EdiSense: A Replicated Datastore for IoT Data

James Hong, Johnathan Raymond and Joel Shackelford
{ jamesh93 , jdraymon , jshack02 } at stanford.edu

Abstract— **We present EdiSense, a miniature datastore designed to service data from low power embedded IoT sensors. EdiSense runs on Intel Edison hardware and serves as a framework for replicating sensor data across a distributed network of nodes. We aim to provide fault tolerance and data availablility in embedded sensor networks. EdiSense is optimized for write availability and lookup with a zero hop DHT. To achieve consistency over the DHT, we implement a protocol for load rebalancing requiring initial agreement from only two nodes, and delayed acknowlegements from the rest of the cluster.**

I. INTRODUCTION

We designed EdiSense to interact with low power wireless sensors acting as Bluetooth Low Energy (BLE) peripherals. These BLE enabled sensors can be extremely low cost to manufacture and deploy. For $25, one can purchase a TI SensorTag development kit which includes built-in sensors for temperature, humidity, pressure, etc... and can run for years on a coin cell battery [1]. However, despite their size and cost advantages, BLE sensors have several disadvantages that prevent them from being able to model more traditional sensor systems. Owing to their low power envelopes, the BLE enabled sensors can be extremely computationally and memory constrained, being able to cache only a few kilobytes of data before running out of memory. While BLE's low power requirements mean that devices can run for long periods on small batteries or on energy harvested from the environment, this can also place limits on range of transmission and on how frequently a sensor can advertise packets. In order to process the data collected on these sensors, it is often neccessary to send the data to the cloud through an intermediary such as a phone. This can be a problem when no internet connected device is present to forward the data or when doing so represents a security risk.

Although pairing BLE enabled sensors with smart phones works well with personal sensors such as fitness trackers, we believe that BLE enabled sensors can also be applied to other static data collection tasks such as building or environment monitoring, including water consumption, air conditioning systems, room occupant tracking, etc. Such BLE wireless sensor networks would be relatively inexpensive and simple to install, requiring few if any changes to existing infrastructure. We do, however, require a reliable means of accessing data on these sensors. This is the problem that EdiSense attempts to address.

EdiSense is designed to run on the Intel Edison IoT development platform. These devices can be as small as a couple of SD cards, while featuring a dual-core Atom processor, 1GB of ram, 4GB of flash (expandable to 128GB with Micro SDXC), and built-in WiFi and BLE radios. The Edisons also run linux and are sufficiently powerful to run EdiSense, yet retain many of the size and power advantages of lower powered BLE only sensors. In our network, the Edisons serve as entry points for data. In addition, EdiSense also supports non-BLE equipped nodes purely for data replication.

II. DESIGN

A. Assumptions

We impose several constraints on the environment in which an EdiSense cluster would operate, including the following:

- EdiSense operates in a trusted environment where the list of members in a cluster is known and provided at initialization.
- New nodes are allowed to join a cluster, but an administrator issues the join and presides over the progress. We also limit the cluster to handling one join or leave at at time to simplify implementation while maintaining consistency.
- When an EdiSense node fails or restarts, it recovers (or is replaced) as it was, due to state information that it checkpoints to disk.
- Data received from sensors can be identified by a 'sensor id' and a timestamp, and does not conflict.

In addition, we assume that although not all sensors (e.g. they could be energy harvesting) in the network need to have access to a reliable power supply, the EdiSense nodes can operate without interruption. The EdiSense nodes can have sensors themselves, though,

more importantly, they should be located strategically to maximize the combined range covered by bluetooth.

B. Goals

1) **Write Availability** Because we assume that sensor data identified by a 'sensor id' and timestamp do not conflict, we write data to the intended replica nodes even when one of the replicas is failed, and the overall Put operation fails. Depending on the failure, the Put can be retried from a different node and successfully committed. Even when rebalancing, we ensure that if none of the intended replicas has failed, that the number of writeable replicas for any partition is equal to the replication factor.
2) **Read Availability** EdiSense inherently provides benefits for read availability for BLE sensor data by making data accessible from anywhere on the network (overcoming BLE range and sensor storage restrictions), allowing access to data even when it is not fully committed/replicated throughout the cluster. Data from a given sensor is also co-located to facillitate and expedite queries against a particular peripheral.
3) **Low Latency** EdiSense supports an expected zero-hop DHT. We bound how out of date a node's cached view of the network can become in the event of a rebalancing/donate operation on a partition. At the same time, to initiate a rebalance and begin servicing writes/reads on the receiving node requires only agreement between and participation of two nodes, the donor and recipient.

C. Interface

EdiSense currently supports two main operations for interacting with sensor data.

1) Put(deviceId, timestamp, expiration, payload): Put is called internally within the EdiSense cluster whenever a node receives data from a sensor. deviceId is a 16-bit identifier that uniquely identifies a sensor. Timestamp is the time recorded by the sensor and expiration is the time until which EdiSense is obligated to retain the data. Payload is an array of raw bytes containing the datapoint from the sensor; its length may vary but is generally limited to 20 bytes due to the payload and MTU limit of using BLE.

2) Get(deviceId, beginTimestamp, endTimestamp): Get is called from any Client application that is not part of the EdiSense cluster. It returns a list of data points from the specified deviceId within the beginTimestamp and endTimestamp range. If the data requested is recent (or the system is experiencing node failures), it may not be fully committed in the cluster; but our goal is to return as much data as possible. Committed data can be requested by contacting a single replica, but to get all available data may require a join across all responsive replicas.

EdiSense does not support an explicit delete operation; instead we introduce 'expiration' as a method for garbage collecting old data. EdiSense is not designed as a permanent data store, but as a temporary store to prevent data loss from sensor failure or exhaustion of local storage on sensors when network access is limited. Our cluster guarantees that data will persist in the cluster at least until the specified expiration time or reaching the limits of storage on the cluster.

D. Architecture

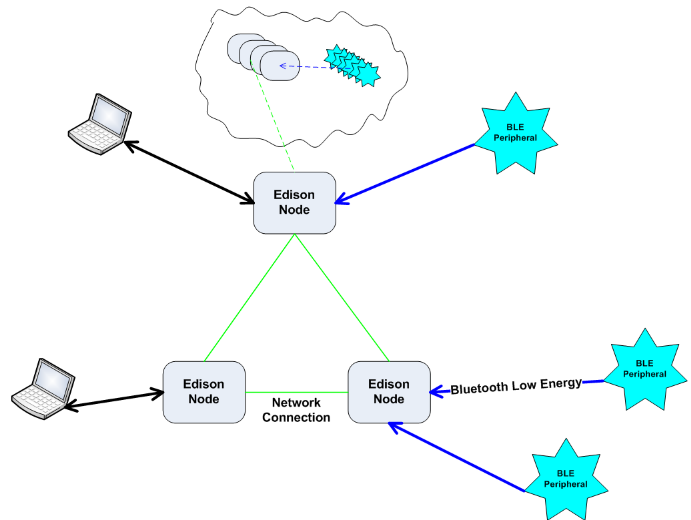

Fig. 1. Example view of deployed EdiSense system

Our EdiSense cluster is asynchronous, decentralized, and fault-tolerant up to the number of replicas. Messaging between nodes occurs with *at least once* semantics on failure of either the sender or receiver.

To handle look-ups for Get and Put operations we implement a distributed hash table based on fixed partitions/buckets, each of which is replicated k times in a k-replica system. As with Amazon's Dynamo [2], this partition system allows us to apply heuristics for replica placement. For each partition/bucket replica, we maintain a database shard at the replica node.

At each EdiSense node, we cache a partition-to-nodes mapping to achieve O(1) hop lookup. All initial

nodes in the cluster receive this mapping at startup. Joining nodes can learn this mapping from existing nodes.

Internally, EdiSense supports Join(), Donate(), and Receive() operations.

III. IMPLEMENTATION

A. Partition Table

Our partition table is implemented as a map from fixed partition/bucket ids to a set of node ids. This is stored as an array in memory, and for a reasonable number of partitions and replicas is efficient to store. We chose this design as opposed to a scheme based on consistent hashing in order to simplify our protocol for synchronizing cached partition tables at each node. Because the ranges covered by each bucket are fixed, it is simple to compute which bucket a deviceId belongs in. Unfortunately, this design limits our ability to rebalance load in the event of hotspots caused by a suboptimal hash function.

Hash values are computed as a function of deviceId, which identifies each sensor. We use deviceId as opposed to a combination of deviceId and timestamp to ensure that data from a single sensor is co-located for more efficient retreival.

At cluster initialization, the partition table is read from a configuration file and all initial nodes start with a consistent view of the partition to nodes mapping. A node that joins later learns the current partition table by contacting all the other nodes in the cluster.

B. Database Shards

Each bucket replica is backed by a database shard, using SQLLite. The databases have the following schema: deviceId, timestamp, expiration, and value. This is to amortize the cost of querying for as single datpoint and to allow for querying of all data for a particular sensor in a given time range. When load balancing is initiated via our donate/receive mechanism, the entire database shard corresponding to the partition being transferred is copied to the receiving node.

C. Put

We consider a Put operation to succeed when all the designated replica nodes have returned success. If the Put operation returns with an error/hint, such as in the case when the partition is in the process of being transferred, the client node initiating the operation retries the Put after updating its cached partiton table. In the absence of node failure at the intended recipients, the second attempt is guaranteed to succeed once the transfer has concluded.

Partial success is supported and preferrable to total failure since our primary goal is to replicate the sensor data, which does not suffer from update conflicts. To avoid new incoming data from becoming permanently backed up behind a failed Put, partially successful Puts are retried asynchronously.

D. Get

The Get operation is called by our Monitor application and any clients of the system. We have two approaches to retreiving data from the cluster.

1) Contact each node responsible for a partition and return immediately once a replica successfully replies.
2) Wait for all replicas to respond up to a certain timeout, and return a join of their results.

Strategy 1 provides lower latency, and guarantees that all data that has been committed is returned. Strategy 2 is more robust under failure conditions that prevent a data from being fully committed. For instance, consider a situation where a node responsible for replicating a partition becomes partitioned from the origin node receiving data from a sensor. In this case, data might not be 'committed' even though other replicas are not partitioned from the origin node. If a monitor were to contact the partitioned node for data using strategy 1, the results could be incomplete, compared to the results that would have been returned by the 2nd strategy. However, We adopt this 1st strategy and accept that data may be delayed due to partitions/failures.

E. Donate/Receive

EdiSense provides a mechanism for load balancing that guarantees that each node's cached view of the partition table remains consistent, within one update for each partition to node entry. In addition to the cached partition table, each node maintains in memory a set of partitions that it is responsible for. Each entry in this set stores the state of a partition on the node: STABLE, RECEIVING, RECEIVED, DONATING. RECEIVING indicates that the partition is in the process of being transferred. DONATING indicates that the node is attempting to transfer ownership of the partition to another node in the cluster. RECEIVED is identical to STABLE, except that the node may not donate the partition.

Protocol:

1) After selecting a partition P to donate, the donor node broadcasts to the cluster asking for ech node's utilization and whether they can accept P. Reasons why P might be rejected are if the candidate receiving node is already a replica of P or if the candidate receiving node is joining/leaving. We call this the CanReceive() phase, and is intended as hint for the donor node in chosing a recipient.
2) The donor node initiates a donation request, contacting a potential recipient with a ConfirmReceive(). In doing so, the donor commits to resending this request for P until it is acked. The selected recipient replies with success or failure, depending on whether it can accept P. If it replies with success, the second node enters the Receiving state for P and begins accepting Puts.
3) If the response is failure, the donation is aborted. If donor node receives a success response, it enters the DONATING state for partition P, stops accepting Put and Get for P, and begins a background transfer of P's database shard. Simultaneously, the donor node broadcasts to the cluster the update to their cached partition tables to reflect the new configuration.
4) When the background transfer of P's database shard completes on the recipient, the recipient marks its state for P as RECEIVED. At this point, the recipient can begin servicing Gets for P as well as Puts, the only restriction being that it cannot donate P.
5) Once all the nodes in the cluster acknowledge the partition table update and the database shard transfer is complete, the donor removes P from the list of partitions that it owns. It then calls CommitAsStable() for P on the recipient until the recipient acknowledges.
6) The recipient, upon receiving a CommitAsStable() request, marks P as STABLE.

From a high level, our donate/receive protocol requires only 2 nodes to participate in the data transfer. However, the donor agrees to remember the donation until it knows that all nodes in the cluster have acknowledged the update. In return, the recipient agrees to wait for the donor's approval for the donotion before attempting to redonate the partition. We persist state updates to disk and log phases to guarantee consistency even on failure at the donor or recipient.

Our system has the advantage of being able to rebalance load quickly. Nodes that are frequently pushing data (sensors can only push data using BLE EdiSense node within range) will quickly discover changes in the partition table (when Put fails). The disadvantage is that for donate/requests to fully complete and partitions to become STABLE and donate-able again, all nodes must acknowledge the change.

F. Join/Leave

To join the cluster, a node sends a Join request to each node in cluster. Each node replies back with success after adding the joining node to their list of cluster members. Once this add completes, the node processing the Join request is obligated to notify the joining node of changes in its partition ownership (on donate). Along with success, each node returns a list of the partition that it owns. We opt for this system because we expect our cluster to be stable.

To leave the cluster, a node must successfully donate all of its partitions. This may take time and is not implemented at the present. For our wireless sensor network, we do not expect cluster downsizing. If a node needs to be replaced, it can be snapshotted and restarted on replacement hardware.

G. Garbage Collection

Garbage collection runs as a separate thread. It serves to reclaim space taken up by expired data. We guarantee that data will be retained for at least until the expiry time, given storage space on the cluster. The GC daemon runs in a delay mode typically of a couple of hours, far longer than any reasonable expectations of clock drift.

H. Monitor Application

Our Monitor application queries data by sensor id from the cluster, using our Get() interface. To do this, the monitor first asks a any EdiSense node for a list of nodes responsible for replicating the sensor's data using Locate(). The monitor caches this and then calls Get() on the appropriate replicas.

Because the Monitor does not 'join' the EdiSense cluster, the cluster nodes are not obligated to notify the monitor when the partition mapping changes due to rebalancing. We handle this case by returning the cause of failure for a Get() request, indicating that the data has been moved and the new owner of the partition replica. The Monitor can resend the request to the new owner, and update its cache accordingly. Since rebalancing of a particular partition is rare, in absence of other failures, Get() operations are expected to succeed the first time.

An alternative approach that we considered for Get() was to proxy the requests through any node in the cluster. Under this approach, the Monitor does not need to consider partiton to nodes mappings directly. The disadvantage is that proxying data requests at the EdiSense nodes can be expensive in memory, time, and bandwidth.

I. Fault Tollerance

For a system configured with k replicas, EdiSense is fault tolerant against k - 1 failures for committed data. We recommend setting up EdiSense nodes such that each BLE sensor is within range of multiple EdiSense nodes, so that if a node fails, the data can still be pushed to another BLE enabled EdiSense node. In the future, hinted handoff is an option so that the data survives up to k - 1 from the time it is received. Upon failure, EdiSense nodes are restarted with the recover flag, where the node's previous state is retreived from disk and pending donate/receive operations are resumed.

For each RPC call that we implement, we have a client and server. On failure of either the client or the server, it is the client's responsibility to reinitiate the operation, spread across the Core, Comms and Monitor modules.

J. Implementation Notes

EdiSense is implemented in c++, and is clonable as a public repository at https://github.com/Edisense. The current implementation is approximately 5,000 lines of code. We implement RPCs and message passing on top of ZeroMQ. We have implented EdiSense to handle multiple requests in parallel, and routine tasks such as initiating donation asynchronously.

IV. Testing

EdiSense compiles and runs on Intel Edison as well as on Stanford Corn machines. We have six Intel Edisons available and have tested Edisense using a mobile phone as a wireless access point. In testing we have focused on correctness. For example, we ran our EdiSense with forced rebalancing every 10 seconds, and then ran diff on the partition tables at each node. This test was performed both with and without fake sensor daemons calling Put(). We have tested EdiSense on Corn with 100 such daemons, and Put requests were able to complete without time outs. Unfortunately, such raw throughput testing does not reflect expected conditions in a deployed wireless sensor network.

We test EdiSense's crash recovery on Corn, by starting a cluster, killing nodes and then restarting them with the recover flag. Since updates to data structures such as the list of cluster members, the list of partitions owned by a node, and the partition table are written atomically to flash before acknowledgements are returned, recovery is simple. Only donations require additional logging, so that they can be resumed followng a crash.

V. Discussion

EdiSense differs from many other systems in that its primary goal is replication, latency and throughput are important to the degree that they do not impair replication. We aim to reduce the window of vulnerability from when a data point is recorded at a sensor to when it safely stored at its destination, such as in the cloud. Ideally, we would want an end-to-end acknowledgement, but this is not always possible with BLE, due to range and connectivity challenges. Instead EdiSense tries to replicate data, as much as possible/configured, in a local cluster to minimize the chance of data loss when end-to-end are not possible or can be indefinitely delayed. Moreover, in situations where sensors are not continuously recording data, EdiSense eliminates the burden on these sensors to serve their own data.

A. Distributed Hash Table

We considered multiple designs for our distributed hash table, including consistent hashing and a Dynamo [2] style approach with fixed partitions. In the end, we chose a simpler design with partitions representing replicated buckets. This approach simplifies our protocol for bounding inconsistency between each node's view of the DHT. In practice, since we expect wireless sensor networks to be deployed statically, frequent rebalances and joins/leaves are not a problem. For our implementation on Intel Edison, the DHT requires no more than 1-2KB of memory to maintain at each node.

B. Replica Placement and Load Balancing

Many factors may be important in selecting locations to place replicas. First, Intel Edisons are well suited for ad-hoc wifi networks. For such a network, we may wish to store replicas nearby to minimize the number of hops. Alternatively, we may wish to spread replicas as far apart as possible to minimize the possiblity of failure due to correleated events or environmental factors. Because the replication requirements vary depending on the deployment, we make it simple to write custom rules for allocation and rebalancing. In our current implementation, we use storage utilization as a criteria for rebalancing. For partition donates, we

select a suitable recipient using rules such as whether the recipient already holds a replica and a randomized ”power of two choices” algorithm [4].

C. Consistency

In our design, we must maintain consistency between each node’s view of the DHT. When we designed EdiSense, we considered RAFT, Paxos, and the 2PC protocols, but ultimately chose to design our own protocol. Because the Edison is a low power device possibly operating on an ad-hoc wifi network, we wished to avoid a single leader as that may increase load and latency. We also want rebalancing happen as quickly as possible so as to not block writes. Consequently, our approach for EdiSense requires immediate agreement from only two nodes. Other nodes in the cluster are notified in a delayed manner. For our use case, this is effective because we expect each node to communicate frequently (about particular partitions) with only as small number of other nodes; for example, if A is the closest node to a BLE sensor, whose data is stored on B, then A and B will frequently communicate. EdiSense ensures that these frequent communcatees will quickly learn of rebalances quickly. Our protocol provides stronger, more bounded, consistency guarantees than one based on gossip.

VI. Related Work

Glouse, Grossklags, and Chuang work on R-DCS (Resiliant Data Centric Storage) [3] for ad-hoc wireless networks uses a number of similar techniques to the ones we use in EdiSense. They use a DHT with consistent hashing but also rely on data type in making replica placement decisions to avoid flooding the sensor network with queries at query time. At the present, our algorithm for determining replica placement is naive. Instead, we store data in binary form and make no assumptions about the content of sensor data. We assume that the sensor/device ids are known and organize our store (DHT) by device id.

VII. Future Work

Maintaining a cached view of the DHT in memory places limits on the scalability of EdiSense. Our current method of deploying EdiSense is also not practical for more than 10-20 nodes. We believe that EdiSense makes most sense in small clusters dominated by BLE peripherals. To support larger sensor networks, we could adopt a hierarchy in organizing the nodes.

EdiSense presumes that end-to-end acknowledgements from BLE a sensor to the final data consumer or store is impossible or impractical. For EdiSense to act as an intermediary between the two endpoints, it needs a reliable way of acknowledging receipt of data for storage to be freed on the sensor. This remains unimplemented, but we considered two approaches. The first is to explicitly acknowledge each data point at an application level as it is received and replicated throughout the EdiSense cluster. This would presumably involve communicating over bluetooth characteristics. The second approach is to bound the amount of time to detect replication failures. Each sensor would maintain a retransmission buffer should the EdiSense node that it is communicating with fail or become unresponsive.

Finally, EdiSense uses Wi-Fi for communication between cluster nodes. It would be interesesting to investigate possibilites for data replication in a distributed sensor network using only BLE.

VIII. Conclusions

We built EdiSense as a prototype for a distributed data store geared toward low power sensors and devices. We make several simplifying assumptions, but in the process have gained valuable insight into the challenges of designing and implementing protocols for a replicated sensor network with low power devices such as Intel Edison and even smaller sensor peripherals.

Acknowledgment

We thank Professors Dawson Engler and David Mazieres for teaching CS244B and for the opportunity to build EdiSense, and Professor Philip Levis for his advice and inspiration for the project idea.

References

[1] Texas Instruments, SensorTag

[2] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon’s highly available key-value store. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP ’07). ACM, New York, NY, USA, 205-220. DOI=10.1145/1294261.1294281 http://doi.acm.org/10.1145/1294261.1294281

[3] Abhishek Ghose, Jens Grossklags, John Chuang. 2002. Resilient Data-Centric Storage in Wireless Ad-Hoc Sensor Networks. In Proceedings of 4th International Conference, MDM 2003 Melbourne, Australia. DOI=10.1007/3-540-36389-04

[4] Mitzenmacher, M., ”The power of two choices in randomized load balancing,” Parallel and Distributed Systems, IEEE Transactions on , vol.12, no.10, pp.1094,1104, Oct 2001 doi: 10.1109/71.963420

[5] LinkedIn. Voldemort. A Distributed Database. http://www.project-voldemort.com/voldemort/design.html