**Web Service Interface-**

This Backend application is a part of an overall temperature monitoring solution. Various temperature

sensors send real time temperature updates to server. There are http clients which are

interested in continuously monitoring the temperature and they subscribe for it with the server.

Whenever server receives an update, it publishes the updates to all interested clients.

The below diagram gives a clear picture of the backend flow:

**Subscribed Client**

**Sensor Client**

**Central Server**

**Important Note: Here, we are using the socket.io(along with node.js) library for realtime communication between clients and servers. For more information you can check out the website:** [**https://socket.io/**](https://socket.io/)

**Connecting to the Web Services Endpoint-**

The various Web services to connect the sensor clients and the subscribed clients to the central server has the following endpoints:

1. The central server is listening on the following http url:

**http://<ip-address> or <hostname>:<port>**

For example, it will be: **http://localhost:6002/**

Any client(sensors clients and already subscribed clients can use the above URL to connect to the central server via a websocket connection)

1. The central server is providing a http **POST** webhook API for interested clients to register with the server by using the following API and JSON body:

**http://<ip-address> or <hostname>:<port>/api/registration**

The following information has to be provided in the JSON body for the POST request:

* **Client\_Token**: Client Token or client API key. This will be used by the server to register the client and later will be used to verify the handshake, then acknowledge and send the relevant data back.

\*You can also check the Future scope section below for more information.

* **SID\_List**: System ID List for which temperature alerts needs to be monitored. Please note that the following SID format needs to be used-

**REF->Refrigerators, ACR-> Air Coolers, ROM-> Room**(currently we are using only 3 types of SID). Note that this has to be sent in the form of an array/list.

* **SID\_Count**: The total SID count for the number of SIDs for which temperature alerts need to be monitored.

**Please note that there is a check in the server to see whether the SID\_List matches with the SID\_List length or not and it will throw an ERROR message in case it does not match**.

For example, if we need temperature data for 2 systems, let’s assume the SIDs are ACR01 and REF02 then following POST API and JSON needs to be used from the subscribed client to register with the central server:

* API: [**http://localhost:6002/api/registration**](http://localhost:6002/api/registration)
* JSON body:
* **{“Client\_Token:”CLIENT1”,"SID\_List":["REF01","ACR01"], "SID\_Count":2}**

The following status code can be used to identify from the client side whether the POST request was successful or not:

* Status Code: **200/201** is successful. Status code **400** means connection failed and the client would receive an ERROR message. Status **500** means internal server error which could mean that there is some error in the POST request that is being sent to the server and will also show a similar error message as status code 400.

1. The sensor clients as of now are generating random temperatures within a specified range and a bi-directional connection can be opened between the sensor clients and the server using the following url and socket eventname:

* URL: **http://<ip-address> or <hostname>:<port>**
* socket eventname: **sensor**

For example, Url: **http://localhost:6002/** .

Let’s assume, data to be sent is dataJSON:{};

socket.emit(“sensor”,dataJSON) . (Here we are assuming socket is the variable used for the socket client)

Sample JSON Data from the Sensor Client to the server:

**{"Timestamp": "2020-07-22 08:48:36",**

**"metrics": [ {"SID1": "REF01","Temperature": 8},**

**{"SID2": "ACR01","Temperature": 28},**

**{"SID3": "ROM01","Temperature": 27}]}**

Socket used

**Central Server**

**Sensor Client**

\*\*You can also check the Future scope section below for more information.

**Note**: In case of server disconnection, the sensor clients keep on generating random temperature values every 5 seconds in the JSON format, although it shows the ERROR message too that it failed to connect to the server. When connection is re-established, all the last JSON data generated is sent to the server in one shot.

Since, Node.js uses the V8 runtime engine, the socket.io emit event keeps on emitting data and the data is buffered using the inbuilt nagle algorithm of node.js by default. The maximum storage value of the buffered data is approximately around 1GB for 32-bit systems and 2GB for 64-bit systems.

Also, the maximum length of the JSON(message) that can be sent at a single time cannot exceed more than 300MB otherwise you might get the error- “RangeError: Invalid length” or “RangeError: Maximum call stack size exceeded”.

1. The interested clients can subscribe using the API and JSON body from point number 2.

Once they register successfully, they would get back a unique token from the server which is used by the client to open a socket tunnel with the server for real-time data.

After registration the subscribed clients need to send the SERVER and CLIENT token pair to the central server for further verification and to get the relevant alerts.

They can use the following way to open bi-directional connection between server and client for real time data:

* socket eventname: <unique identifier from server>
* socket data/string/arguments: <unique key value from the JSON body in point number 2>

For example, if we send the following JSON from subscription:

**{“Client\_Token”:”CLIENT1”,"SID\_List":["REF1","ACR01"], "SID\_Count":2}**

And after successful subscription we get the following unique identifier from the server: **“106”**

Then we would do the following for getting alerts:

socket.emit(”106”,”CLIENT1”) .(Here we are assuming socket is the variable used for the socket client)

HTTP/Socket

**Subscribed Client**

**Central Server**

1. The output(monitoring alerts for temperature) for the subscribed clients would be in JSON object format and would look like the following:

**{<SID>:{“Temperature”: <Temperature>,”Timestamp”:<Timestamp>}}**

For example, let assume we have a subscribed client who wants the temperature for REF01 and ACR02 SIDs, then the output to the relevant interested client would be:

**{"REF01":{"Temperature":0,"Timestamp":"2020-07-20 19:01:23"},"ACR02":{"Temperature":20,"Timestamp":"2020-07-20 19:01:23"}}**

Kindly note if in the above example, due to some issue if ACR02 data is not available, then the subscribed client would still get the data for just REF01 in the following way:

**{"REF01":{"Temperature":0,"Timestamp":"2020-07-20}}**

Also, since we have unique CLIENT token for each client mapped to the server, multiple subscribed clients will get the information at the same time and the server also keeps a track of what data is being sent to which client at any particular instance.

**Providing Credentials-**

Currently there are no credentials maintained for the socket connections or the http connection. Since real time data is required, we deal the authentication mechanism using SERVER token and CLIENT token pair between clients and the server. If required, we can have authentication measures such as Basic or Digest Authentication in the future.

**Future Scope-**

* \*\*We will also create handshake mechanism to connect multiple sensor clients to the server at the same time instead of using “sensor” eventname, rather we will establish a unique key transfer mechanism similar to server-subscribed client. This would change the point number 3 going forward.
* \*The client API key will be generated in the respective clients going forward. As of now every interested client will provide a unique CLIENT token on their own to the central server during subscription. In response the server would send a unique SERVER token(random generated value by the server) to the subscribed clients.
* Currently every client has a connect error feature to determine whether the server is reachable or not, if server is not reachable, the client shows an ERROR message and tries to reconnect to the server via the websocket. In the near future as per requirement we can also scale up the disconnect, reconnect features and like the handshake mechanism these features would also be automatically taken care in the backend.

**N.B**.: Please note that in the above guide, the sensory client(s), central server and the subscribed client(s) are assumed to be present in the same OS. Hence, in the place of **<ip-address or hostname>** we have used **localhost** and we are also using **6002** as the port here. For our demo purpose, we are using a constant interval of **5000ms**(5 seconds) for transmission of data between clients and the server. The codebase also contains a **config.json** file to store fields such as ip, port, frequency so that these values can be changed dynamically based on the environment and requirement. The Timestamp values in sensor client(s) are converted to **UTC** format to maintain time zone consistency throughout any region.