CIS 505: Software Systems

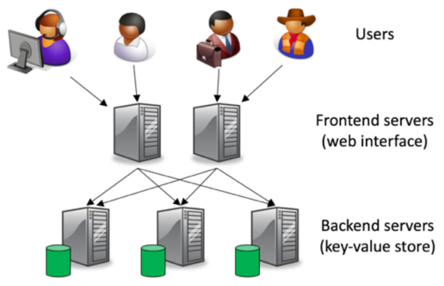
Fall 2016

Final project: PennCloud (Draft)

Teams must form by November 11, 10:00pm EST
Code due on December 12, 10:00pm EST
Demos in the week of December 15-21
Project report due on December 20, 10:00pm EST

1 Overview

The final project is to build a small cloud platform, somewhat similar to Google Apps, but obviously with fewer features. The cloud platform will have a webmail service, analogous to Gmail, as well as a storage service, analogous to Google Drive.



The figure on the right illustrates the high-level structure of the system. Users can connect to a set of *frontend servers* with their browsers and interact with the services using a simple web interface. Each frontend server runs a small web server that contains the logic for the different services; however, it does not keep any local state. Instead, all state is stored in a set of *backend servers* that provide a key-value store abstraction. That way, if one of the frontend servers crashes, users can simply be redirected to a different frontend server, and it is easy to launch additional frontend servers if the system becomes overloaded.

The project should be completed in teams of four. There are several different components that need to interact properly (this is a true "software system"!), so it is critical that you and your teammates think carefully about the overall design, and that you define clear interfaces before you begin. In Section 3, we have included some example questions you may want to discuss with your team. It is also very important that you work together closely, and that you regularly integrate and test your components - if you build the components separately and then try to run everything together two hours before your demo, that is a sure recipe for disaster. To make integration easier, we will provide shared Git repositories for each team. Please *do not* use Github or some other web repository for this project.

In the specification below, we have described a minimal solution and a complete solution for each component. The former represents the minimum functionality you will need to get the project to work; we recommend that you start with this functionality, do some integration testing to make sure that all the components work together, and only then add the remaining features. The latter represents the functionality your team would need to get full credit for the project. Finally, in Section 5, we describe some suggestions for additional features that we would consider to be extra credit. The set of extra-credit features is not fixed, however; you should also feel free to be creative and add functionality of your own.

The project must be implemented entirely in C or C++. You may not use external components (such as a third-party web server or key-value store, external libraries, scripting languages, etc.) unless we explicitly approve them.

2 Major components

2.1 Key-value store

Your system should store all of its user data in a distributed key-value store, somewhat analogous to Google's Bigtable. (We have not covered Bigtable in class yet, but you do not need to implement any of its more advanced features; all you need to know is the interface below.) Conceptually, the storage should appear to applications as a giant table, with many rows and many columns. The storage system should support at least the following four operations:

- PUT(r,c,v): Stores a value v in column c of row r
- GET(r,c): Returns the value stored in column c of row r
- CPUT(r,c,v1,v2): Stores value v2 in column c of row r, but only if the current value is v1
- DELETE(r,c): Deletes the value in column c of row r

The table should be *sparse*, that is, not every row should have to have a value in every column. One way to implement this could be to store the contents of each row as a set of tuples {(c1,v1), (c2,v2), ...}, where the c_i are the columns and the v_i are the values in these columns. The row and column names should be strings, and the values should be (potentially large) binary values; for instance, applications should be able to invoke PUT("linhphan", "file-8262922", X), where X is a PDF file that user "linhphan" has stored in the storage service (see below).

Minimal solution: An initial version of the storage backend could consist of just a single server process that listens for TCP connections, accepts the four operations defined above (you and your teammates can define your own protocol), and stores the data locally. You should be able to reuse some of your HW2MS1 code for this.

Full solution: The complete version should be *distributed*: there should be several storage nodes that each store some part of the table (perhaps a certain range of rows). You may assume that the set of storage nodes is fixed; for instance, there could be a configuration file that contains the IPs and port numbers of all storage nodes, analogous to HW3. It should also *replicate* the data - that is, each value should be stored on more than one storage node - and it should offer some useful level of *consistency* as well as some degree of *fault tolerance* - that is, it should avoid losing data when nodes crash, and the data should continue to be accessible as long as some of the replicas are still alive.

2.2 Frontend server

Your system should also contain at least one web server, so that users can interact with your system using their web browsers. Your web server should implement a simple subset of the HTTP protocol (RFC2616). Below is a simple example of a HTTP session:

```
C: GET /index.html HTTP/1.1<CR>
C: User-Agent: Mozilla<CR>
C: <CR>
S: HTTP/1.1 200 OK<CR>
S: Content-type: text/html<CR>
```

```
S: Content-length: 47<CR>
S: <CR>
S: <html><body><h1>Hello world!</h1></body></html>
```

As you can see, the client issues a request for a particular URL (here: `/index.html`) and potentially provides some extra information in header lines (here: information about the user's browser), followed by an empty line. The server responds with a status code (here: `200 OK`, to indicate that the request worked), potentially some headers of its own, and then the contents of the requested URL.

Your server should internally have several handler functions for different kinds of requests. For instance, one function could produce responses to `GET /` requests, another for `POST /login` requests, and so on. You should take care to avoid duplicating code between the handler functions; for instance, the handlers could each return the response as an array of bytes, and there could then be some common code that sends these bytes back to the client.

Importantly, your server should check whether the client includes a cookie with the request headers; if not, it should create a cookie with a random ID and send it back with the response. This is important so that your server can distinguish requests from different clients that are logged in concurrently. For more information about cookies, please see https://www.nczonline.net/blog/2009/05/05/http-cookies-explained/.

Minimal solution: An initial version of the server could be based on the multithreaded server code you wrote for HW2MS1 (with some adjustments for the different protocol). For a quick introduction to HTTP, see https://www.jmarshall.com/easy/http/. Initially, you may want to just implement GET requests, as in the above example; to get something working, you can leave out anything nonessential, such as transfer encodings, persistent connections, or If-modified-since. You can also initially leave out the cookie handling; however, keep in mind that without this, only one user will be able to use the system at a time.

Full solution: For a fully functional server, you'll need some additional features, including support for POST requests (for submitting web forms and uploading files to the storage service), as well as HEAD requests and cookie handling.

2.3 User accounts

Your system should support multiple user accounts. When the user first connects to the frontend server (a `GET /` request), the server should respond with a simple web page that contains input fields for a username and password. Once the form is submitted, the server should check the storage system to see if the password is correct, and if so, respond with a little menu that contains links to the user's inbox and file folders (and perhaps to extra-credit features, if your systems supports any). If the password is not correct, the server should respond with an error message.

Minimal solution: To get something to work quickly, you could simply preload a few usernames and passwords into the key-value store and check these against the credentials that the user enters.

Full solution: The complete solution should also allow users to sign up for a new account, and users should be able to change their passwords.

2.4 Webmail service

Your system should enable users to view their email inbox, and to send emails to other users, as well as to email addresses outside the system. When the user opens her inbox, she should see a list of message headers and arrival times; when she clicks on a message, she should be able to see its contents, and she should be able to delete the message, write a reply, or forward it to another address. There should also be a way to write a new message.

Minimal solution: To get something to work quickly, you could restrict email transmissions to users within your system.

Full solution: A complete solution should accept emails from the outside world. For this you can adapt the SMTP server from HW2 so that it puts incoming emails into the storage system instead of an mbox file. It should also be possible to send emails to users outside your system; for this, you'll need to add a simple SMTP client for sending emails. Please keep in mind that modern SMTP servers have a variety of anti-spam measures built in (such as greeting delays and temporary rejections); if your client does not work with external servers but works with your own SMTP server, you may want to have a look at https://en.wikipedia.org/wiki/Anti-spam_techniques.

2.5 Storage service

Users should have access to a simple web storage service, similar to Dropbox or Google Drive. They should be able to upload files into the system (which would then be stored somewhere in the key-value store), they should be able to download files from their own storage, and they should be able to see a list of the files that are currently in their account.

Minimal solution: Initially, you could just implement a flat name space without folders.

Full solution: Your final solution should also have a way to delete files, to create and delete folders, to rename files, and to move files from one folder to another.

2.6 Admin console

Your system should also contain a special web page that shows some information about the system. The page should be accessible through some special URL (say, `http://localhost:8000/admin`). At the very least, this page should show the nodes in the system (frontend servers and backend servers) and their current status (alive or down), and it should provide a way to view the raw data in the storage service. Depending on which features your team implements, you may want to add other things to this page; for instance, if your storage service implements fault tolerance, you may want to add a button that can be used to disable individual storage nodes, so you can test what happens when a node fails.

3 Suggestions

Below are some suggestions for questions you and your teammates may want to discuss during your first meetings:

- ☐ **Design:** How should the system be structured? What components should there be, and how do they interact? What would be the steps in a typical user session?
- ☐ **Responsibilities:** Which team member is responsible for each component? (You can always help each other out, but it's good to have a specific person be responsible for each piece.)
- ☐ **Schema:** How would the data be organized in the key-value store? What should be in each row? What columns should there be?
- ☐ **Protocol:** How do the frontend and backend servers interact? For instance, what port number(s) will the backend servers listen on? What is the format of the requests and responses?
- ☐ **URL space:** What URLs will there be, and what approximately will be on each page? (E.g., `/login` shows a login screen, `/inbox` shows the email inbox, and so on.)
- ☐ **Code structure:** How will you organize the repository? For instance, will there be subdirectories for each component? How will the application code (email and storage) interact with the web server? (Example: When `/login` is requested, the server calls `foo()`, which is given the user's cookie as an argument and returns the page as an array of bytes.)
- ☐ **Collaboration:** How often do you want to meet, and what would be a good time? What are the rules for Git checkins? (E.g., need to test to avoid 'breaking the build')
- ☐ **Milestones:** What would be a few good milestones along the way, and when do we want to reach them? (Be sure to include a bit of extra time at the end for integration and testing, as well as for unexpected problems.) What should happen when a milestone is not reached? What other commitments does each team member have between now and the due date?

4 Logistics

4.1 Submission checklist

Before you submit your solution, please make sure that:

- ☐ Your solution compiles properly.
- ☐ Your code contains a reasonable amount of useful documentation.
- ☐ You have completed *all* the fields in the `README` file.
- ☐ You have checked your final code into your team's Git repository.
- ☐ You are submitting a `.zip` file, which contains all of the following:
  - ☐ all the files needed to compile and run your solution (especially all `.cc` files!);
  - ☐ a working Makefile; and
  - ☐ the `README` file, with all fields completed - including the instructions for building an running your project. The instructions must be sufficiently detailed for us to set up and run your application.
- ☐ Your `.zip` file is smaller than 10MB. Please do not submit large binaries or large data files. If you got approval to use third-party material (libraries etc.) and are not including these in your submission, please state in the `README` file where these can be obtained (URL plus one sentence saying what the library does).
- ☐ You submitted your solution as a `.zip` archive via the web interface (!) before your demo. Submissions in any other form (email etc.) will not be accepted. Jokers cannot be used for the project.

4.2 Project demos

Your team must do a short demo during the finals period. A number of time slots on different days will be posted to the discussion group near the end of the semester. All team members must be physically present for the demo in order to receive credit.

4.3 Project report

Each team should submit a short project report of up to three pages. Your report should include 1) a brief description of your design, including an architecture diagram that shows the major components and how they interact; 2) an overview of the features you implemented, including any extra-credit features; and 3) a discussion of major design decisions you made and/or major challenges you encountered. Your report should also clearly identify which team member was responsible for which component(s).

5 Extra Credit ideas

Backend: One possible extra-credit feature would be support for *dynamic membership*; this would make it possible to add and remove nodes at runtime, and the data would be rebalanced if necessary. For instance, when a new node joins, the existing nodes should transfer some of their content to it, so that every node stores roughly the same amount of data.

Frontend: You could also implement some of the remaining HTTP features, such as *persistent connections* and chunked *transfer encoding*.

Email service: You could add support for *folders*, an *address book*, and/or a way to send and receive *attachments*.

Storage service: You could add a way for users to grant *access rights* to other users (perhaps by sending them an email with a URL that contains some kind of token), and/or yout could implement *storage quotas* to limit how much storage is available to each user.

Additional services: For instance, you could add a chat service, a discussion forum, or some other service.