COMPSCI 677 Lab 3 Design Document

Team Members
Chandra Sekhar Mummidi
Sai Venkata Maneesh Tipirineni

Overview

The overall design proceeds in three phases: 1. Server Replication and Load Balancing, 2. Consistency and Caching, 3. Fault Tolerance. The detailed descriptions of these phases are described below. Thread Synchronization was achieved through Python GIL.

Server Replication and Load Balancing

When a device or sensor comes online, it contacts both servers and gets the number of devices connected to each server at that time. It connects to the server with lesser number of devices connected. If both servers have same number of connected devices, the device connects to the first server. By connecting to the server with lesser connections (lesser load), load balancing is achieved. Each server gives a unique device id to each device/sensor. To ensure that the device id's stay unique for each device, one server only gives odd device ids and the other gives only even device ids.

Consistency

We used strict consistency between the servers. When a server registers a device or gets an event message i.e., a state change notification or polling data, it updates its database through its backend and sends the data to the other server which then updates its database too on receipt of data from another server. Both servers use different database files, so file access is independent and shared data access handling is not needed for our design.

Cache

Cache stores all the latest events. We implemented an LRU cache. Cache store all state change events, polling events and events received from other server too to make sure that they are exact replicas of each other. The data from the cache is used by our security system algorithm to make decisions. This is another reason why maintaining the events received from other server in the cache is a good idea, we need to store data that is useful for making decisions and having device event data received from other servers helps. If we do not store this data, we have to access the database to receive event data of devices not connected to this server. Also our cache states at a server are always consistent with the database because data to the database goes from the gateway and the backends are not connected to each other. When events pass through the gateway, they are stored in the cache and then sent to the backend and hence, all events are recorded in the cache. No event occurs in the database which can cause the cache to become outdated. The max number of items stored in the cache has to be configured in the bash file run.sh. For the extra credit part, all servers are assumed to have the same cache size for simplicity.

Fault Tolerance

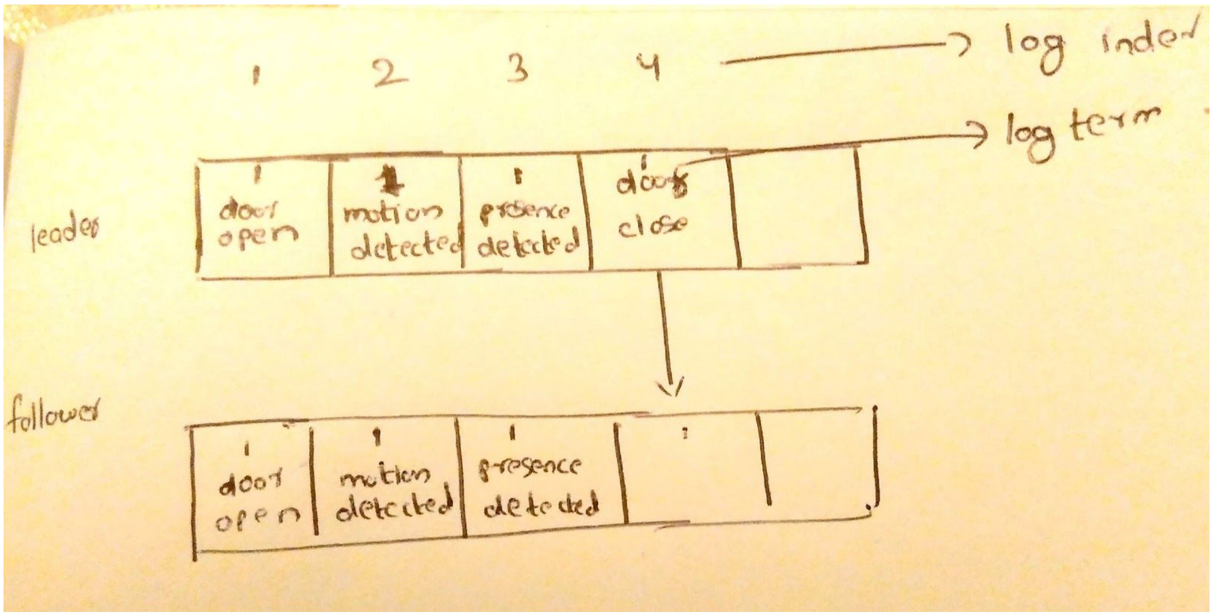
Every gateway keeps track of all the devices in the network and all the connected devices. Every gateway has information about which device/sensor is connected to which server. The gateway keeps polling the other gateway for heartbeat messages. If a communication error occurs (when the message is not received) while polling for heartbeat, failure recovery is started. The gateway which detects the failure already has information of the devices connected to the failed server. This gateway sends a change_server message to those devices/sensors and they update their connections to connect to this server. Data is lost when the failed replica stores some information in its database but fails before sending this information to the other server. The impact of the lost data depends on the type of the event lost, if the event is a polling, it might not have as much of an impact as if it were a state change. To counter this, the other server polls and records the current state of the device/sensor in the database after it sends the change_server message.

Design for Consensus (RAFT)

Consensus Technique is needed to implement the state machine shown in Figure 1 on replicated servers. Using consensus techniques, for the system to work, it is sufficient for majority of servers function even if some of the servers fail. RAFT consensus algorithm is used in our design. A log containing requests that are to be processed by server is maintained at each server replica. Each request is entered into log based on a order agreed by majority of replicas in a voting. Server will process the request in same order they are available in it's log. The design we followed to implement RAFT consensus algorithm-

Leader Election - Leader is elected among server replicas using bully algorithm. A request will be only be entered into log once leader made an election on the request. Leader will send heartbeat to other server replicas and replicas stop receiving heartbeat because of leader failure, they initiate leader election again. After each leader election, server node that is elected as leader will send reregister message to each sensor and devices. In bully algorithm, log term and log index are used to elect leader. Leader with highest log term and if log terms are equal then log index is used to decide leader. In the startup of the system, device ids are used to elect leader because each node will have same log index and log term.

Log Replication - Every client registers with only leader server node. Client sends request to leader and the request will go to request queue of leader node. When leader node is free it will take the first request in the queue and conducts an voting for the request if it receives majority vote from other server replicas leader will commit the request that is adds the request to the commit log. Commit log has to implemented on hard disk because if the node crashes and recovers then the commit log has to taken from external disk to continue. This is achieved by storing the logs in a text file called 'latest_event$1.txt' where $1 in our design represents id of the server node replica. It also sends commit the request message to follower nodes which makes the follower nodes to commit the request and state machine of each server replica process the commit request.



For the state machine shown in figure 1, first every server node will start idle state with empty logs and match index and next index are same equal to -1. First consider a path in the state diagram from idle to door ‘open’ state. This request is added into the request log queue and next index is incremented each time a request is added into the queue. Elected leader will conduct an election with all the other (k-1) server nodes and commits this request if it received a successful vote. While doing the election if any other request arises like after door open detection, motion detection and presence sensor will be stored in the queue with nextindex incrementing for each time. While election leader will ask other nodes to commit this door open state and this will increase the match index after successful voting. Next, it will take motion detection from request queue and makes an election, upon which every server node will move to state 2 that is someone entered the room. Presence key signal vote will move the server node systems state to intruder alert or user home. If user home is the state, leader will send response to client i.e by switching on lights and smart outlets. If intruder is detected, leader will send to client by switching on lights and switching off smart outlets.

Safety and consistency after leader changes-
This can be achieved by applying restrictions on leader election and request commitment.
Restriction on leader election-
During Server leader election, leader with highest index and term should be elected as leader. This is achieved in our design by using bully algorithm and electing leader based on the index and term values.
Restriction on request commitment-
This is achieved by committing the requests only when majority of server nodes have atleast one commit for leader term .

Server_Gateway

This is the application tier which interacts with sensors, devices, database and makes security decisions. It also registers the devices and gives them unique device id’s and also

polls temperature and door sensors regularly. It also receives state_change messages from presence, door and motion sensors and sends these event data to other gateways and maintains LRU cache for events.

Server_Backend

This is the tier which manages the database. We have 3 database files. devices.txt which contains information about the devices and sensors. events.txt which is a log of all the events that occurred and latest_events.txt which contains information about the latest events, the ones required to make security decisions. Splitting the events database into two files helps us differentiate the older events from the latest ones and determine if we need to wait for a new event to happen to decide if the "user left home" or "user entered home". In both those cases, we need both doorsensor data and motionsensor data and the ordering between them. Suppose the user left home and now is coming back and we do not have a latest_event database, we see old motionsensor data in the database from when the user was home earlier and doorsensor trigger when user opens the door and move state to "user entered home". This is wrong as the user just opened the door and is not home yet, we still need to wait for the motionsensor trigger before declaring that the user is home. Having a latest_events database and clearing it after a state change makes it easier to accomplish this. Every write to a database file is counted as an event of server_backend. Note that polling will be considered as an event at the backend as the state is written to events.txt

The schema of our tables is as follows:
Devices.txt: Device_unique_id, Device_type, Device_name (separated by tabs)
Events.txt: Device_id, Event_description, Device_physical_time, Device_logical_time
Latest_events.txt: Device_id, Event_description, Device_physical_time, Device_logical_time

Devices and Sensors

Devices register themselves with the server_gateway when they come alive. Their names and ids are stored both in devices.txt and also in-memory for faster access and decreasing network traffic. Each device also knows the name of the leader once leader election is completed. Each device also gets the list of devices during the lead_election process.
4 sensors were implemented with their respective push/pull/change functionalities.

Security System Algorithm and Event Ordering

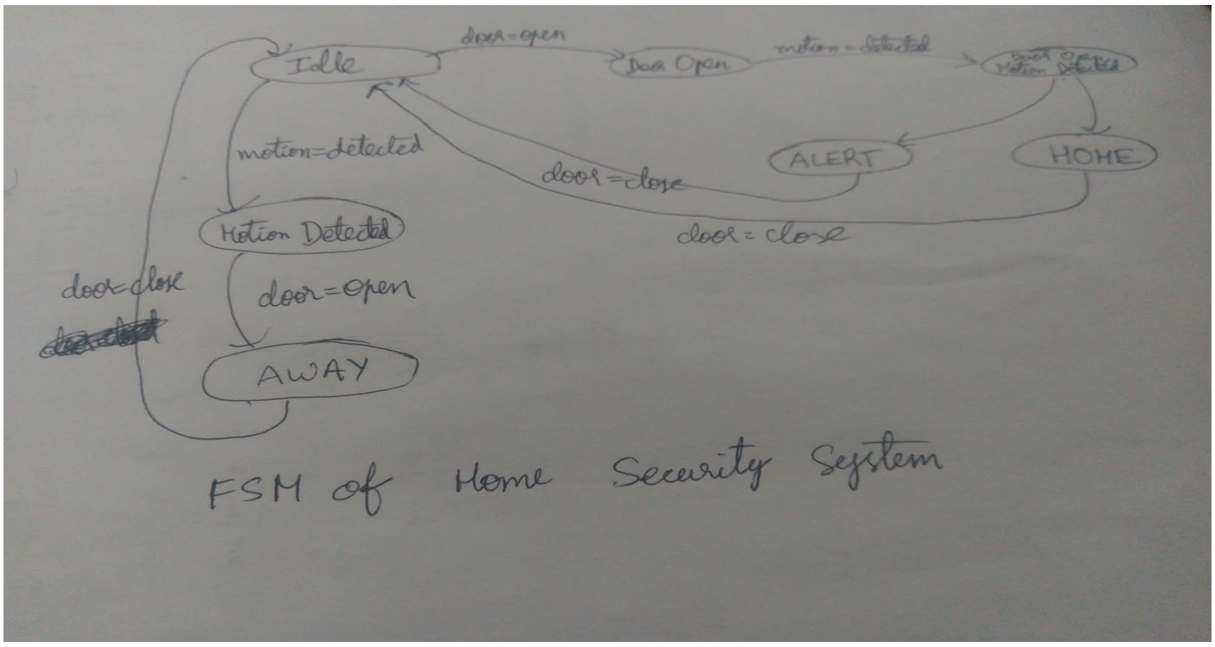


Figure 1:Base State Diagram of our Security System algorithm

We are using the above state diagram and physical time from the cache and latest_events.txt to build our home security system algorithm and determine whether the user is Home, Away or ALERT in case of a burglar entering the house. The order in which the transitions occur in the above figure is determined by event ordering.
We used numbers for states in the implementation of our algorithm. The following table maps those state numbers in our implementation to the above states.

0	Idle
1	Door open
2	Door open and motion detected
3	User Home(presence key detected)
4	Intruder Alert(presence key not detected)
5	Motion detected
6	Door open (user away)

While implementing event ordering in code, after every message received, server gateway checks for recent event for each sensor, door sensor and motion sensor which are push based.
Physical time is retrieved from the cache or the database. For example, physical times for motion detection and door opening are used to make decision about whether user entering into home or leaving home. This data is retrieved from latest entries of each sensor. If an event is used to make decision then that event is removed from latest event entries and if server doesn't have enough data to make a decision it will wait for message to reach it.

If the user comes home, both the light bulb and the smart outlet are switched on. If the user leaves, both are switched off. In the event of an intruder alert, the lights are switched on but the smart outlet is switched off.

Design Tradeoffs

1. Only one instance of each device/sensor can be run.
2. We assumed that the door is closed by the user while leaving or entering home.
3. Message exchange between servers for polling data increases network traffic.
4. Cannot recover from failures where server crashes while devices are in the process of registration.

Further Improvements

Restarting servers after detecting their failures would be an improvement. One more improvement could be dynamically setting the cache size based on network load (number of events arriving). If the network load is less, it makes sense to decrease the size of the cache and save space while if it high, we would prefer a bigger cache size to decrease the number of misses. Another obvious improvement would be to allow two or more instances of the same device/sensor run simultaneously. And another could be adding support for dynamically adding and removing sensors and devices during runtime.

Note: Master branch contains the code for basic lab requirements (without the extra credit part). The extra credit part and all the lab requirements can be found in the **extracredit** branch.