DENDRITE ARCHITECTURE

Dendrite is a matrix home server similar to synapse. The whole idea of Dendrite's development is speed. This is a log based architecture in order to communicate between components. The best part being that each component can run separately in different machines .Here is a list of components used in dendrite: Federation Receiver, Room Server , Client sync , Client API, Typing server, Presence Server, Receipts Server, Client Typing setter.

Dendrite's Design/Architecture:

The concept used while designing Dendrite is **Decomposition and Decoupling**: (As mentioned earlier, matrix is basically component based system, where each component can run separately in different machines) or in short we can say each component in dendrite are not glued together. Matrix home server is built around append-only event logs built from messages, receipts, presence etc.

The sever is decomposable and can be categorized into two types : a) Readers , b) Writers

a) Writers: It adds to the append – only logs.
b) Readers : It reads from those logs.

**The important part on readers and writers is that they both has to agree on the format of appends done in the logs.*

However, the way matrix-dendrite handles states events, create some kind of complications in the model:

a) The writers will require the room state at an event to check if writing is allowed or not.
b) The readers too will require the room state at an event to determine the users and servers that are allowed to see the event.
c) The client can query the current state of the room from a reader.

The writers and readers cannot extract the necessary information directly from the event logs because it would take too long to extract the information as the state is built up by collecting individual state events from the event history.

Hence the readers and writers need access to something that can store copies of event states that can be efficiently queried. One possibility for this is to maintain a local database where both readers and writers can store copies of state events. Another possibility will be to maintain

a dedicated component to store these events and expose an API which can be queried by readers and writers. The idea of dedicated component is possibly the best because it stores every event in a single location.

**The room servers can annotate the events it logs to the readers with room state so that the readers do not have to query the room server unnecessarily.*

Component by Component functionalities:

Let the Local client send a PUT request to an HTTP loadbalancer,the loadbalancer actually routes the request to a ClientAPI.

ClientAPI:

- Authenticates the local client with the access_tokens sent in HTTP request.
- Checks if it has already processed a request or is processing a request with same transactionID.
- Queries the necessary state events and latest events in the room from the room server.
- A confirmation is done if the room exists and check whether if the event is allowed by authentication checks.
- Writes the event to InputRoomEvent ,kafka topic.
- Sends 200 ok response to client.

Room server reads from InputRoomEvent Kafka topic:

- Checks if it has already has a copy of the event.
- Checks if the event is allowed by the auth checks using the auth events at the event.
- Works out what the latest events in the room after processing this event are.
- Calculate how the changes in the latest events affect the current state of the room.
- Writes the event along with the changes in current state to an "OutputRoomEvent" kafka topic. It writes all the events for a room to the same kafka partition.

The ClientSync reads the event from the "OutputRoomEvent" kafka topic:

- Updates the copy of current state in the room.
- Works out which users has to be notified about the event.
- Wakes up pending requests to users.
- Adds the events to the recent timeline of events in the rooms.

The FederationSender reads the event from the "OutputRoomEvent" kafka topic:

- Wakes up a pending event to servers.
- If the request is in progress then add the event to the queue to be sent,when the previous request finishes.

Remote server sends an event in an existing room:

- The remote server sends a PUT request and the HTTP loadbalancer routes the requests to a Federation Receiver.

FederationReceiver:

- Authenticates the remote server using “XMatrix” authorisation header.
- Checks if the state event is already being processed or is under processing with the same transactionID.
- Checks the signature of the event.
- Queries the roomserver for a copy of the state of the room at each event.
- If the roomserver does not know the state of the event then query the state of the room at the event from the remote server using GET requests.
- Once the state at each event is known check whether the events are allowed by the auth checks against the state at each event.
- For each event that is allowed write the event to the "InputRoomEvent" kafka topic.
- Send a 200 OK response to the remote server listing which events were successfully processed and which events failed