CIS 5050: Software Systems

Spring 2025

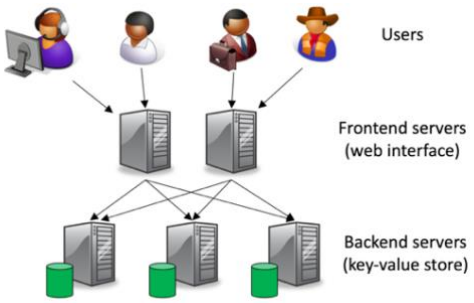
Final project: PennCloud

Teams must form by March 7, 10:00pm ET
Project proposal due on March 19, 10:00pm ET
Work-in-progress demo I (minimum solution) on April 1-4
Work-in-progress demo II (intermediate solution) on April 15-18
In-person final project demo on May 5-9
Project code and report due on May 9, 10:00pm ET

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 critical 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, an intermediate solution, and a complete solution for each component. The first 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 second represents an intermediate milestone that your team would need to complete to be on track. The complete solution

represents the functionality your team would need to get full credit for the project. (On Pages 14-16, we include a checklist of features to help you keep track of the progress.) 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. If you do add your own features, it is recommended to check with us in advance.

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("linhphan", "file-8262922", X), where X is a PDF file (i.e., a sequence of bytes representing the file content) that user "linhphan" has stored in the storage service (see below) and "8262922" is a unique-ID of the file (similar to the UIDL of an email in POP3).

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 PUT, GET, CPUT and DELETE defined above (you and your teammates can define your own protocol), and stores the data locally. The server could be multi-threaded or event-driven; if you are going with multithreaded solution, you should be able to reuse some of your HW2MS1 code for this.

Intermediate solution: An intermediate version of the backend key-value store should be truly *distributed*: the key-value store (the giant table) should be partitioned into multiple smaller *tablets* (also called partitions) – e.g., each tablet could be a certain range of rows. There should be several storage nodes that each store some subset of the tablets. Note that in a typical deployment, the storage nodes would be on different physical machines, each with their own disk and memory; hence, they must be implemented as processes (and not threads of the same process), and they must communicate with one another only via the network

(i.e., you should not use IPC mechanisms or assume they can read each other's tablets). 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. Your system should also *replicate* the data – that is, each value should be stored on more than one (ideally, at least three) storage node. 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) if some replicas are still alive. The system should offer some degree of *scalability*.

Full solution: A complete version should also provide efficient *fault detection* and fault handling, as well as *fast recovery*. It should be able to efficiently detect a node fault, and it should handle the fault accordingly to ensure that *the system continues to be available and consistent as long as at least one replica is alive*. For example, suppose your solution uses primary-based replication, then if a secondary node of a tablet fails, the primary node should exclude it from the rest of the replicas for the tablet; if the primary node fails, the system should select one of the alive secondary nodes to be the new primary. Likewise, suppose your system uses quorum-based replication, then if some replicas fail and the remaining number of alive replicas is less than the read (or write) quorum size, the system should update the read (or write) quorum size to ensure that clients can continue to read and write while avoiding read-write and write-write conflicts. Each storage node should also perform *periodic checkpointing* of its tablets (i.e., write the contents of the tablets to disk) and *logging* of update requests (i.e., append the request to the log file before applying it on the tablet in memory). The system should also perform *recovery* when a replica becomes alive, based on its local checkpoint and log files and those of a currently active replicas (e.g., the current primary's).

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 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.

Intermediate solution: An intermediate version of the server should support additional features, including support for POST requests (for submitting web forms and uploading files to the storage service) and HEAD requests, as well as persistent connections and cookie handling.

Full solution: For a fully functional server, you will need to extend your intermediate solution to support chunked transfer encoding. In addition, you will need to support multiple frontend servers and a way to distribute clients among them via a frontend load balancer. If any frontend server is down, the load balancer can redirect new clients to a different server that is alive.

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.

Intermediate solution: The intermediate solution should also allow users to sign up for a new account, and users should be able to log in using their username and password, change their passwords, and access their webmail and storage services. (Note: A full solution has the same features, but since you need to complete this much earlier, we list them as an intermediate milestone.)

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 core focus here is on the functionality and less 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 PennCloud system. The webmail interface should be able to show the user's Inbox that contains emails (with header information about senders, subject, and time for each mail). The user should also be able to view an email and compose a new email.

Intermediate solution: An intermediate solution should also 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. You should also extend the webmail interface to enable user to perform all common mail actions, such as view, reply, forward and delete emails.

Full solution: A complete solution should allow users to send emails to remote users outside your system (e.g., Gmail or SEAS 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 the Supplemental Notes of HW2 handout 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.

Intermediate solution: An intermediate solution should support all common file operations, including upload, download, delete, and rename files. It should also support hierarchical folders and folder operations such as folder creation, (nested) folder deletion, rename folders, and move files and folders from one folder to another folder.

Full solution: Your final solution must also be able of 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 `linhphan/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 `linhphan`. If this client now sends a HTTP GET for the email inbox page, the server will recognize that this is `linhphan`, 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 10 (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?

5 Logistics

5.1 Project proposal and first meeting with TAs

Each team will be assigned a lead TA, who will be your main point of contact for the project. Your team's lead TA will closely monitor your progress, so you should check in with your lead TA regularly (in-person, via Zoom, or over emails). If there are any issues or potential concerns that arise while working on the project, please do let your lead TA and me (Linh) know as soon as possible.

Your team must meet with one of the TAs before the project proposal deadline; we have several office hours available, so please feel free to attend any of the office hours. The goal of this meeting is for you to present your proposed ideas to the TA and seek feedback from the TA before finalizing your project proposal. If you cannot make it to the office hours, you may discuss your proposal with your lead TA offline.

Each team must submit a short project proposal by the due date on the first page of this handout. The report should be a PDF up to 3-4 pages; one member of each team should email it to me and your lead TA by the due date on the first page. Please use "`CIS 5050 PennCloud Proposal: Team XX`" as the subject of the email, and please name your proposal as `PennCloud_Proposal_XX.pdf`, where `XX` is your team number (to be assigned). The proposal should contain at least the following:

- a description of your design (e.g., a list of major components and how they interact);
- a brief summary of the design decisions (e.g., what consistency model you will support and how replication will be handled, how to go about recovery, how to do load balancing);
- a division of labor (i.e., which team member(s) will be responsible for what);
- a collaboration plan (e.g., how often are you planning to meet, how will you coordinate, etc.); and
- a set of milestones for the project (i.e., what is going to be finished when).

The proposal will not be graded; a major goal is to make sure that each team is on track, and to point out potential pitfalls early in the project, if any. (We won't hold you to the design decisions in the proposal; you can always change them later if necessary.) To provide an extra incentive, we may award a bit of extra credit to a small number of teams that submit particularly thoughtful proposals and/or have already made a substantial amount of progress.

5.2 Work-in-Progress demos

Each team must present two work-in-progress demos with your lead TA during the dates specified on the first page of this handout. For these demos, your team need to present the system you have built thus far and to discuss your next steps. Again, a major goal of the demos is to make sure that each team is on the right track. For instance, it is very essential that you have already had the minimum solution working by your first work-in-progress demo, and the intermediate solution working by the second demo.

The intermediate milestone is not strict – it is okay for you to implement the system in a different order (e.g., implement some features and functionalities listed in the full solution before those listed in the intermediate solution). However, it is important that you have completed roughly equivalent portion of the work to be on the right track. Your work-in-progress demos will be recorded, so that I can review them.

5.3 Final project demos

Your team must do a short demo during the finals period. A number of time slots on different days will be provided near the end of the semester, and your team will need to sign up for a slot. **All team members must be present physically for the final demo to receive credit.** Before your final project demo (with me), it is *strongly encouraged* that your team have a rehearsal demo with your lead TA.

5.4 Project code submission checklist

Each team should designate one member to submit via GradeScope the complete project code by the deadline stated at the beginning of this handout. 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 can reuse the README file template from the homework assignments.)
- ☐ You have checked your final code into your team's GitHub 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;
  - ☐ the `README` file, with all fields completed – including the instructions for building and running your project (the instructions must be sufficiently detailed for us to set up and run your application); and
  - ☐ at least five screenshots that show your most impressive features and/or your best results.
- ☐ Your `.zip` file is smaller than 20MB. 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 GradeScope by the due date on the first page of the handout. Jokers cannot be used for the project.

5.5 Project report

Each team should submit a short project report of about four to five pages. The report should be a PDF file (called `PennCloud_Report_XX.pdf`, where `XX` is your team number), and it should be submitted via email to me (Subject: `CIS 5050 PennCloud Report: Team XX`) by one member of each team by 10pm ET on the due date specified on the first page of this handout. Your team's report should include:

- a brief description of your design, including an architecture diagram that shows the major components and how they interact;
- an overview of the features you implemented, including any extra-credit features; and
- a discussion of major design decisions you made and/or major challenges you encountered.

Many teams should be able to reuse at least some text from their proposal document for the report. Your report should also clearly identify which team member was responsible for which component(s).

6 Evaluation criteria

The project will be evaluated holistically based on the following key criteria (in no particular order):

- The quality of your architecture and design.
- The supported features: Does your system implement all required features?
- Stability: Is the system stable? Does it work seamlessly without any problem?
- Performance: How fast is your performance? Is there any potential bottleneck inherently in your design and implementation? Etc.
- Scalability: How well does your design scale to many users and to a very large amount of data?
- Consistency, fault tolerance and recovery: Does your system continue to work correctly and ensure consistency even when some node fails, when all but one node fail? Can a node recover state correctly after being restated? Etc.
- Usability: Is the system easy to use and intuitive to the users?
- Report: How well written is your report? Does it describe the system well? Is it insightful?
- Implementation: Quality of your code and implementation, as well as your documentation.

These criteria will be evaluated based on knowledge I obtain from your final project demo (very critical!), the project report, the work-in-progress demos, as well as by examining your code and running it.

6 Extra Credit ideas

Backend: One possible extra-credit feature would be support for *dynamic membership*; this would make it possible to add and remove backend 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.

Email service: You could add support for *folders*, an *address book*, and/or a way to send and receive *attachments*. You could also enhance the user interface to look beautiful.

Storage service: You could add a way for users to grant *read and write access rights* to other users (perhaps by sending them an email with a URL that contains some kind of token), and/or you could implement *storage quotas* to limit how much storage is available to each user. You could also support larger files (e.g., hundreds of MBs or GBs) by spreading them across multiple key-value pairs, or you could provide a way to upload and download files via FTP.

Additional services: For instance, you could add a chat service, a discussion forum, a message board, a game, or some other service of your choice.

Deployment: Instead of running your system on your local machine, you could deploy it on the cloud, e.g., on Amazon Web Services. This provides you an opportunity to deploy your system on a real distributed environment, which each frontend and backend server runs on a separate EC2 instance. If you are interested in doing this, please let me know so that I can investigate obtaining free credits from AWS Academy.

Please consider the extra credits only after you have completed and carefully tested the full solution. Since not all extra credits are equal, we will grade extra features based on the quality of the implementation and based on their difficulty. Please think carefully which ones you want to attempt based on the available time you have remaining. The above ideas are only suggestions – you may implement your own ideas for the extra credits. Again, all creative ideas for the extra credits that are related to concepts taught in class are welcome, but please do consult with us in advance.

PennCloud Feature Checklist

1) INTEGRATION:

All components (e.g., frontend servers, frontend load balancer, backend servers, backend coordinator, admin console) are integrated into a single working end-to-end system.

2) ACCOUNT:

- ☐ Signup for a new user, sign in, password check, password change.
- ☐ Main page showing webmail, drive, and account services.
- ☐ Cookies working (e.g., opening a new tab will direct to the same user, unless it is in a private tab).
- ☐ Concurrent login: two different users can log in concurrently (e.g., by using a public tab and a private tab, or by using two different browsers) and operate on their emails, drives and accounts concurrently. In addition, different clients of the same user can access the system concurrently.

3) FRONT-END LOAD BALANCER:

The user should access PennCloud system by providing the address of the front-end load balancer on the browser address, and the browser should redirect the user to an available front-end server (e.g., using a suitable strategy such as random, round robin, least load, etc.).

4) FRONTEND SERVER:

- ☐ Support for HTTP requests: GET, POST and HEAD.
- ☐ Support for persistent connections.
- ☐ Support for cookie handling.
- ☐ Support chunk transfer encoding.

5) DRIVE:

- ☐ Show list of files and folders, support navigation between folders (including nested folders).
- ☐ Support both text and binary files (including, e.g., PDF, image, video, and audio).
- ☐ Support file size of at least 10MB for both text and binary types.
- ☐ Upload files to any folder.
- ☐ Download files from any folder (including root); downloaded files are displayed correctly.
- ☐ Upload and download speed is fast (i.e., should be almost instantly for a 10MB file, and no more than 1-2 seconds).
- ☐ Rename files and folders; renamed file and folder names are properly displayed (should not contain the entire path!)
- ☐ Move files and folders between folders (at any level).
- ☐ Delete files and folders (recursive delete for non-empty folders).
- ☐ Proper choice of row key and column key to enable scalable operations (e.g., moving or renaming).

6) MAIL:

- ☐ Display Inbox containing a list of emails, with header information such as Subject, Sender, Time.
- ☐ View any email in the list.

- ☐ Delete an email.
- ☐ While viewing an email, the user can reply and forward the email to another recipient.
- ☐ Compose a new email to a local user within PennCloud.
- ☐ Compose a new email to a remote user (e.g., linhphan@seas.upenn.edu or your gmail account).
- ☐ Receive emails from Thunderbird (sent to a local user within PennCloud).

7) ADMIN CONSOLE:

- ☐ Show front-end and backend servers.
- ☐ For each server, show the server's address, status, on/off features to turn on/off the server, the tablets stored on each server organized in rows (e.g., using some tables, with next page or scrolling features).
- ☐ The backend KV store must be partitioned into many tablets, which are distributed to different groups of servers. Thus, no server can contain the entire KV store.
- ☐ Validate that you have at least 3 copies of each tablet on 3 different servers.

8) BACKEND COORDINATOR:

- ☐ Maintain the partitioning of the KV store into tablets, and the mapping of tablets to backend nodes.
- ☐ If primary-based replication is used, need to keep track of the current primary and the secondary nodes for each group (or each tablet).
- ☐ Need to monitor the status of backend nodes
- ☐ Need to detect faults of backend nodes and select a new primary if needed.
- ☐ Need to allow a frontend server to LOOKUP an address of the server that store data for a user (e.g., when a user logs in, or when the current backend server serving the frontend server is down).
- ☐ Must not serve PUT/GET/CPUT/DELETE requests! These should go directly to the backend servers.

9) BACKEND SERVERS:

- ☐ Implement the KV interfaces: PUT, GET, CPUT, DELETE operations
- ☐ Implement proper synchronization for concurrent requests
- ☐ Implement periodic checkpointing of the tablets.
- ☐ Implement logging of write requests before applying to the memory tablet(s).
- ☐ Implement a replication protocol (e.g., primary-based replication with remote write).
- ☐ Should support multiple tablets per server.
- ☐ Implement recovery when the server is being restarted by the user. (This should be done based on local checkpoint and log files, and/or the remote checkpoint and log files of currently alive servers.)
- ☐ Request are served in memory (not directly on disk!), with logging of update requests (see above).
- ☐ Servers do not share disk or memory with one another; they must be communicated via the network!
- ☐ All data should only be stored in the memory tablets, the checkpoint files, and the log files. The tablets contain the actual data (e.g., the content of the files and not just a path to a file); there should not be any user files on disk – all users' files are serialized in the checkpoint files or stored in the memory tablets).

10) FAULT TOLERANCE AND FAULT DETECTION:

- ☐ The coordinator should detect that a node is inactive when a server is killed.
- ☐ Need to support both killing a server from the command line (using CRTL-C) and via the admin sole.
- ☐ An inactive server should have its status reflected as so in the admin console, and it should show any tablet information.

- ☐ Suppose you have 3 replicas per tablet. Then, the system should continue to work normally if you kill one replica server, perform some more operations (write emails, upload files, etc.) and then kill another one, then continue to do more operations.
- ☐ If primary-based replication is used and the killed node is the current primary, a new primary will be selected. If quorum-based replication is used and if there are fewer available nodes than the read/write quorum size (R and W), the system updates the value N (number of alive replicas), R (read quorum size), and W (write quorum size) to allow users to continue accessing the system.

10) FAULT RECOVERY:

- ☐ Need to support both restarting the server from the terminal (if it was killed from the terminal), and restarting the server from the admin console (if it was killed from the admin console).
- ☐ The recovering node must bring its state up to date based on its checkpoint and log files and that of the current primary (or any alive node with the latest version of the data in quorum-based replication).
- ☐ Avoid transferring the entire checkpoint and log files from another node for recovery unless necessary (i.e., only transfer the missing information).
- ☐ **A testing scenario:** Suppose a tablet is replicated on 3 servers A, B, and C. Then, populate some data into the tablet (e.g., upload files, write emails), which should show on all three servers (via the admin console). Then:
  - Kill A from the terminal – A is shown as "inactive" but the system continues to work normally.
  - Next, perform a series of read and write requests (e.g., send emails, upload files, rename folder, delete files, etc.) – the requests should succeed and are reflected on the web interface.
  - Kill B from the admin console – B is shown as inactive, but the system continues to work.
  - Continue to issue a series of several read and write requests – all requests should succeed.
  - Kill C from the terminal – now, all replicas of the tablet are inactive.
  - Try to access the user – this will lead to system being "unavailable".
  - Restart C from the terminal – recovery should be done on node C (based on local checkpoint and log files), and the system should become available again. All data should be current (nothing missing, nothing new!)
  - Issue a series of read and write requests – everything should work as expected.
  - Restart B from the admin console, then kill C quickly afterwards.
    - B should show all the most updated data (since it must have recovered the latest data based on its local checkpoint and log files, and those of C before C was killed).
  - Issue more update requests – everything should work as expected.
  - Restart A from the terminal, then kill B from the terminal.
    - A should show the most updated data (just like before killing B)
  - Continue to issue requests and everything should work normally.

- ☐ As long as (at least) one replica server of a tablet is alive, the user (whose data are stored in that tablet) should be able to access and make updates to their drive and mails.

11) PERFORMANCE AND STABILITY:

- All operations (including fault detection and recovery) should be very fast (no more than a second or so).
- Try to deploy on your local host if possible (as the VM can be slow).
- Make sure nothing crashes in the middle!
- Stress test your system with more complex test cases.