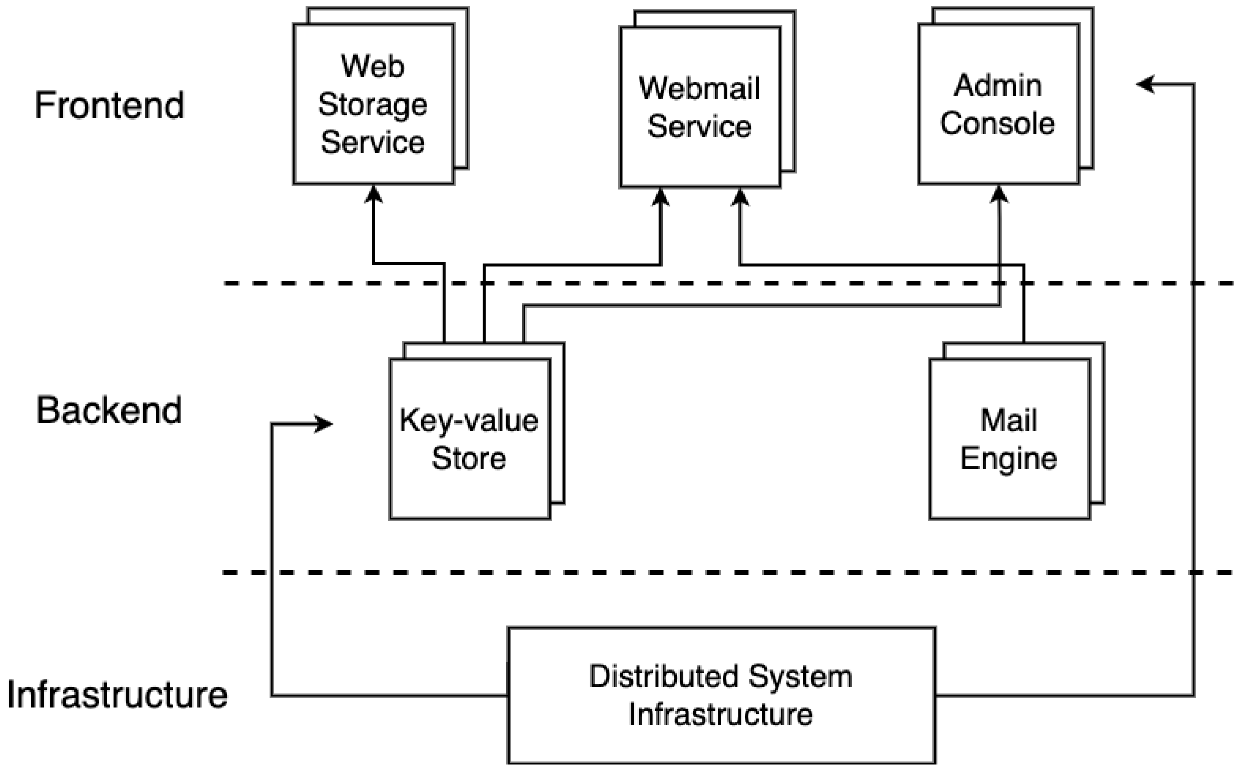
PennCloud Proposal - Team T07

Description of Design:



Key-value Store

Design: The core of this component is a bigtable, which is sparse and distributed. It support four operations:
• PUT(r,c,v): Stores a value v in column c of row r
• GET(r,c): Returns the value stored in column c of row r
• CPUT(r,c,v1,v2): Stores value v2 in column c of row r, but only if the current value is v1
• DELETE(r,c): Deletes the value in column c of row r
As a typical abstraction of the bigtable, it will support replicas of data with some level of consistency and operate in scale. We take fault tolerance into consideration and it can efficiently recover from failures.

The row will be each user name, and each row can have many columns such as user's password, user's filenames. Each file can take one column and the value will be its data blob. Each email can also take one column and the content will be its corresponding value. Each folder can take one column and the value will be a list of file or folder names in that folder. This way we can have nested folders.

Web Storage Service (Frontend)

Files storage of key-value stores will be displayed as the storage service in the frontend. The generated HTML will contain <input type="file"> element for file uploading. We will implement some high level functions to create, rename, and delete folders or files.

The root url for each user will be www.penncloud.com/username. File storage will use www.penncloud.com/username/files followed by each file's path, like www.penncloud.com/username/files/outermostFolder/innerFolder/fileName.

Mail Engine

We will adapt our HW2 solutions to serve as a mail server for our system. We will adapt our mailbox system to use the KV store implemented above instead of using a local .mbox file. In addition, we will adapt our solutions to send mail from other addresses such as Gmail or Outlook.

Webmail Service (Frontend)

The frontend user interface for webmail would be a simple web page with view, edit, send, and delete functionalities.

Distributed System Infrastructure

Modules we have mentioned above are **applications** executed in multiple server nodes in a distributed fashion. We need to build the **infrastructure** at the very basic level to power them, and provide generic templates to make the implementation more efficient.

There are two major modules that we will implement for this infrastructure component. The first is the implementation for each single server node that we will execute each frontend and backend application instance. We will provide task management & scheduling, generic multithreading/event-driven interfaces to customize the function based on requirements. The second module, which is the most important, is the coordination and communication between different server nodes to realize this robust, consistent and efficient distributed system. We will consider the most important design decisions including Replicas, Consistency, Fault Tolerance, and Load Balancing. The choices of models and strategies will be described in detail in the next section.

Admin Console (Frontend)

This is the admin page where we can monitor the status of the system and manage different nodes. This application is run on a single node but will retrieve state from each node. Correspondingly, for each node, it will provide the interface to timely report status and manage status based on the instructions from the admin page. These will be implemented at the infrastructure level. Specific functionalities include:

- Monitor all the nodes, either frontend and backend, and track their status
- View raw data in storage
- Disable and restart individual storage nodes
- Shutdown or restart the entire system

Design decisions

Purpose + Model

- Replicas

1. Reason:
    a. Durability: Once data is stored, the system should not lose it.
    b. Availability Data should be accessible whenever we need it.
    c. Speed: Accessing the data should be fast.
2. Methods:
    a. We will use Quorum-based protocols because we prefer fully distributed replications to avoid single point of failure and bottleneck.

- Consistency

1. Models:
    a. Sequential consistency: The strictest model as if all clients 'see' operations are executed in the exactly same order.
    b. Casual consistency: Related operations must be seen in the same order by all the nodes.
    c. Client-centric Consistency: Eventually all replicas should converge to the same order. We can specify what individual clients can see including Monotonic reads, Monotonic writes, Read your writes, and Writes follow reads.
2. Our choice: We think Casual consistency fits our user case best.
    a. Though sequential consistency is nice for programmer but it requires heavyweight mechanisms (like Paxos), may include bottlenecks or single point of failure, and limits concurrency and scalability.
    b. Causal consistency is weeker than sequential consistency but makes sure that all related operations are seen in the same order.
    c. The client-centric consistency only puts requirements from the clients' view. Our backend is complicated and includes many different components. It's better to

think beyond only clients’ view. Hence, client-centirc consistency is too week in this scenario.

- Fault Tolerance
    1. There are many types of faults including Crash faults, Rational behavior, and Byzantine faults. However, in this project, we pay special attention to the crash faults which means that if even one frontend server fail, it doesn’t affect clients much. In this scenario, we should store all the values and states into Key-Value store and cashe the data at the server.
- Load Balancer
    1. We will build a special web service which is a single node that will accept all the incoming requests and takes round robin to relay each request to different node such that the workload of each node is roughly comparable.

Milestones and Collaboration Plan

MS1: KV Store
MS2: HTTP server
MS3: Webmail server
MS4: UI

The most basic component in our system is the key-value store. All of our other systems depend on the k-v store. We will strive to have a minimal version of the system available as soon as possible.
The webmail system and http server depends on the kv store. However, they can be implemented in parallel as they mostly do not depend on each other.
All of our user interfaces depend on successful implementation of all servers. Thus it should be done last.

Division of Labor

Task	Deadline	Yao	Shidong	Zhouyang	Jiaxuan
KVStore Server	11.21	✅			✅
Webmail Server	12.6			✅	✅
General server	12.6	✅	✅		
Account page	12.15	✅			

Storage UI	12.15		✅		✅
Webmail UI	12.15			✅	✅
Admin console	12.15	✅			