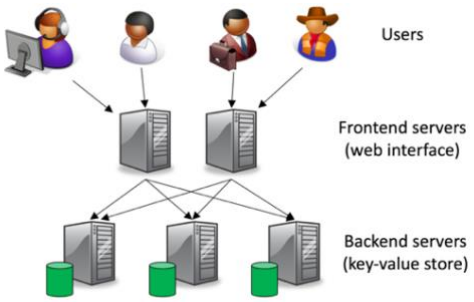
Final project: The Cloud

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 GitHub repositories for each team (after all teams have been formed). Please **do not** use your own personal 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 feel free – and are encouraged – 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 (the backend)

Your system should store all of its user data in a distributed key-value store, somewhat analogous to Google's Bigtable. (Unless you want to, you do not need to implement 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,v_1,v_2): Stores value v_2 in column c of row r, but only if the current value is v_1
- 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 {(c_1,v_1), (c_2,v_2), ...}, 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("linwang", "file-8262922", X), where X is a PDF file that user "linwang" has stored in the storage service (see below).

Minimal solution: An initial version of the backend key-value store 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 truly *distributed*: there should be several storage nodes that each store some part of the table (perhaps a certain range of rows). Note that in a typical deployment, the storage nodes would be on different physical machines, each with their own disk and memory; hence, please do not implement the storage nodes as threads of the same process or assume that they can communicate with one another using other means rather than via the network. 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 (ideally, at least three) 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 (read or modified) as long as at least one of the replicas are still alive. It

should also be able to efficiently *recover* from failures (as nodes may fail and be restarted at runtime), and it should offer some degree of scalability.

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; please see https://www.ietf.org/rfc/rfc2616.txt). 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 HW2(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) and HEAD requests, persistent connections, cookie handling and chunk transfer encoding.

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 (e.g., gmail). 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. Note: The focus here is on the functionality and not on making the service "look pretty" (or Gmail-like), so feel free to use simple HTML elements to display the emails, e.g., `<ul>` for a list of email headers or `<textarea>` for editing the text of an email.

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

Full solution: A complete solution should accept emails from outside your system, i.e., from an SMTP client (e.g., Thunderbird) running on the same host machine. (Accepting emails from remote users on other machines is much harder and requires control over a DNS entry, so this is not required.) 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 remote users outside your system (e.g., Gmail or other email accounts); for this, you'll need to add a simple SMTP client for sending emails. The SMTP client should use the DNS to look up the MX records for the recipient's domain, and connect to one of the servers that are specified in these records. 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. You may also find the suggestions in our SMTP+POP3 guide on Ed Discussion useful here.

2.5 Web 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. Notice that this is intended as a simple storage service and not as a Google Docs clone; you need to support uploads and downloads, but not creation or editing files.

Minimal solution: Initially, you could just implement a flat name space without folders. Users could upload their files with a HTML form that contains a `<input type="file">` element; downloads could simply be done using HTTP `GET`. It is okay to impose a maximum file size (e.g., a few MBs) so that each file fits into a single key-value pair.

Full solution: Your final solution should also have a way to delete files, to create and delete folders, to rename files and folders, and to move files or folders from one folder to another. Your solution must support nested folders, files of at least 10MB, and it should support both text and binary files (e.g., PDF, image, audio, video, etc.)

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, e.g., by showing a table of key-value pairs (maybe ten at a time, with prev/next buttons). It should also provide a way to disable and restart individual storage nodes, e.g., using a button, so you can test what happens when a node fails (which is useful in testing fault-tolerance) or a node is restarted (which is useful in testing recovery). It is okay to implement additional methods (besides PUT/GET/...) in your storage system to support the admin console; for instance, a function for listing row keys may turn out to be useful.

3 Implementation notes

This section contains some tips and suggestions; these are not part of the specification and are simply meant to make your job easier. Feel free to implement your system differently.

3.1 Organizing the communication

You will probably find that the components in your system have to communicate with each other frequently. For instance, if the user wants to view the contents of her storage folder, her browser would send a request to one of the frontend servers, which in turn would have to send some 'GET' or 'PUT' requests to the storage nodes. The details of how the frontend server does this are up to you; however, one simple approach would be to have a 'server loop' in each storage node that opens a TCP port and listens for incoming connections, just like the SMTP and POP3 servers did in HW2. When the frontend server wants to look up some key-value pairs, it would open connections to the storage node(s) it needs to talk to, and send its requests over these connections – perhaps some kind of string. For instance, if the frontend server wanted to delete a key-value pair, it could send something like `DELETE row123 key456`. (The details of the protocol are up to you!) The storage servers could then parse the requests and send responses over the same connection – again, just like the SMTP and POP3 servers did in HW2.

You do not need to worry about fancy authentication schemes or encryption; if you add these, it would be considered extra credit. Also, you could consider using third-party RPC/serialization frameworks, such as Google's protobuf/GRPC, Apache Thrift, or Boost, for the server-to-server communication (only); however,

please remember to ask for explicit permission on Ed Discussion before you use any third-party libraries or third-party code.

3.2 Load balancing and fault tolerance in the front-end

Recall from the first page that the frontend is supposed to be replicated across multiple machines, for load balancing and fault tolerance. This raises the question how clients would pick the machine that they want to connect to – users generally won't know the IP addresses of the frontend machines or how many of them there are, and they certainly won't want to manually type these addresses into their browser. Many data centers contain a network-level load balancer component for this that transparently redirects each new connection to one of the frontend servers. This is a little beyond the scope of this project, but you can approximate this in various ways; for instance, you could build a tiny special-purpose "web server" (load balancer) that accepts the initial HTTP request from new clients and simply responds with a temporary redirect to one of the real frontend servers. Thus, clients would only need to know the address of this special web server (and this would also be the address that would be stored in the DNS). The special web server could keep track of which frontend servers are "alive" at any given moment and/or how busy these servers currently are, and redirect new requests to one of the "live" servers, perhaps even the least busy one. Notice that this special "web server" would only be involved in the first request from each client; after the redirect, the client would send further requests to the chosen frontend server directly.

To achieve good fault tolerance, it is probably a good idea not to keep 'hard' state on the frontend servers. If you store all the state (user accounts, files, emails, ...) in the key-value store, the failure of a frontend server should not affect clients very much: they can simply connect to the site again and be redirected to a different frontend server; in this case, all of their data would simply be loaded from the key-value store again. Having said this, you may want to cache key-value pairs on the frontend servers for a short amount of time in order to improve performance. You could use the conditional put primitive (CPUT) to prevent inconsistencies: for instance, you could include a version number in important key-value pairs and, whenever a frontend server needs to change a key-value pair that is in its local cache, it could issue a CPUT with the cached value and the new value. If the CPUT fails, another frontend server has modified the same key-value pair.

3.3 Page rendering and session management

You do not need to do anything fancy to produce the HTML pages your frontend servers send to the clients; you could simply write some basic HTML to an internal buffer, roughly as follows:

```
#define append(x...) do { int space = bufferSizeBytes - strlen(buffer); \
        snprintf(&buffer[strlen(buffer)], space, x); while (0)

void renderLoginPage(char *buffer, int bufferSizeBytes)
{
  ...
  append("<html><head><title>Login</title></head>\n");
  append("<body><h1>Login</h1>\n");
  append("<form method=\"post\" action=\"/checklogin\">\n");
  append("<input name=\"username\" size=\"10\">\n");
  ...
}
```

Then you could write the buffer back to the client over the TCP connection, just like you sent back emails in your POP3 server from HW2.

You could use cookies to identify different clients. To send a cookie to a client, include a `Set-Cookie:` header in your HTTP response (Example: `Set-Cookie: sid=123`). This will cause the browser to store the key-value pair (here, `sid=123`) in a local file, and it will include the key-value pair in all subsequent requests to the same server (as a `Cookie:` header, e.g., `Cookie: sid=123`). One way to use this is to associate requests with clients. Suppose, for instance, that client A sends a HTTP GET request to ask for the login form, and is given a `sid=123` cookie as sketched above. Later, client A sends a HTTP POST with her username and password, say `linwang/secret`. Since the cookie is included in that request, the frontend server can remember that the client with the `sid=123` cookie has logged in as `linwang`. If this client now sends a HTTP GET for the email inbox page, the server will recognize that this is `linwang`, and it will return her emails; if a different client sends the same request, it will have a different cookie, or no cookie at all, so it will be shown a different inbox, or be redirected to the login page first. (Obviously, for this to be secure, the cookies have to be random and hard to guess.) For more information about cookies, please see RFC6265.

3.4 Partitioning the backend storage

One way to divide up the key-value pairs between the storage nodes is to define ranges of row keys, similar to BigTable's "tablets", and to assign each range to a specific storage node, or to a set of storage nodes. For instance, suppose the row keys are alphanumeric; then you could have one tablet for row keys that start with a certain range of variable-length prefix of alphabets (e.g., a tablet for row keys with prefix in '`aa-af`', another tablet for row keys with prefix '`ag-az`', and so on). Or, you could also assign the row keys to tablets based on some hash of the row keys, etc. The coordinator could keep the mapping from ranges to storage nodes, and it could give the mapping to clients upon request, who could then send their GET and PUT requests to the relevant storage node(s) directly. The tablets should be small enough to allow good load-balancing (if you have one huge tablet, one poor storage server has to do all the work!), but they should not be too small, either (if you have one tablet per row key, lots of bookkeeping will be required). The ranges could even be dynamic; for instance, you could start with a few big tablets and then 'split' tablets at runtime once they become too large.

If your design contains a coordinator node, please avoid putting it on the 'critical path', e.g., by involving it in every single GET or PUT operation, or potentially even sending all the data through it. It is fine to have the coordinator do some coordination (e.g., remember which row ranges are stored where, keep track of which storage nodes are currently alive, trigger re-replication of failed tablet copies), but all the "heavy lifting" should be done directly between the clients and the storage nodes.

3.5 Consistency, fault tolerance and recovery

If your storage servers are multithreaded, you'll need to use locks to prevent inconsistencies if multiple clients are issuing PUTs and GETs in parallel. There are many possible locking schemes – for instance, locks could be associated with tablets, rows, or even individual cells. Your team should think about the tradeoffs, and then decide.

It is important to have a way to handle node crashes, so the data doesn't become inconsistent if a storage node fails. One simple way to do this is to keep all the tablets in memory and to keep an append-only 'log' on disk in which all operations (GET, PUT, ...) are recorded. Perhaps there could be one log per tablet. If the server ever crashes, you can 'replay' the operations in the log to get back to the same state the server had before the crash. To prevent the log from growing too much, your server could periodically write a checkpoint of each tablet to disk and then clear the corresponding log; that way, the server only has to start replay from the most recent checkpoint. Notice that this approach limits the capacity of your key-value store to the amount of available memory, but you could always use the memory only as a cache and keep only those tablets in it that are currently being used. If the cache is full, you can 'evict' tablets from it by writing a checkpoint to disk and then erasing them from memory.

You should keep more than one copy of each tablet in your system (i.e., store each tablet in several backend servers), so as to avoid data loss if a storage server fails. Several different consistency models for replicated storage (sequential consistency, eventual consistency, etc.), as well as several techniques for achieving them (primary/backup, quorum-based replication, etc.) are discussed in Lecture 11 (Replication). We recommend sequential consistency, since it is most desirable for our setting (which needs to support concurrent updates to the same row key) while also being more intuitive to reason about (e.g., easier to debug if something goes wrong). Your team should pick a suitable technique for achieving consistency and there is a wide range of possible choices; my only advice is to stay away from Paxos (to be covered in later lectures), at least initially, since this protocol is quite complex.

You need to have a way for storage servers to get back 'in sync' with the others after a crash. For instance, suppose servers A and B are storing copies of tablet T, but then A crashes and is restarted after a while. A's copy of T may be outdated at that time. One simple way to deal with this is for A to ask B for a recent checkpoint of T. B could then keep an eye on the corresponding log file and send that to A once the download has completed; at that time B could briefly suspend write (but not read) operations on T to allow A to finish the download and to confirm that it is up to date. At that point A and B are in sync again. (This is just a simple example of a recovery protocol; there are many other, more elaborate/ambitious options!) You do not need to handle all possible corner cases for this project (e.g., what if B fails too at this point?); the important thing is to have a solid, thoughtfully designed protocol.

If your design contains a coordinator node, a failure of the coordinator is another possibility to consider; however, this should be a fairly low priority – please do not implement fancy recovery schemes for the coordinator until all the other components in your system are working very well.

4 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. In the past, a common division is to have two members responsible for the backend and two for the frontend.)

- ☐ **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 check-ins? (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?