. Type `<ctrl-X> <ctrl-S>` to save and then `<ctrl-X> <ctrl-C>` to close the file.
5. Point your favorite browser to:

   `ec2-34-209-142-24.us-west-2.compute.amazonaws.com/~<comp_id>/test.txt`.

   If you see "hello" printed in your browser, congratulations are in order: you're on the cloud!

PHP scripts

PHP is a popular language for web scripting, or scripting in general. PHP scripts will provide the connection between the Apache web server and your database. The scripts will be invoked by Apache when a request is made from the Internet, whether from a browser or the Arduino that controls your greenhouse. The scripts will parse the request and interact with your database using SQL commands. Depending on the SQL statements, the script may add information to the database or retrieve data to be displayed. Others may also delete or update records – it's all a matter of what you tell the script to do. Before you start to interact with the database, however, you'll be introduced to the basics of PHP and verify that your account is set up properly.

There are two basic types of PHP scripts: `GET` scripts and `POST` scripts. The fundamental difference is that a `GET` scripts receives information appended to the HTTP request itself, whereas a `POST` script receives information packaged in a separate container. `POST` scripts are more useful when you send lots of data or don't want to have sensitive information such as passwords printed in clear text in a URL. That said, `GET` scripts are better for learning how the scripts work (and debugging!), so we'll concentrate on them.

The basic format of a `GET` request is as follows:

```
my_script.php?<field1>=<value1>& ... &<fieldN>=<valueN>
```

where `my_script.php` is the filename of the script and the `?` tells it that a list of `field=value` pairs follows. Each of these pairs will be parsed within the script itself, which allows the user to send search parameters or data to the script.

Procedure

1. Login to your account on the virtual AWS machine. Switch to your `public_html` directory.
2. Using your favorite editor, open a file called `repeater.php` and add the following to it:

   ```
   <?php
   $input_str = $_GET['input'];
   echo $input_str;
   ?>
   ```

3. Save the file and exit the editor.
4. Switch to your favorite browser and enter the following URL (all on one line):

   ```
   ec2-34-209-142-24.us-west-2.compute.amazonaws.com/
             ~<comp_id>/repeater.php?input=this is awesome
   ```

   where `<comp_id>` is your computing ID, of course. Your browser should print "this is awesome" or whatever else you type after the `=` sign.

A few notes:

- PHP scripts are denoted by the header and footer in the first and last lines of the code above. All PHP code must go between such delimiters, and it is possible to embed PHP within other scripts (e.g., HTML pages).
- Variables in PHP are denoted by the `$` sign. Note that you don't need to declare a variable type – PHP figures out what type of variable to use automatically (which is both a blessing and a curse...).
- You retrieve the parameters using the format: `$_GET['field_name']`, where `field_name` is the name of the field typed into the URL. In this case, when you entered `input=<some text>`, the script will grab `some_text` and assign it to `$input_str`, which is the variable within the PHP script. Note that `POST` calls work essentially the same, except that they use `$..._POST...$`.
- A variable can be written to a web page using the `echo` command, which outputs the current value to a stream, which in this case is picked up by the Apache web server for outputting to your browser.
- Finally, when calling your script from a web browser, you simply prepend the script name with the server name and its location. Apache, running on the server, will then be able to find your script and invoke it. In this case, Apache automatically translates `~<comp_id>` into your `public_html` directory.

4 Creating a database

A PHP script that simply repeats information isn't terribly useful, but if you want to interact with a database, you'll first need to create one. You've been given access to MySQL on the server, and full access to a database that shares a name with your computing id (it can be a little confusing that your login id and database name are the same, but MySQL makes it easy to set things up this way).

Procedure

1. Log in to the AWS virtual machine as you normally do.
2. Enter the database environment by typing: `mysql -u <comp_id> -p`. When prompted for your password, type `tlp_rules`. You may change this if you wish, depending on how well you trust your classmates.

3. The first thing you will need to do is create a database and make it the active one. The only database you have access to will be one with your computing ID, and you will have full permissions on it. At the `mysql` prompt, create a database by typing: `CREATE DATABASE <comp_id>;` Note that the semicolon is necessary, but if you forget it, you can just type it on the next line. Also note that while capitalization does not actually matter, traditional SQL capitalizes keywords – you may do as you want, but all of the examples here will follow tradition.

4. Type `SHOW DATABASES;`. You will see both your database and another called "information_schema", which is a database that contains meta-information about your account.

5. Before you can access a database, you first have to tell MySQL which one you will be using. To do that, type: `use <comp_id>;`

6. You can now list the tables in the database by typing `SHOW TABLES;` Of course, this isn't particularly interesting, as you have not yet made any.

Creating tables

A table is the fundamental container for data storage in a database. In MySQL, tables consist of a set of fields, called *columns*, each of which is defined to be a specific data type (e.g., `INT`, `VARCHAR`, a type of string). Records are stored as *rows*, analogous to the rows in a spreadsheet.

There are special attributes that can be assigned to each column to make the database run more efficiently:

- `PRIMARY KEY`: A primary key is a unique identifier for each record. It helps ensure database integrity, since each record must have a unique primary key. It's generally good practice to always have a primary key, even if it's just an `id` field.
- `INDEX`: When a column is indexed, references to the index are stored in a manner that makes the column much easier to search. For example, if you wanted to find all the records that corresponded to a specific sensor, then indexing *sensor_id* would allow the database to find the correct records without having to go through the records one by one.
- `UNIQUE`: Indicates that each row must have a unique value for the specified field. This could be useful for, say, a UVa computing id or your social security number.
- `NOT NULL`: Indicates that each record must have a value for this field. For a table of sensor readings, it wouldn't make sense to have empty values for the timestamp, for example. On the other hand, a table of addresses might have an optional field for an apartment number, which would be left empty for many records.
- `DEFAULT`: The default value that is assigned to a new a record if the value is not otherwise specified.
- `AUTO INCREMENT`: Similar to default, a way to automatically assign an number that is increased by one with each new record.
- `FOREIGN KEY`: A way to link two tables together to make data storage more efficient. For example, if an inventory database had a table of orders, it wouldn't make sense to include the address of the purchaser in each order record. Instead, you could link each order to a customer table, which would store the relevant information in one record.

The syntax for each of these options can get confusing, to say the least. If you make mistakes, however, it's always possible to see the existing table structure using `DESCRIBE TABLE` and make changes using `ALTER TABLE` commands. You can find a complete description of each command in the MySQL Manual, but that document can be hard to follow so it's often easiest to just use the search engine of your choice to help you find the correct syntax.

Procedure

1. If you haven't already, log in to MySQL and create and select a database with your computing id.
2. Create a table of NCAA basketball teams using the following syntax:

```
mysql> CREATE TABLE teams(
   -> team_id INT NOT NULL AUTO_INCREMENT,
   -> university VARCHAR(40) UNIQUE NOT NULL,
   -> mascot VARCHAR(40) NOT NULL,
   -> conference VARCHAR(10),
   -> tournament_region VARCHAR(10),
   -> tournament_seed INT,
   -> PRIMARY KEY ( team_id )
   -> );
```

Note that you can enter commands on multiple lines – just enter a semicolon when you're finished with the complete command.

This table will hold a list of teams and some information about the post season. Note that the primary key is just an index, and *university* has to be unique. The mascot is mandatory (all teams have a mascot), but the conference can be null (for independents) and so can the tournament data, since not every team gets to the NCAA tournament.

Type `DESCRIBE teams` to verify that all the fields are correct.

Now enter some data:

1. Use the `INSERT` command to add a record:

```
mysql> INSERT INTO teams
    -> (university, mascot, conference, tournament_region, tournament_seed)
    -> VALUES
    -> ('UVa', 'Cavaliers', 'ACC', 'South', 1);
```

2. Add a second record

```
mysql> INSERT INTO teams
    -> (university, mascot, conference)
    -> VALUES
    -> ('ISU', 'Cyclones', 'Big 12');
```

3. And a third record

```
mysql> INSERT INTO teams
    -> (university, mascot, conference, tournament_region, tournament_seed)
    -> VALUES
    -> (‘Kansas’, ‘Jayhawks’, ‘Big 12’, ‘Midwest’, 1);
```

4. Now show the records in the table with:

   ```
   mysql> SELECT * FROM teams;
   ```

   You should see three rows. Note that the tournament columns for ISU are `NULL` since the Cyclones didn’t make it to the Big Dance this year (where’s Fred Hoiberg when you need him?).

5. Now select the teams from just the Big 12 conference by qualifying your search with the `WHERE` clause:

   ```
   mysql> SELECT * FROM teams WHERE conference = ‘Big 12’;
   ```

6. Finally, what do you think will happen if you try entering a fourth record:

   ```
   mysql> INSERT INTO teams
       -> (university, conference, tournament_region, tournament_seed)
       -> VALUES
       -> (‘Duke’, ‘ACC’, ‘Midwest’, 2);
   ```

You may have expected that the record would be rejected since we declared *mascot* to be `NOT NULL` and we didn’t enter a value. It turns out, however, that there is a difference between a `NULL` value and an empty string. `NULL` means there is no value for a field, whereas an empty string is a value – it’s just a zero-length string.

Joining tables

While it was useful for demonstrating how to make tables, the organization of the database from the preceding section is dismal. An easy way to see this is with a simple demonstration.

Procedure

1. Add another record, being careful to enter the conference verbatim:

   ```
   mysql> INSERT INTO teams
       -> (university, mascot, conference)
       -> VALUES
       -> (‘KSU’, ‘Wildcats’, ‘Big12’);
   ```

2. Re-enter (or use the up arrow to scroll though your history) the `SELECT ... WHERE` statement above. What do you see?

The problem is that there is no way of ensuring that the names of the conferences are consistent. You could enter, “Big 12”, “Big12”, “Big Twelve”, or even “Big 0x0C” and the database would blindly accept them all. A fundamental “best practice” of relational databases is to avoid repeated data, and the best way to do that is to divide data up into multiple tables, each of which holds a minimal dataset. Let’s work through an example.

Procedure

1. Create another table, called *conferences*, with the following structure:
   - *conference_id* as the `primary key` with `AUTO_INCREMENT`, and
   - a `NOT NULL` *conference_name*.
2. Use `INSERT INTO` to add entries for the Big 12 and the ACC.
3. Alter the *teams* table as follows:

   ```
   mysql> ALTER TABLE teams DROP conference;
   mysql> ALTER TABLE teams ADD conference_id INT DEFAULT NULL;
   ```

4. Now assign teams to a conference using the `UPDATE` statement:

   ```
   mysql> UPDATE teams SET conference_id = 1 WHERE university = 'ISU';
   ```

   Do this for each entry in the *teams* table.
5. Now enter the following:

   ```
   mysql> SELECT university, conference_name FROM teams
        -> INNER JOIN conferences ON
        -> teams.conference_id=conferences.conference_id;
   ```

You should see a list universities and their conferences. The form of the statement is a little hard to remember, but it's easy enough to do a web search to get the syntax correct. The important point is that we define an `INNER JOIN` to create what looks like a new (but temporary) table with information from multiple sources. *The power of relational databases is that it is possible to combine information from multiple tables to meet the specific needs of the user.*

Foreign keys

Let's break the database once again.

Procedure

1. Add another team:

   ```
   mysql> INSERT INTO teams
       -> (university, mascot, conference)
       -> VALUES
       -> ('Carleton College', 'Golden Knights', 3);
   ```

2. Execute the previous `JOIN` query. Where is Carleton?

The problem now is that there is nothing that ensure that the specific conference id exists in the converences table. What we really want is to ensure that the conference field is correct, and we can do this with something called *foreign keys.* Foreign keys are formal links between tables within a database that can be used to guarantee data integrity. Before you add one though, you'll need to put in a proper entry (and get some more practice!):

Procedure

1. Add a conference record for the MIAC conference. Use a `SELECT` statement to determine its *conference_id*. Make sure it matches the *conference_id* for Carleton College in *teams*.
2. Add a foreign key to the *conference_id* as follows:

```
mysql> ALTER TABLE teams
    -> ADD FOREIGN KEY fk_conf(conference_id)
    -> REFERENCES conferences(conference_id)
    -> ON DELETE CASCADE
    -> ON UPDATE CASCADE;
```

3. Repeat the university/conference query from before. Is Carleton there now?
4. Now try adding a university in an entirely new conference. What happens?

5 Web interface

Now that you've seen how to create and manipulate databases, it's time to write more powerful PHP scripts that can interact dynamically with the data in your database. The basic premise is straightforward: we'll have PHP scripts perform SQL statements and pipe the results to a web page. We'll get started once again with an example.

Procedure

1. Type 'quit' to exit the MySQL environment. On your local machine, grab a copy of `teams.php` from collab and upload it to the server with `scp`, which is short for "secure copy". Type the following all on one line:

```
scp -i "fyn.pem" teams.php <comp_id>
@ec2-34-209-142-24.us-west-2.compute.amazonaws.com:public_html
```

2. On the AWS server, `cd` to the `public_html` directory and open the file in `emacs` or your favorite editor.
3. Change the lines starting with `$db_user` and `$db_pass` to reflect your username and MySQL password. Set `$database` to your computing id (which is also the name of your database). Save the file.
4. Point your favorite web browser to:

```
ec2-34-209-142-24.us-west-2.compute.amazonaws.com/~<comp_id>/teams.php
```

   If all goes well, you should see the result of the query you did in Section 4.

The ability to query the database from a web browser if useful, but the real power lies in being able to query on specific criteria. Using the code from Section 3 as a guide, edit your PHP script to accept *conference_name* as an input. Then alter the SQL statement so that it only returns teams in the specified conference. To so this, you'll need to make use of the `WHERE` clause:

```
"...WHERE conference_name = '$conf_name'...";
```

where I've assumed a variable named `$conf_name`. Note that the entire query must be in double quotes and the variable in single quotes. This tells PHP to interpret the variable as its actual value and then the query literally.

Assignment (5 points)

This is an individual assignment. Due Wednesday, 3/28 at the start of class.

1. **Create a table to hold sensors.** Create a table called "sensors" with four fields:
   - *id* of type `INT`,
   - *name* of type `VARCHAR(32)`,
   - *short_name* of type `VARCHAR(12) UNIQUE`, and
   - *units* of type `VARCHAR(12)`.

   Set *id* as the `PRIMARY KEY`. Make all of the fields `NOT NULL`.

2. **Create a table to hold readings.** Create a table called "sensor_readings". Add the following fields:
   - *id* of type `INT AUTO_INCREMENT`,
   - *sensor_id* of type `INT`,
   - *time_stamp* of type `TIMESTAMP`, defaulted to `CURRENT_TIMESTAMP`, and
   - *value* of type `DECIMAL (5,2)`. Make the *id* field a `PRIMARY KEY`. Make the *sensor_id* field indexed using `KEY 'sensor_sid' ('sensor_id')`. Make all of the fields `NOT NULL`.
   - Create a link (foreign key) from *sensor_readings.sensor_id* to *sensors.id.*

3. **Insert some data.**
   - Insert two sensors:
     (a) Inside temperature: Set *id* to `1` and *short_name* to `T_in`. We roll Celsius in this class.
     (b) Outside temperature: Set *id* to `2` and *short_name* to `T_out`.
   - Insert several readings for each sensor. Note that the *time_stamp* should default to the current time on the server, so you'll need to remove references to it in your `INSERT` statement.

4. **Make it interactive.**
   - Write an SQL statement to SELECT all of the readings of sensor 1. Write the SQL statement here:
   - Following the format for joining tables, write a statement to select all of the readings the sensor with the *short_name* of `T_out`. Write out the result as: *name, timestamp, value, units.* Write the SQL statement here:
   - Write a PHP script that accepts *short_name* as a parameter and returns the results of the previous query. Write out the result as: *name, timestamp, value, units.* Write the URL here: