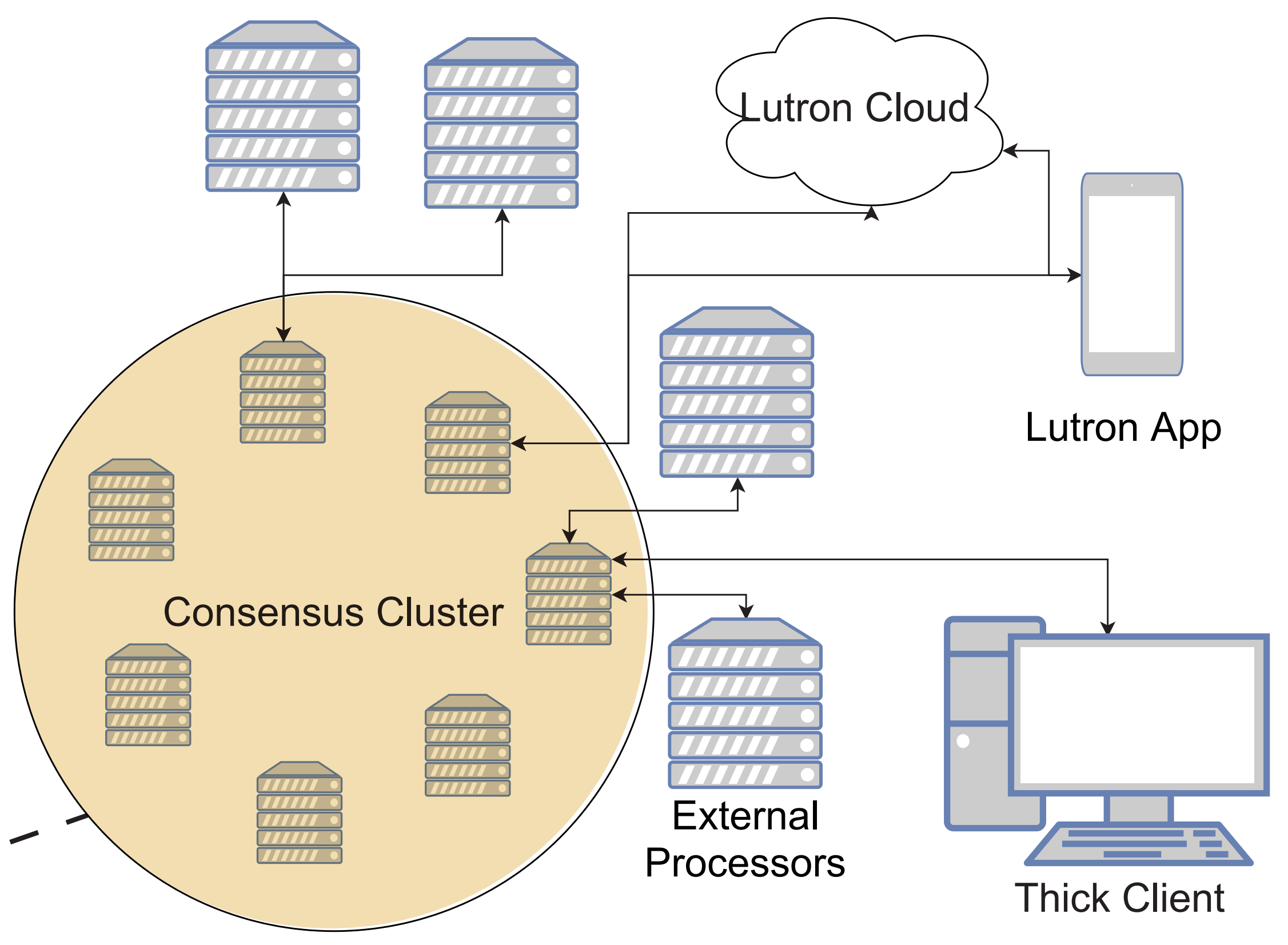
DISTRIBUTED CONSENSUS FOR LARGE SCALE IoT SYSTEMS

TAL DEREI, MARC FERDINANDUSSE, HAILEY GOLDSCHMIDT, JACOB OAKMAN
ADVISOR: BRIAN COLVILLE, SPONSOR: LUTRON ELECTRONICS COMPANY

CSB / CSE Project 2021

Abstract

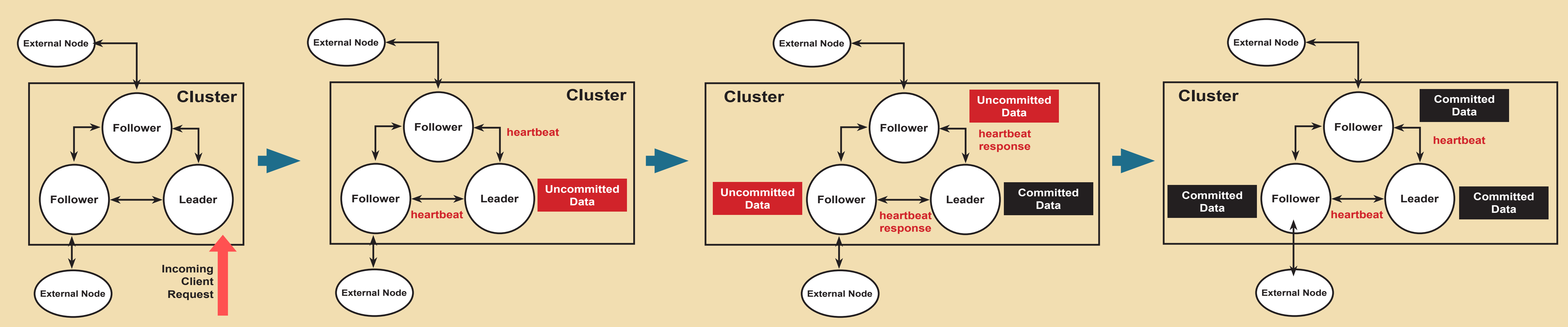
Lutron's smart lighting systems consist of a network of controller boards which drive lighting devices. These boards need to agree on the configuration of the system to provide a unified experience and work even when part of the system is offline. Currently this is achieved through a brute force approach which does not scale effectively. Our project was to research methods to solve this problem scalably, implement a prototype using the researched methods and test this prototype to determine whether our solution was successful. Our research led us to the Raft Consensus Algorithm. Our prototype is built on top of an implementation of Raft, written in golang, developed for the distributed data store etcd. We then developed an automated test platform which measures the performance of our prototype running on a network of AWS instances. From this we identified that our system currently has an unbounded growth in memory usage which we are working to remedy. Other than this issue, our approach is promising and has been shown to scale effectively.



Motivation

Lutron builds distributed IoT lighting systems. These systems need to share important configuration information. Lutron's current solution can only support a few nodes in a system, maxing out at around 16 nodes. Implementing a system which can handle a much larger scale could lead directly to a better experience for customers and Lutron's ability to support bigger systems.

Solution



Raft Cluster receives request from a client.

Data is appended to log but not committed. The request is sent to the follower nodes on the next heartbeat.

The follower nodes append the data to their logs but do not commit the data. They send a response to the leader. The leader commits the data.

On the next heartbeat, the follower nodes commit the data in their logs.

Raft Consensus allows us to coordinate committing data across processsors efficiently even if some of the network is down.

System Architecture

In our system, there are two types of nodes: cluster and external. Cluster nodes are part of the raft consensus cluster and handle requests from the client. External nodes do not participate in raft consensus. Instead, they query for specific data from the raft cluster nodes. We added objects that represent the data in the client requests. The objects are sent between nodes with a REST API. When the cluster reaches a consensus on the configuration data, it is persisted in each of the nodes' local storage. Each cluster node can also be queried for its peers as well as a copy of its local database.

1 Research

Lutron provided an analysis of the open source distributed data store, Etcd, finding that it provided the desired functionality but was not optimized for embedded systems. This led us to center our project around building a lightweight system around a consensus algorithm. Paxos and Raft are two such algorithms that we investigated as potential options and we eventually settled on Raft because it is more understandable than Paxos and has been shown to be equivalent.

2 Prototyping

Overview

Etcd provides a robust and lightweight implementation of Raft which can be accessed without importing the entire system. Our prototype uses this implementation for consensus, provides a REST API for communication and uses sqlite3 to persist the data. This means that when a change needs to be made to the database, external clients will send requests to our API, which will then propose those requests using Raft, which will finally allow us to commit those changes to the database which can be accessed for reading directly.

Challenges and Solutions

The examples for how to use Etcd's Raft implementation showed an example of a key-value store which which was not backed up by a database. We were able to extend and adapt from this example to produce an object store with custom object types representing what would exist in a Lutron system and linking them to a sqlite3 relational database where they would be saved once consensus took place. Additionally we needed a way to communicate with the nodes in the cluster so originally we developed a simple command line interface which we eventually replaced with a REST API. Another challenge we ran into was identifying bugs in our codebase. We solved this by implementing automated unit testing in our repository.

3 Testing

Overview

We used AWS to represent the functionality in a real system by creating instances with similar specifications to Lutron's processors. These instances are 64-bit ARM machines based on the AWS t4g.nano profile, configured with a single CPU core and 0.5GB of RAM. Any number of instances can be spun up to stress test various different system configurations and workloads. We also needed to create a basic client to simulate different types of request workloads to exercise our system. This client allowed us to simulate constant and bursty loads, the latter being more realistic to a Lutron system.

Challenges & Solutions

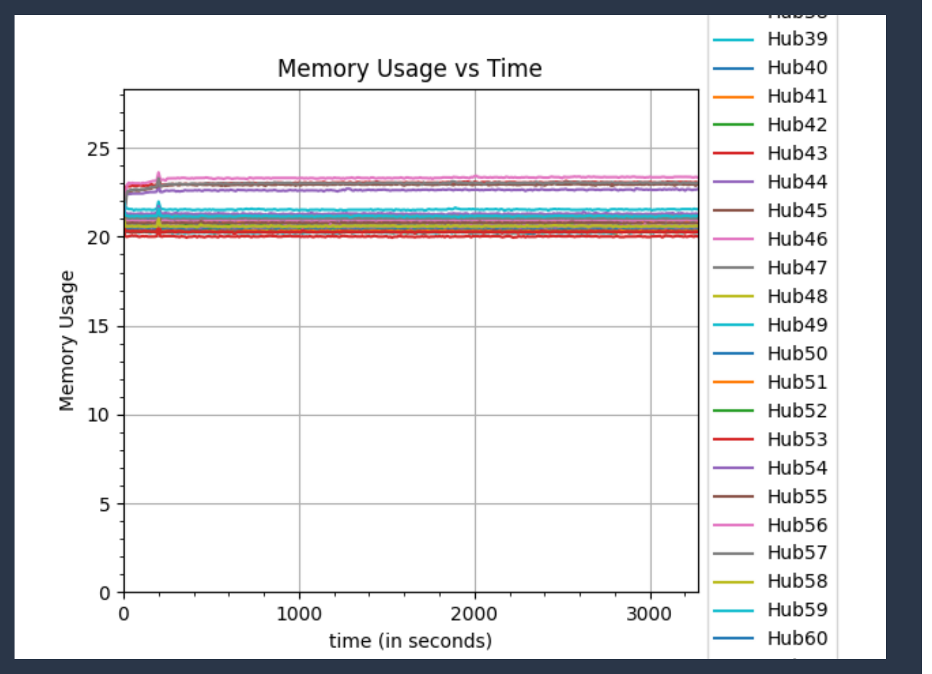
Testing at scale was originally incredibly time consuming because it was conducted manually. We remedied this by developing an automated testing framework which automatically coordinates running tests with different workloads, monitors performance and plots that performance into graphs. This culminated in shrinking work that initially took several weeks into a single command which produces fully plotted data in little more than the duration of the test itself. This automated testing framework was integral to identifying flaws in our prototype.

Results

Out test cases included idle and burst workloads on 100 node systems. Idle is the state of a cluster recieving no requests while burst is when the cluster periodically recieves several requests at a time.

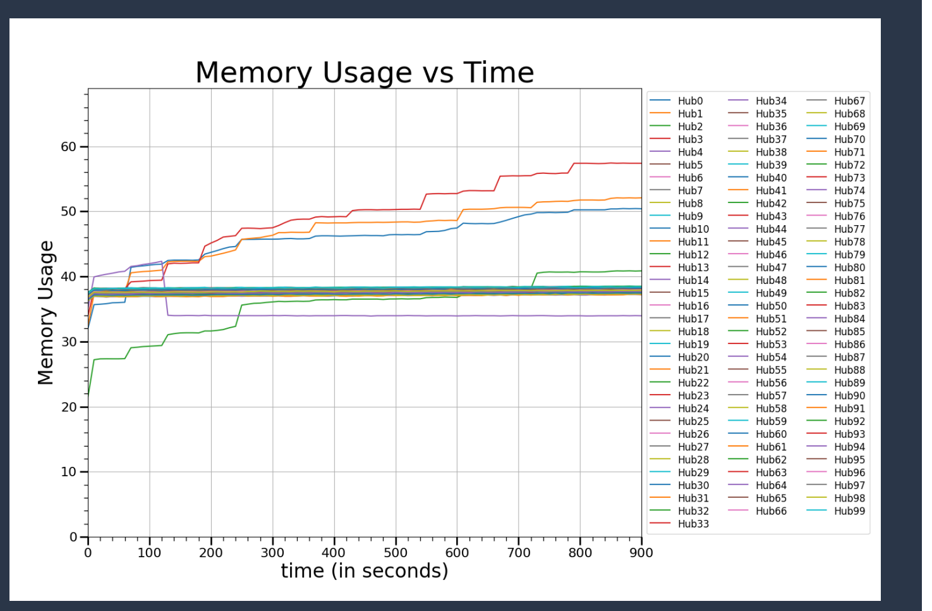
Idle Test

The 5-node cluster (16 nodes) reports slightly higher averages, but still maintains 20-30% averages on 100 nodes (nearly 50-60% reduction), compared to Lutron's 40-70% memory on 16 nodes.



Burst Test

Burst Results indicated memory utilization between 20-60% on average. The bunch of lines hovering around 40% represent the client nodes



Executable Size

Our lightweight implementation produced a 30% smaller executable than Etcd which is important when working with memory constrained embedded systems.

Conclusion

Our prototype and testing shows promise for a highly scalable solution to Lutron's problem. However, there is currently a flaw in our implementation which is leading to growth in memory usage over time which has not yet been resolved. We believe this isn't a problem inherent to the system but merely a bug in the implementation. Once fixed, we should see stable performance even up to 100 node systems which is nearly an order of magnitude greater than Lutron's current solution. While our prototype isn't perfect, it was successful in demonstrating that a lightweight, raft-based system can provide a scalable backbone for Lutron's next generation IoT systems.

Next Steps: Snapshotting

Relying on our sqlite database as a living snapshot of the raft logs would reduce both memory and storage usage compared to our implementation.

Next Steps: Fault Tolerance

Dynamically adding and removing nodes from the raft cluster when processors go offline could provide more fault tolerance without increasing the cluster size.

LEHIGH UNIVERSITY

LUTRON®