**Development**

1. TPAggregator is built using cpprestsdk (<https://github.com/microsoft/cpprestsdk>). This listens to the /flows endpoint for GETs/POSTs. It creates a threadpool and assign every new connection to one of the threads in the pool.
2. Endpoints are implemented in microsvc\_controller.cpp which do the HTTP parsing and pass the JSON body to the parser.
3. JSON parsing and calls to store the key/val in in-memory-db is done in parser.cc
4. Key/vals are stored in in\_memory\_db.hpp. micro-service is very write-heavy and hence designed the db to have very small critical section to avoid contention in between threads.
   1. db = hash\_map(int hour, Table \*table). Keeping a separate table for every hour, so GET query needs to read one specific table.
   2. Table stores the key/values, where key is src\_app:dest\_app:vpc\_id and value is {src\_app, dest\_app, vpc\_id, bytes\_tx, bytes\_rx}. Locking is done at bucket level, so multiple POSTs whose keys are mapped to different buckets will work in parallel without contention. Table size in bytes for the Table is also maintained, so memory for output of GET query can be reserved (avoiding multiple allocs).
5. transacton.hpp provides an interface between parser and memory db, keep transient store and store to db.
6. redis\_client.hpp provides a redis client interface which subscribe to the channels from all other instances. Upon receiving messages from other redis clients, it will update in memory db.

Development Enhancements:

1. In current implementation, there are 1031 buckets allocated for per-hour table. We can have one thread which periodically monitor the load factor (total elements/table size) per table and if load factor exceeds certain threshold, table can be resized and rebalanced. This can be done with one table level lock, atomic refcounting and conditional variable. Fast path get/set should still only have bucket level locks.
2. Logger/stats/metrics

**High Level Proposed Design:**

**Service Characteristics:**

1. write-heavy
2. aggregating, so assuming data-correctness is not very important. Few missed data points might be ok.
3. Many instances of many service app publishing data, so higher concurrency to the service is critical.
4. Expecting frequent POSTs with same {Src\_app, dest\_app, vpc\_id} as services periodically advertising throughput. So, keys/values will fit in a single large physical server with GBs of memory.

Redis Server

TPAggregator

Database

Cassandra

LoadBalancer

Service Apps

**Design Consideration:**

1. Run multiple instances of TPAggregator on different physical machines.
2. All TPAggregators keep all keys/values in memory. All Instances sync up using redis pub-sub.
3. Database (Cassandra) can be used to provide persistent storage. As the service is aggregating, every update is not required to publish to database. There could be one dedicated instance of TPAggregator which periodically publish the table to Cassandra.
4. All TPAggregators to publish an endpoint to check its health to be used by loadbalancer.
5. Loadbalander to distribute traffic across all instances. Loadbalancing strategy could be simple round robin or least connection/requests served.
6. In the failure event, node running TPAggregator goes down. It will come back up, read Table from Cassandra and once has all key/values in memory, start its health endpoint to start taking traffic from LB.
7. Cassandra DB partition key is hour, so all POSTs from same hour goes to same partition.