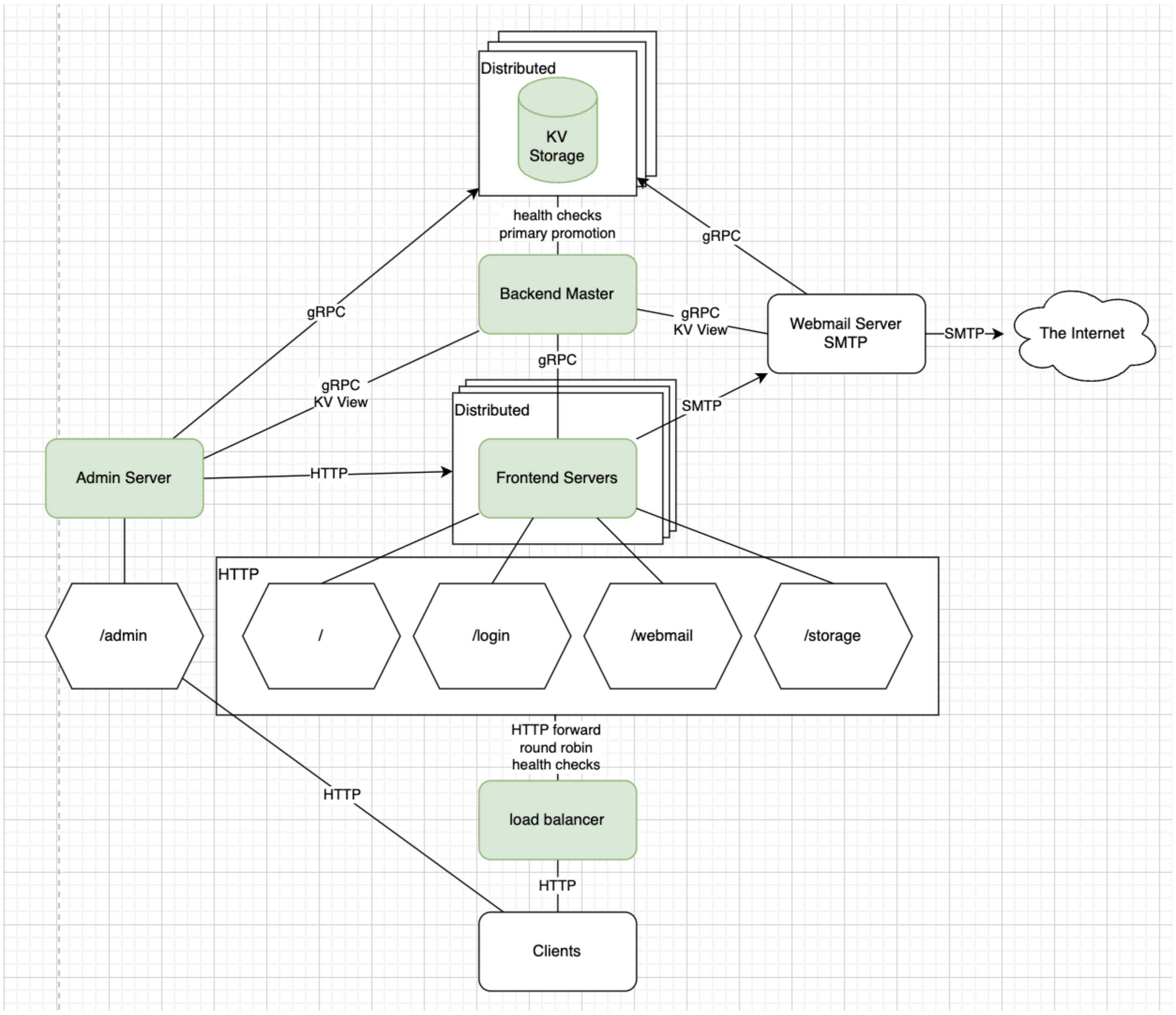
# Penn Cloud code documentation

CIS 5050 Spring 2023

Group T02: Andrew Zhu, Jingyi Lu, Jingwei Bi, Zhengjia Mao

## Overview

Penn Cloud is a small distributed cloud platform that features a webmail service, analogous to Gmail, and a storage service, analogous to Google Drive. The platform relies on a distributed key-value storage file system, analogous to Google's Bigtable. This report serves as a companion document to Penn Cloud and provides necessary information to understand the project architecture and subsystem designs.



The project architecture is shown above. You will find more detailed information about each subsystem in the sections later.

Clients are directed to one of the active frontend servers through a load balancer. The frontend servers firstly return HTML files as the interfaces for authentication, webmail service, storage service, and then handle service logics for other HTTP requests from the clients. The admin server displays and manages the status of all the frontend, backend servers, as well as all key-value pairs stored in the backend.

A single backend master node keeps track of the statuses of all kv servers. The kv servers support distributed key-value pairs storage that is modeled after Google BigTable. The frontend servers, webmail server, and admin server communicate with the backend servers via gRPC, and have the access to the kv storage and can perform GET, PUT, CPUT, DEL operations.

## Recorded Demo

https://drive.google.com/file/d/1LAjSiRZjqRHlBTJlTVG3KLC8T2JYwcrF/view?usp=share_link

This demo video showcases the various components as an integrated piece, such as authentication, webmail, storage, admin, load balancer, kv storage interaction, fault tolerance, and consistency. During the live demo, we didn't properly link the port numbers, and we also missed a newline character ("\n") in the README commands, resulting in the last line of commands not being executed. Hopefully, this demo video can better represent the project.

## User Accounts

*Contributors: JInwei Bi*

Initially, when the user first connects to the frontend server, the server responds with a homepage. If the user tries to access the services like storage and webmail, the frontend server will respond to the user with a login page because of no authorization. The user can also login in by clicking the login button on the navigation bar.

The system also allows users to sign up for a new account, and reset their password. To sign up for a new account, the server will return the register page where the user can input a new username and password. To reset the password, the user will be navigated to change the password by submitting the old password and a new password. After the user logs in, they can access the webmail or storage services and they can also logout to end their session.

### Session Manager

The system supports multiple user accounts by maintaining a “session manager”. For each front-end server, there is exactly one session manager to store and handle different session IDs for different users. The implementation of “SessionManager” class includes methods to create a session_ID, retrieve session data, update session data, delete sessions, and generate a cookie value for session management. It also includes a “session_check” method to validate session IDs in the Set-Cookie header of the HTTP request. Note that the generated session ID consists of 32 random hexadecimal digits which ensures the security and uniqueness.

### Login

After the client inputs their username and password and clicks the submit button, the frontend server will receive an HTTP POST request. The frontend server will talk to the KV store and GET the correct password through the KV client. If the KVClient cannot find the password, it should throw an error. In this case, the frontend server catches that error and responds to the client with a “username does not exist” message. If the username exists but the password is incorrect, the frontend server responds to the client with an “incorrect password” message. If the password matches, the client should get a “successful login” message and be directed to the homepage again (with authorization). From the server end, a unique and random session ID would be generated, stored in the sessions map and set in the “Set-Cookie“ header.

### Register

The user needs to input a username and password. The frontend server will get a POST request after the client clicks the registration button. Then the frontend server talks to the KV store via KVClient and PUT the password into the KV store with corresponding username.

### Reset Password

The user needs to input username, old password and new password to reset the password. The frontend server will get a POST request from the client. Then the frontend server will talk to the KV store via KVClient and update the password using CPUT primitive.

## Admin

*Contributors: Jingyi Lu*

The Admin part contains Admin webpage (HTML, JavaScript), Admin Console (C++) and Admin Server (C++). It keeps doing health checking on both frontend and backend nodes every 2 seconds, and shows their current status(alive or down). It also provides a way to view raw data in storage. It enables clients to disable and restart individual storage nodes on the Admin webpage by clicking buttons.

The Admin console is a subclass of KVMaster Client. The KVMaster Client is an implementation of a Key-Value (KV) store master server using gRPC. The server is responsible for managing the health of nodes within the KV store and updating the primary node for each partition in case the current primary goes down. In this way, the Admin console inherits all methods and attributes of KVMaster Client

## Health Checking

For the backend nodes, Admin Console can just call the health checking method in KVMasterClient. For KVMasterClient, it iterates through all the partitions and their respective nodes. For each node, it performs a gRPC call to the node's health check service. This is done by calling the _healthcheck method on the node object. The _healthcheck method sends a request to the node and waits for a response to determine if the node is alive or not. Based on the response, it updates the alive status of each node in the partition. The health check process is performed periodically.

For the frontend nodes, it loops through each frontend server, creates a TCP socket and establishes a connection with the server using the frontend server's IP address and port. If the connection is successful, it reads the server's response and checks if it starts with "HTTP/1.1 200 OK". If it does, the server is considered alive. Otherwise, it's considered down.

## Restart and Shutdown Nodes

For the backend nodes, we have a global boolean variable named “serving” in KV, which is used to keep track of whether the backend node is set as “restarted” or “shutdown” by Admin. Whenever the clients restarted a node, we set “serving” to be true. Whenever the clients shutdown a node, we set “serving” to be false. Then for each KV RPC implements such as “PUT” and “GET”, we first check if “serving” is not true, then KV RPC implements only return Status Unavailable, and won’t do anything. In this way, we can restart and shutdown nodes in a soft way, so that shutdowned nodes won’t respond until it is restarted.

For the frontend nodes, it is similar logic. We create a global boolean variable named “serving” that keeps track of and updates serving status for frontend nodes. We shutdown a specific frontend server by sending a request to its /shutdown endpoint. We extract the IP address and port from the given node string, and create a TCP socket and establish a connection with the server using its IP address and port. If the connection is successful, we send an HTTP GET request for the /shutdown endpoint, read the server's response and check if it starts with "HTTP/1.1 503 Service Unavailable". If it does, the server is successfully disabled, and “serving” is set to be false. To restart a specific frontend server, it is similar logic as shutdown. The only difference is that if the server's response starts with "HTTP/1.1 200 OK", we consider the server is successfully restarted, and “serving” is set to be true.

## View Raw data

On the Admin webpage, we perform a GET request to retrieve key-value pairs from the Admin server. The Admin server can call a method in KVMasterClient that goes through each partition node, and retrieves the row and column value, since Admin server is a subclass of

KVMasterClient. Then the Admin web page can display all row and column values of each partition node. Clients can now click the “check” button, which will send a POST request to Admin server to get details about a specific key-value pair. Admin server can call kvClient->get(row, col) to get the value of the cell in a specific row and column, and send it back to the Admin webpage, so clients can see the details now.

## Run the Admin Server

Run with: `$ bazel-bin/src/fe_servers/admin_server kvconfig_full.txt`
You may want to change the “kvconfig_full.txt” file to be another file that contains the backend nodes’ addresses.

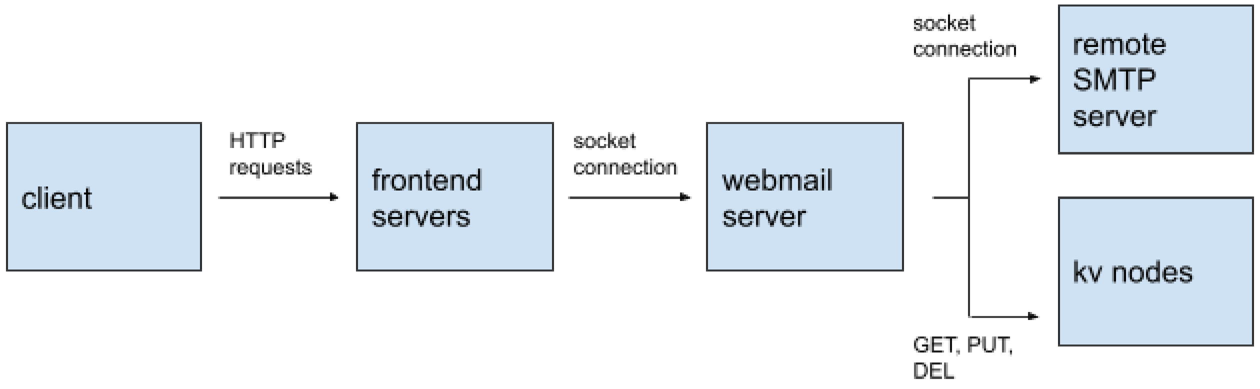
# Webmail

*Contributors: Zhengjia Mao*

The webmail service consists of a webmail client webpage (HTML, JavaScript) and a webmail server (C++). It features sending and receiving, replying and forwarding, and deleting emails.

## Email Pipeline

The webmail client takes the actions from users, formats the data to a single message that contains: headers, subject, timestamp, body, sender, recipient, UID. The client then sends HTTP requests to the frontend servers. The frontend servers handle the endpoints by talking to kv storage, or connecting to the webmail server through socket and communicate using SMTP commands. The webmail server parses the recipient domain and decides to send to remote or local. The pipeline is shown below.



## Emails in KV

In KV, (username, box) stores a list of UIDs, which each UID uniquely represents an email, and (username, UID) stores the email metadata. We have 3 boxes for each user: “inbox”, “sent”, “deleted”, and when an email is processed, both (username, box) and (username, UID) will be updated. If an email is deleted, we simply move the email UID from (username, “inbox”) to (username, “deleted”) without deleting the actual email.

## Send to Local

If the domain of the recipient is “localhost”, the webmail server PUT the email at (recipient, UID) and update the UID list of (recipient, “inbox”), also PUT the email at (sender, UID) and update the UID list of (sender, “sent”).

## Sent to Remote

If the domain of the recipient is not “localhost”, the webmail server connects to the remote SMTP server by IP address and DNS, and sends the email via SMTP commands. Then, it PUT the email at (sender, UID) and updates the UID list of (sender, “sent”).

## Run the Webmail Server

Start the kvmaster, kv nodes, and frontend servers (with load balancer) first:

```
$ bazel-bin/src/webmail/webmail -v
```

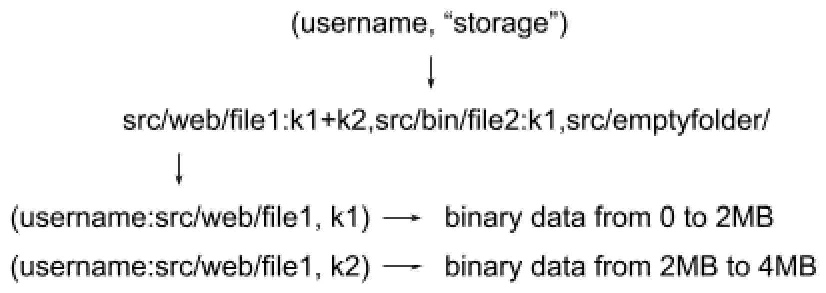
If want to talk to webmail server directly as a client:

```
$ telnet localhost 2500
```

# Storage

*Contributors: Zhengjia Mao, Andrew Zhu*

The storage service is delivered through a storage client webpage (HTML, JavaScript), and the operations are handled via kv storage (the storage handling code is in frontend_server.cc). The features include uploading and downloading files, creating folders, renaming files and folders, moving files and folders. The size limit of file uploading is 16MB.


(username, “storage”)
src/web/file1:k1+k2,src/bin/file2:k1,src/emptyfolder/
(username:src/web/file1, k1)
binary data from 0 to 2MB
(username:src/web/file1, k2)
binary data from 2MB to 4MB

## Storage File Structure

In kv, (username, “storage”) stores a list of file paths, and the actual file is stored as (username:filepath, key), as shown above. A folder ends with “/” and is not actually stored in kv except a path. Only files will have the keys.

## File Operations

When a new file is uploaded, the file is partitioned into segments of 2MB, and the segments are assigned keys and stored across the kv nodes.

When a file is requested to download, the keys associated to the file path are used to retrieve the segments from kv nodes and reassemble to respond.

When a file is moved or renamed, we simply update the path list and redistribute the segments in kv (GET from the old path, PUT to the new path, DEL from the old path).

When a file is deleted, we update the path list, retrieve the keys, and DEL all segments in kv.

## Folder Operations

When a folder is moved or renamed, we update all the paths that contain the old path as prefix, and redistribute the segments in kv. For the subfolders, only the paths are updated. For the subfiles, we retrieve the keys, GET the segments from the old path, PUT to the new path, DEL from the old path.

When a folder is deleted, we find all the paths that contain the path of the folder to delete as prefix, and retrieve the keys to DEL all segments in kv, and then update the path list.

When a folder is created, we simply update the file path list, since a folder is not actually stored in kv, and only files will have the keys and stored as segments.

# Frontend Servers

*Contributors: Jinwei Bi, Jingyi Lu*

The frontend servers handle the HTTP requests from the web browser (clients). Based on the request endpoints, the servers talk to the corresponding components to process the requests and return responses to the clients. We will have more than one frontend server, managed by the load balancer.

## Load Balancer

We implement a load balancer for frontend servers, so that clients only need to connect to load balancer, and load balancer redirects clients to an available frontend server. The load balancer listens on port **8090** and accepts incoming HTTP requests from clients. It then redirects the clients to one of the frontend servers with the least number of clients. The frontend servers are assumed to be running on the same machine (127.0.0.1) with port numbers **8080**, **8081**, and **8082**.

The load balancer uses a map to store frontend server ports and their corresponding client counts. It also uses a map to store the status of each frontend server (alive or not). It uses a mutex to protect shared resources.

The load balancer selects the next frontend server by determining which server currently has the fewest clients connected to it. It first checks if the server is alive and then compares the number of clients for each server. The server with the least clients is chosen as the target for the incoming client request. This approach ensures that the load is distributed evenly among the available frontend servers.

Run with: `$ bazel-bin/src/fe_lb/loadbalancer`

## Run the Frontend Server

To run the frontend server, run:

```
$ bazel-bin/src/fe_servers/frontend_server <port no>
```

For example, to run node with address 127.0.0.1:8080, you should run:

```
bazel-bin/src/fe_servers/frontend_server 8080
```

You can run multiple frontend servers by specifying different port numbers.

# KV Store

*Contributors: Andrew Zhu*

The KV store is divided into 2 components: the master node, which supervises each of the storage nodes and handles sending the storage node view to clients, and the storage nodes, which partition and replicate the data. The system uses a static config file defined by the user to layout the partitions and replicas per partition.

The KV store guarantees sequential consistency within each partition (if partitions are independent, globally). Within each partition, it can tolerate *p - 1* faults, where *p* is the number of nodes in the partition. Each node implements logging and checkpointing facilities to minimize the risk of data loss, and is capable of communicating with its peers to learn of writes during faults.

Clients are independent of the KV store's layout and only need to know the address of the master to interact with the system. In our project, clients (i.e. frontend, webmail) used a high-level KVClient implementation we wrote, which provides GET/PUT/CPUT/DELETE primitives.

## Replica Definition

To define the nodes, create a file in the following format:

```
master IP + port
partition node IP + port,node IP + port,...
...
```

For example, to run 3 data partitions with 3 storage nodes per partition:

```
127.0.0.1:5000
127.0.0.1:6000,127.0.0.1:6001,127.0.0.1:6002
127.0.0.1:7000,127.0.0.1:7001,127.0.0.1:7002
127.0.0.1:8000,127.0.0.1:8001,127.0.0.1:8002
```

This lays out the KV service with one master node at :5000 and three data partitions, each with three replicas. The partitioning and replication logic is defined below.

## Run the Server

To run the server, run:

```
$ bazel-bin/src/kvmaster <path to config>
$ bazel-bin/src/kv <db file path> <log file path> <path to config> <partition
no, 1-indexed> <node index, 0-indexed>
```

For example, with the config defined above, to run the node with address `127.0.0.1:7001`, you would run the command `bazel-bin/src/kv data/db.bin data/log.bin config.txt 2 1`.

## Partition Logic

The KV server partitions the data across the defined partitions by the row key. Specifically, if there are *N* partitions defined, the data is partitioned as such:

Nodes in partition *i* are responsible for all rows with key *r* such that `lsb(md5(r)) % N == i` (assuming little-endian), where lsb() is the least significant byte.

## Replication Logic

When a KV node starts up, it calls the master to figure out who the primary of their partition is.

When a KV node gets a request for a write:

- if it does not know who the primary of the partition is, it returns an error and contacts the master to learn who the primary is
- if it is the primary, it multicasts the write to its partition and applies it locally
- if it is not the primary, it forwards it to the primary

## Logging and Checkpointing

The KV server logs all write (i.e. PUT, CPUT, DELETE) requests to a local append-only log, which it uses to replay operations after a crash which does not allow for a complete checkpoint to be saved.

This log is checkpointed every 1 minute.

The log consists of a sequence of either of the following messages. We include the number of bytes of each argument in the header as all fields are allowed to contain arbitrary character sequences, including newlines and null bytes.

```
PUT <SP> rowbytes <SP> colbytes <SP> databytes <LF>
row <LF>
col <LF>
(databytes bytes of data follow) <LF>
```

As CPUTs are resolved by the primary (which decides whether the condition holds), CPUTs are logged as PUT operations.

```
DELETE <SP> rowbytes <SP> colbytes <LF>
row <LF>
col <LF>
```

The checkpoint consists of a similar format: a sequence of the following message, in an arbitrary order.

```
DATA <SP> rowbytes <SP> colbytes <SP> databytes <LF>
row <LF>
col <LF>
(databytes bytes of data follow) <LF>
```

## Tablet Sync

If a KV node goes down while other KV nodes in its partition (its "peers") are still serving requests, it will communicate with its peers to bring its local copy of the partition up to date with the rest of the peers. This takes the form of a streaming RPC (as a tablet can be larger than a single gRPC message).

Writes are locked (by locking writes to the log file) from the peer serving the sync until the data transfer is complete; reads are still served during this time.

## Health Checking / Primary Failover

The KV master node sends a healthcheck request to each data node every second, and maintains a view of healthy/unhealthy nodes. Clients are not required to sync their view with the master; they can simply retry on another data node until their request succeeds.

If a partition’s primary node becomes unhealthy, the master node will choose a new node in the partition to appoint as the primary. It then multicasts a message to all nodes in the partition to inform them of the new primary. When the old primary rejoins the partition, it contacts the master to learn of the new primary.

## Test Scripts

We also include a number of test/standalone programs to demonstrate the features of and interact with the KV store. These are:

**KV Test**
Each time the script is run, it connects to a KV master at 127.0.0.1:5000 and increases the value of `(test, foo)` by 1 each second. If the key is undefined, set it to 1.
Prints the value of the key before it is updated.

```
$ bazel-bin/src/test/kvtest
```

**KV Shell**
A simple shell to demonstrate operations against the KV store. Connects to a KV master at 127.0.0.1:5000.

You can use the following commands to interact with the KV server:

```
PUT row col data
GET row col
CPUT row col v1 v2
DELETE row col
```

Run with:

```
$ bazel-bin/src/test/kvshell
```

# Improvements

Due to time constraints, we had to make a number of compromises or simplify components of the project, which introduced additional limits that our system would encounter in the real world. Here, we document some of these limitations, as well as proposed changes to circumvent them.

## KV Store: Peer Tablet Sync

When a KV node comes online, it asks a peer for the latest version of its partition. While this guarantees that restarted nodes have the latest version of the data, it also adds a huge bandwidth load to the network when a node restarts. While this was acceptable for the penncloud project (as we only stored a small amount of data for the demo), a real world system would not want to transmit all the data in a tablet.

Instead, each node could track a version number for each key (row, col), or a sequence number of write operations (as we enforce sequential consistency within each partition). Then, on startup, a restarted node could request only the keys that have changed since it went down (i.e. have a different version number), or the logged operations since it went down (i.e. log entries with a sequence number higher than the last logged op).

## Frontend Server: Streaming HTTP Requests

Due to limited time, we were not able to implement streaming HTTP requests/responses. This meant that we had to enforce a request limit of 16MB - well over the 10MB requirement for the project, but still far less than modern systems allow for.

The difficulty lies in that we process all HTTP requests atomically; refactoring the request handling code to parse the headers then chunk the request body would require refactors over the entire code base, which we are unfortunately limited by time.

In the future, we would like to refactor our HTTP logic into two parts:

1. Read the request headers - this happens atomically and gives us information about the rest of the request, like the body length (in the Content-Length header).

2. Stream the request body - this would be endpoint-specific. Some endpoints only take metadata (e.g. deleting a file, logging on) and can read the entire body into a buffer. Other requests can stream the content (e.g. uploading a file writing chunks to KV), allowing us to process large requests in small chunks without using a large amount of memory.

Similarly, we would refactor our HTTP response logic similarly, sending our response headers and body separately - this would primarily reduce memory usage when sending large files in storage back to the client.

## Storage: Inode-like Layout

When a user uploads a file, the storage system takes 2 main steps:

1. Write the file content to the KV store.
    a. If the file is >2MB, split the file into multiple chunks, and record which keys the file lives in.
2. Update the user's metadata record.
    a. This is a list of all the files the user has uploaded, plus which keys the data for each file lives in.

This system allows us to upload large files and is a simple flat-directory implementation ("directories" are just file prefixes, similar to S3). The main limitation to this approach is the size of the metadata record - if the user uploads hundreds of thousands of files, the row will grow quite large.

To improve this, we could take a page from inodes - have each directory be its own key in the KV store, and contain a list of both files in the directory and keys where other directories are stored. This would allow each directory to grow to hundreds of thousands of files, rather than enforcing the limit on the flat record. However, this would also make requests slightly slower, as clients would have to traverse the file tree to find the entry for a specific file. As always, it's a tradeoff!