Real-time Storm Continuity-of-Service:

Recovering a Storm task's in-memory state after failure (R&D)

By: Noelle Milton Vega

Instrumented while architecting a PUB/SUB Complex Event Processor (C.E.P.) for Bank Of America using **Apache Storm**, **Apache Cassandra** and **Redis** to identify **F.I.X. Trade-Stream Anomalies** at a rate of **12-million events/second**

What is Storm's behavior when a task/instance of a component fails?

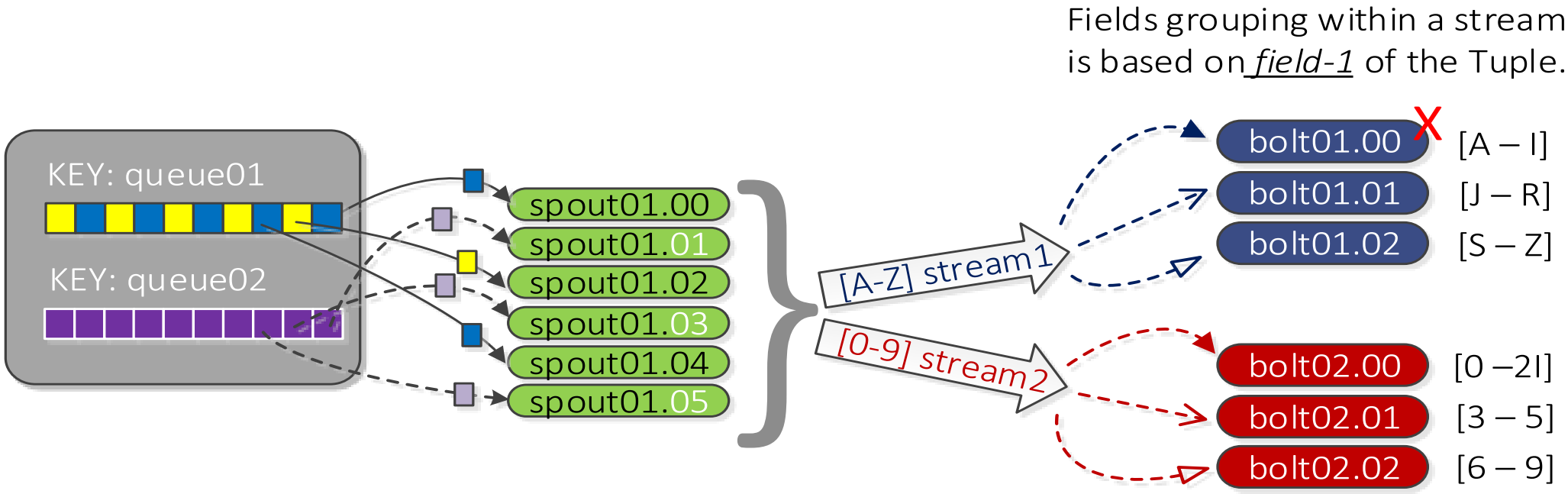
Specifically, what happens below when, say, task bolt01.00 (i.e. instance "00" of bolt01) dies?

1) Does Storm redistribute the [A - I] tuples that were fields-grouped to it to the surviving instances?

2) And will the task instance that replaces it be given the same identifier? In other words, will the return value of **context**#*taskIndex()* still be "00"?

I posed this question more precisely here: http://bit.ly/1bsBooT

The question went unanswered and no documentation was available on these behaviors, so I conducted lab experiments to observe them. As will be illustrated, *if* across task fail/restarts Storm (a) persists the fields-grouping to the task and (b) persists the task*Index* identifier to the task, then it will be possible to recover the in-memory state of the task prior to its failure. In other words: preserve a Continuity of Service across task failures.



What is Storm's behavior when a task/instance of a component fails?

By: *Noel Milton Vega*

Specifically, what happens below when, say, task bolt01.00 (i.e. instance "00" of bolt01) dies?

1) Does Storm redistribute the [A - I] tuples that were fields-grouped to it to the surviving instances?

2) And will the task instance that replaces it be given the same identifier? In other words, will the return value of **context***#taskIndex()* still be "00"?

Programmatically what we are asking above is, across component task fail/restarts, are the two conditions below true?

```
// ================================
// Bolt Constructor.
// ================================
public bolt01(Properties properties) {
}

// ================================
// Bolt prepare() method
// ================================
public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
}

// ================================
// Bolt execute() method.
// ================================
public void execute(Tuple inTuple) {
}
```

Is task identification remembered here?

Is grouping remembered here?

R&D Test setup and results

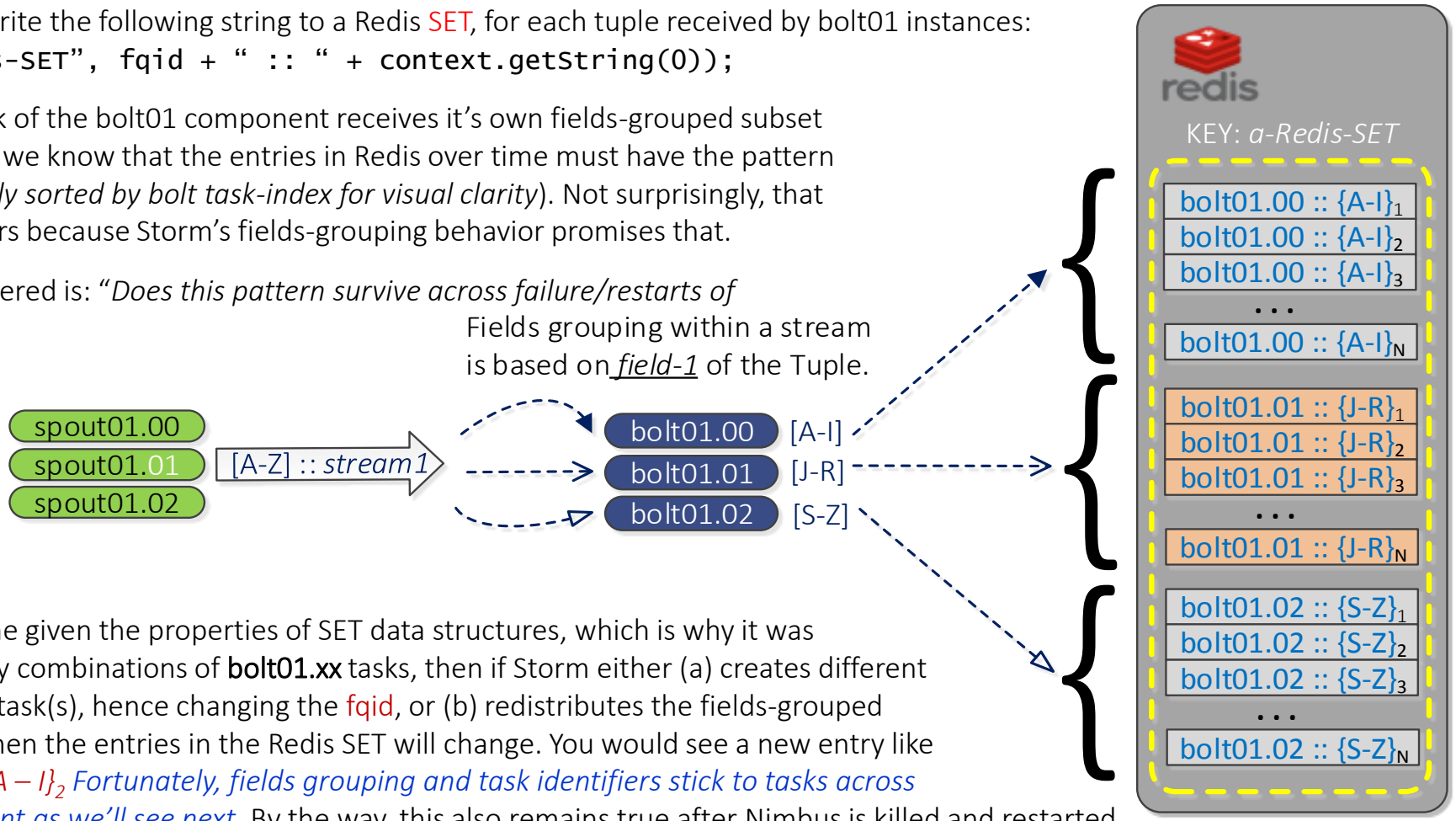
- Java code for every task in a Storm topology can assign a unique Fully Qualified I.D. String, using the following statement:
  fqid = context.*getThisComponentId()* + ".0" + context.*getThisTaskIndex()*;
  - Where context.*getThisComponentId()* is the component name (e.g. **spout01**, **bolt01**; **acmeBolt**; etc.)
  - Where context.*getThisTaskIndex()* is the task index number for that component (e.g. **0**, **1**, **2**, …, **N-1**)
  - Where, therefore, a task's *fqid* becomes (by examples): **bolt01.02** or **acmeBolt.05** or **spout01.00**, and so on.

- Leveraging that *fqid*, let's write the following string to a Redis SET, for each tuple received by bolt01 instances:

```
redis.sAdd("a-Redis-SET", fqid + " :: " + context.getString(0));
```

Since, as noted below, each task of the bolt01 component receives it's own fields-grouped subset of tuples based on *field-1*, then we know that the entries in Redis over time must have the pattern shown to the far right (*artificially sorted by bolt task-index for visual clarity*). Not surprisingly, that pattern is precisely what appears because Storm's fields-grouping behavior promises that.

But the question we want answered is: "*Does this pattern survive across failure/restarts of component tasks?*"



This is easy enough to determine given the properties of SET data structures, which is why it was selected: If you "**kill -9**" arbitrary combinations of **bolt01.xx** tasks, then if Storm either (a) creates different identifiers for the replacement task(s), hence changing the fqid, or (b) redistributes the fields-grouped tuples among surviving tasks, then the entries in the Redis SET will change. You would see a new entry like this, for example: *Bolt01.01 :: $\{A - I\}_2$ Fortunately, fields grouping and task identifiers stick to tasks across fail/restarts, and this is important as we'll see next.* By the way, this also remains true after Nimbus is killed and restarted.

Stickiness of fields-grouping & task IDs across tasks restarts is important!

- Since we observed (at least anecdotally) that a tasks's *fqid* sticks to it across fail/restarts, we can use that *fqid* as a unique key/index into Redis to store duplicate copies of task-index specific data out-of-band (i.e. external to the task JVM). Task-index specific data includes, for example, fields-grouped data that only that task index (instance) receives.

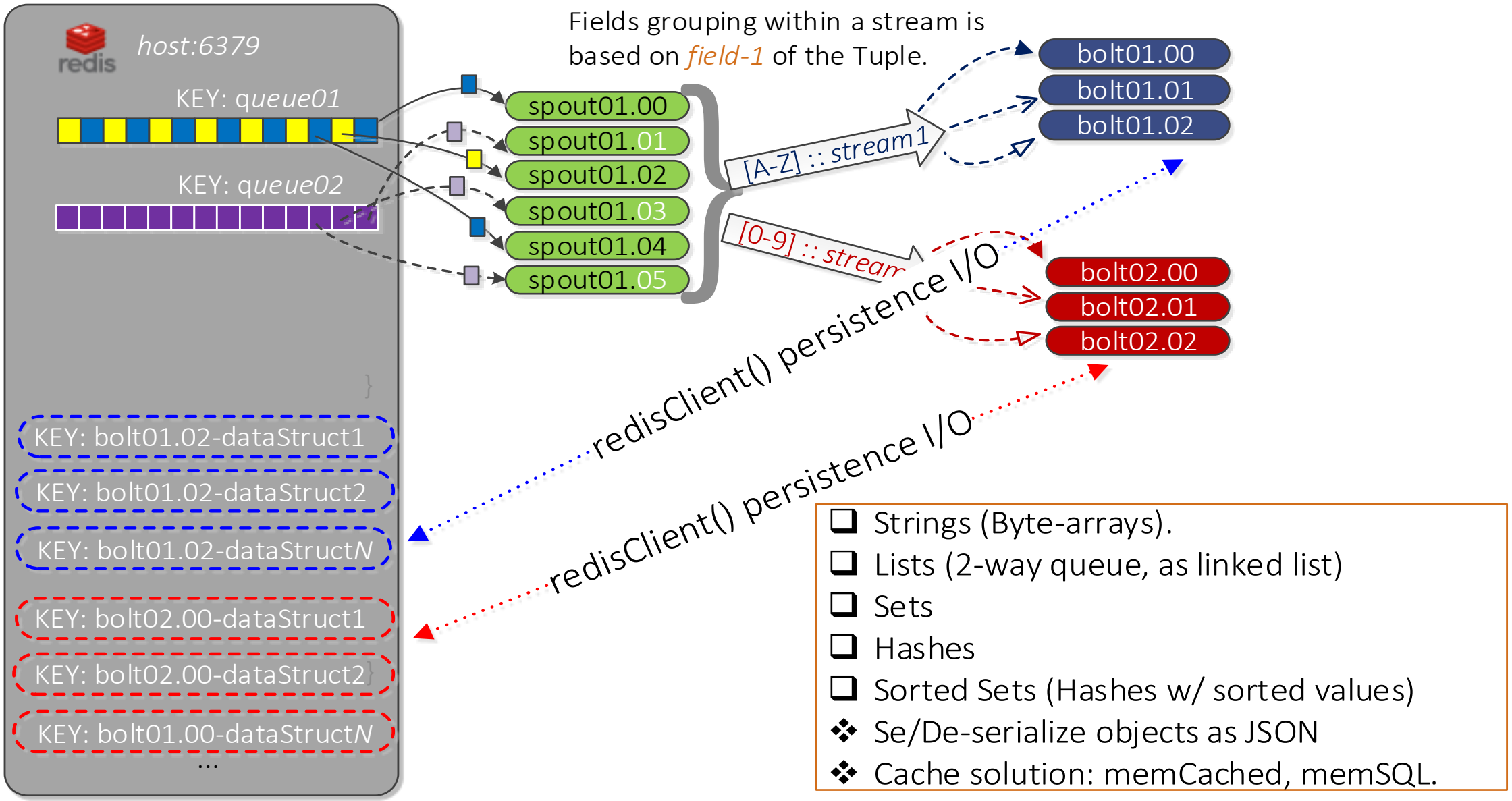
- Since we also observed that tasks will continue to receive the same fields-grouped collection of tuples before & after task failure, we know that - after recovering the lost data structures from Redis - we can resume processing because (a) tuples that timed out will be replayed by Storm's Guaranteed Message Processing facility, and (b) the fields-grouped tuples that pertain to the data structures recovered from Redis will continue to arrive. The implementations of this recovery strategy is shown in the code fragments below, and diagram on the next page.

```
// ================================
// Bolt prepare() method
// ================================
public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
    [ ... snip ... ]

    componentID = context.getThisComponentId();                    // e.g. bolt01; spout03
    taskIndex   = context.getThisTaskIndex();                      // [0-(N-1)]; N = number of tasks for this component.
    fqid        = componentID + ".0" + Integer.toString(taskIndex); // bolt01.04; spout03.00

    redisKeyPrefix = fqid; // Use your unique Fully Qualified I.D. as a Redis key prefix.

    // Establish connection to Redis [not shown], and recover lost data structures, if any.
    // In this example statement, we are recovering a HashMap.
    hashMap  = jedisClient.hgetAll(redisKeyPrefix + "-myMap");      // Redis key for bolt03.04: "bolt03.04-myMap"
}

// ================================
// Bolt execute() method
// ================================
public void execute(Tuple inTuple) {
    [ ... snip ... ]

    String customer = inTuple.getString(0);
    double balance += inTuple.getString(1);

    // After computing a new balance for the customer, we update/store it in two places ...
    hashMap.put(customer, balance);                                 // Store it in-band (in JVM)
    jedisClient.hput(redisKeyPrefix + "-myMap", customer, balance); // Store it out-of-band (in Redis).
}
```

FQID is maintained across task Fail/Restarts (i.e. for the lifetime of the topology).

Tuple grouping/partitioning is maintained across task fail/restarts (i.e. for the lifetime of the topology).

State Persistence strategy illustrated for *bolt01.02* & *bolt02.02*



- ❑ Strings (Byte-arrays).
- ❑ Lists (2-way queue, as linked list)
- ❑ Sets
- ❑ Hashes
- ❑ Sorted Sets (Hashes w/ sorted values)
- ❖ Se/De-serialize objects as JSON
- ❖ Cache solution: memCached, memSQL.

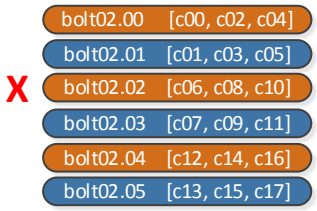
PRISMALYTICS, LLC. | DATA ANALYTICS TO LITERALLY COUNT ON
P: 212.882.1276 | E: CONTACT@PRISMALYTICS.IO | W: PRISMALYTICS.IO

Storm (by twitter) / Redis testbench

By: *Noel Milton Vega*

Architect/Written By: *Noel Milton Vega*
Document version: *0.81*

THE THOUGHT EXPERIMENT



Bolt02.02 = Tony / Bolt02.05 = John:

- Tony works for ***A1-Accountants*** and his role (via bolt02.*02*) is to deposit checks he receives from their client, *ACME* corp. into ACME's bank account. John also works for ***A1-Accountants*** and performs the same role , but for client, *FOOBAR* corp. (via bolt02.*05*).

- One evening Tony deposits checks into the ACME bank account, and places the deposit receipts on his desk. The next day Tony will record entries into ACME's check-register booklet for those receipts, and complete the *transactions*.

- Unfortunately Tony gets hit over the head before he is able to update ACME's check-register booklet.

- Two days later, Tony wakes up suffering from partial amnesia: He remembers that he works for *A1-Accountants* and general his role there is (bolt02), but he cannot remember his name (bolt02.*02*). So when he returns to work, he sees stacks of unregistered deposit receipts on both Tony's and John's desk but, because he doesn't know his name, he doesn't know which stack to resume from.

- The result is that Tony has ATOMIC operations that never completed. And other than desk proximity, John (bolt02.05) doesn't know about Tony (bolt02.02) or see his backlog, and so can't complete Tony's *transactions*. John has his own (*fields grouped*) customer, checks, bank, and check-register to deal with.

If only Tony could remember his name:

If he could remember *"I'm Tony not John"* (which is analogous to the need in Storm to re-self-identify distinct task-instances across Storm task fail/restart events), then Tony would know to look for unregistered bank receipts on his desk (i.e. to check his task-exclusive Redis/MemCache/C* temp *cache*) and resume where he left off. Tony won't look at John's desk (i.e. his task-exclusive Redis/MemCache/C* temp cache), because those receipts don't pertain to his pending incomplete transactions.

THE ANALOGY (using a Redis List, but it could be any data structure): The bolt02 component uses the PREFIX, "myDesk" for a Redis key-name, but task instances of bolt02 (e.g. bolt02.02, bolt02.05, etc.) distinguish their caches by SUFFIXING (appending) the key-name with a FULLY QUALIFIED task specific identifier, like so:

REDIS KEY: `myDesk`-bolt02.*02* (Tony's desk)

REDIS KEY: `myDesk`-bolt02.*05* (John's desk)

The above architectural thought experiment, then, led to my having to find the answers to the below Storm questions. And that is what drove the design of a Topology (thoroughly documented in pages that follow) to answer critical ***undocumented*** Storm questions, on which BAML ARC's success depends as they employ it. The knowledge/results obtained from this Big Data R&D exercise, are what I would ultimately provide to off-shore developers to leverage. There will be a lot of Storm (and other Big Data) questions as we move along.

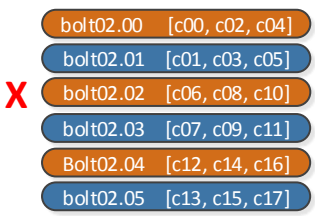
Q1: What is Storm's behavior when Task fail/restart event occurs?
When a Storm component (Spout/Bolt) Task fails and restarts, does the component's **CONSTRUCTOR** and get re-executed and/or does its **open() (Spout)** or **prepare() (Bolt)** get re-executed? Or, alternatively, does a serialized representation of the object (serialized after open() / prepare() complete), simply get de-serialized to bring it back to life, thus going straight to the **nextTuple() (Spout) / execute() (Bolt)** methods? Not documented.

It matters because connection code may not get re-run in a task fail/restart scenario.

Q2: Does Storm provide the ability to distinguish/self-identify between component task instances? (i.e. I'm Tony versus I'm John).
Indeed Storm does provide a method **`context.taskID()`**, but this is a topology-wide ID, not a component specific ID, so its not as useful. Through testing, I was able to determine that Storm provides what I wanted via **`component.taskIndex()`** method (though it's not explicitly nor well documented). I use it as follows to **FULLY QUALIFY** a Storm topology-wide task like this: "**`context.componentId() + "." + context.taskIndex()`**". Ex: "**`bolt02.02`**".

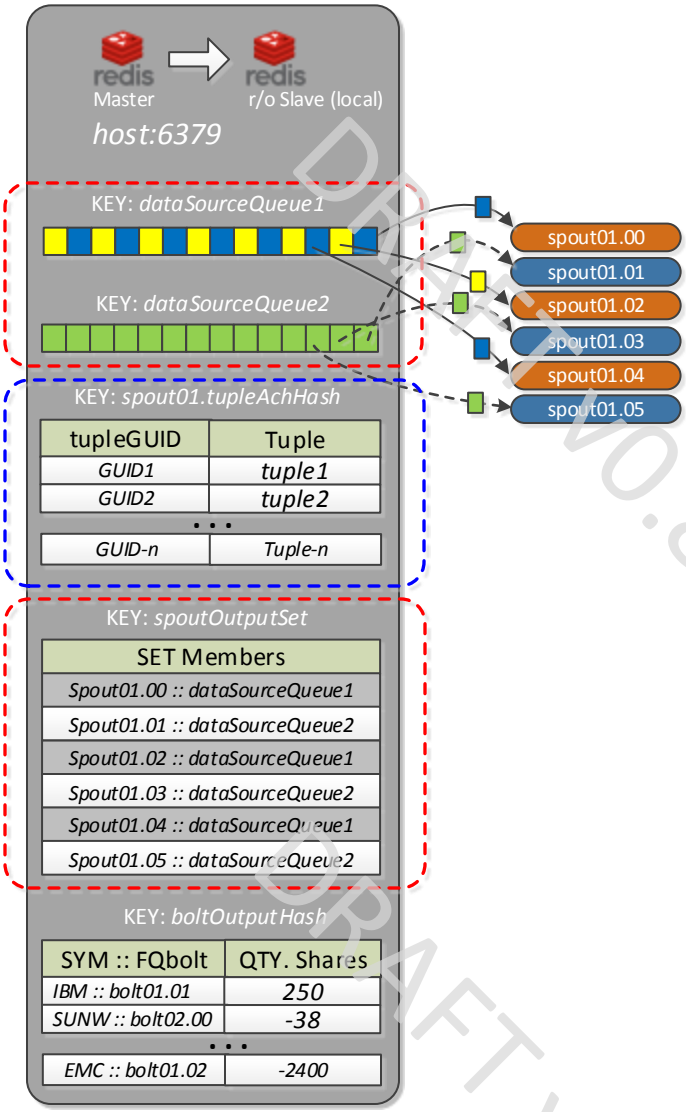
Q3: Now, the necessary follow-on to Question-2 is, does the **`context.taskIndex()`** return-value persist across task fail/restarts (i.e. sticky)? This is what allows Tony and John to know who they ***were*** and ***are*** upon task recovery and to, using the **FULLY QUALIFIED** topology-wide task identifier above, re-connect to their specific temporary caches (e.g. **`myDesk-bolt02.02`** -vs- **`myDesk-bolt02.05`**) and resume processing. (In other words, context-aware restart-ability).

Q4: Next, across task fail/restarts, do Tuples that were grouped to go to a specific task instance (as would be the case for Fields Grouping), continue to get routed to that task instance after recovery? Or does Storm re-distribute that collection of Tuples (e.g. INET, SUNW, YHOO) to other tasks instances? There is no documentation on this behavior either – whether during one task instance failure or multiple simultaneous task instance failures, as happens when they are running on the same failed Executor / Worker (... not uncommon).




```
Bolt01() {
   // Constructor code. Runs only once.
}

prepare() {
   // Preparation code. Runs only once.
   // Evaluate "context.componentId() + "." + context.taskIndex()"
   // and, if necessary, use it to reconnect to any temporary caches this instance may have
   // been using to facilitate recovery (and complete incomplete ATOMIC operations).
}

execute() {
   // Runs many times per second.
}
```

Architect/Written By: *Noel Milton Vega*
Document version: *0.81*



Design goal(s):
- To minimize load on the PUB/SUB message bus, the en-queue bridge (not shown) feeding the dataSourceQueue01/02 queues will subscribe only once to each topic. The intermediate serialization is quickly de-serialized by having one Spout consume from multiple queues, with multiple de-queue connections to each.
- During simulated Storm failures, observe whether the fail/restart of a Task instance continues to service the same queue (i.e. stickiness). This will depend on Storm's taskIndex number persistence.
- Gauge concurrent access behavior of Redis data structures (List/Hash in this case).

Implementation:
Using the remainder of the modulo calculation, "**this.TaskIndex % 2**", EVEN numbered Spout tasks are steered to dataSourceQueue01; while ODD numbered Spout tasks are steered to dataSourceQueue02. For ever increasing throughput rates from canonical sources and/or or increases in the number of data sources themselves, this strategy can adjust to meet big-data-Volume increases by increasing the number of Spout tasks and using a larger modulo operand (3, 4, and so on).

Next, to track whether Spout-Task to Data-Source bindings persist across Task fail/restarts, a Redis SET (**key: spoutOutputSet**) is employed: Each Spout task will write the signature string "***Spout01.nn :: dataSourceQueueX***" to indicate which data source it is handling. If Storm persists taskIndices across Task fail/restarts, then the set will always appear exactly as shown. If, on the other hand, say, Spout01.03 handles requests for dataSourceQueue2 and later for dataSourceQueue1, then the set will have two member entries for Spout01.03, indicating a transition has occurred. The String contents (and code) was designed ensure that a query of this Redis SET at any random time, will reveal Spout task handling transitions that may have occurred.

Design goal(s):
- Implement an out-of-band (outside Storm) field/value store to leverage Storm's Guarantee Message Processing (GMP) facility. Here we employ a Redis HASH to store <tupleGUID, Tuple> pairs for each in-flight tuple that has not yet received an ack() in the responsible Spout task.

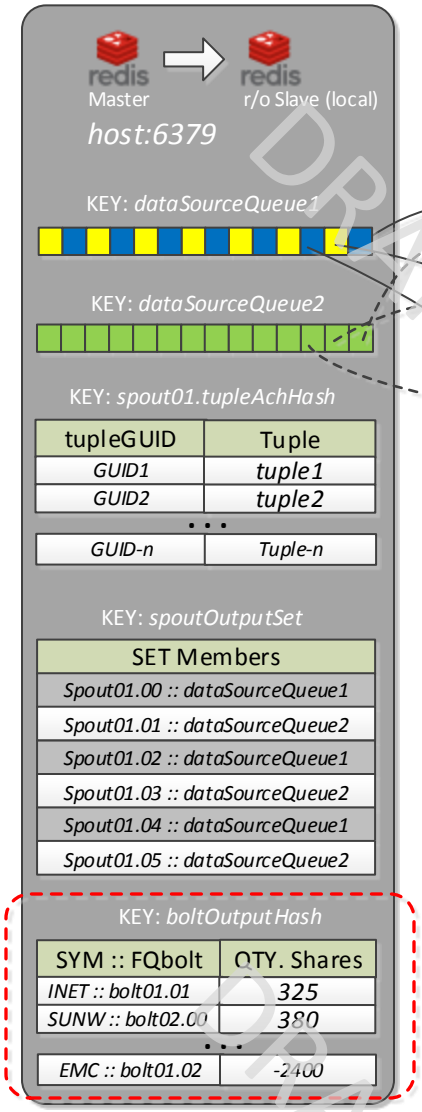
Implementation:
A Redis HASH is employed: **key: spout01.tupleAckHash**. The Fields are tupleGUIDs, (*java.util.UUID.randomUUID().toString()*) and the Values are serialized Storm Tuples themselves. The Spout makes a "hexists()" call to Redis to ensure the generated tupleGUID field is not already in use, and generates another in the rare case when it is in use. This data structure is shared among all Spout01 Task instances, as is hinted by it's key name: **spout01.tupleAckHash**. Had I chosen to employ per-Task Hashes, the key names for each would have included per-Task instance information in them (for example, "spout01.*NN*.tupleAckHash").

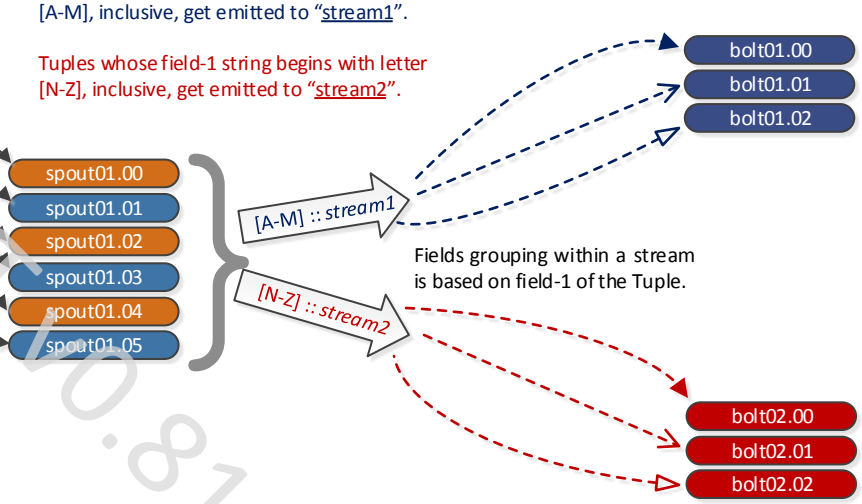
Per-Task Hashes were not used because, here again, we might be able to glean information about Redis' ability handle multiple concurrent access to its Hashes.

Finally, the memory foot-print of this data structure will initially grow at Topology start, but then stabilize to some capacity range as long as Storm tasks are not blocked from ack()ing the Tuples they receive. To reduce the possibility of this happening, where possible connections and calls to external I/O resources (such as databases, message services, etc.), should make use of timeout clocks, and call _collector.fail() if/when they expire.

DRAFT v0.81 (NMEGA)

Architect/Written By: *Noel Milton Vega*
Document version: *0.81*





Design goal(s):
- Route (i.e. stream) Tuples to specific Bolts (and accompanying logic) for Tuple-content and/or state-specific specific processing. The ARC topology will require this.
- Send the same group of Tuples to same Bolt task based on some collection of Tuple fields(a.k.a, Storm Fields Grouping). The goal here isn't to test fields grouping necessarily, but to observe fields grouping behavior across fail/restart of Bolt tasks, since tasks often carry state.

Implementation:
- Two Spout01 streams are defined in the RedisTopology class: "stream1" and "stream2". Bolt01 subscribes to "stream1", and within that (non-default) stream, Bolt01 tasks receive "fields grouped" Tuples based on field-1 of those Tuples. Similarly, Bolt02 subscribes to "stream2", and within that (non-default) stream, Bolt02 tasks receive "fields grouped" Tuples based on field-1 of those Tuples.

How are streams subdivided:
Spout01 will route incoming messages to "stream1", if the message begins with the letter [A-M], inclusive.
Spout01 will route incoming messages to "stream2", if the message begins with the letter [N-Z], inclusive.

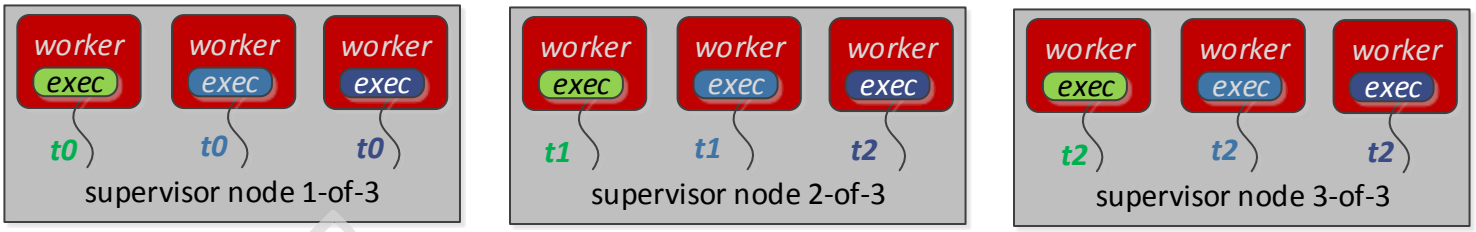
The expected format of the incoming messages, by the way, is "SYMBOL:QTY". For example: "***ORAC:250***" (sans the quotes). *Note: The reader shouldn't become distracted by the message meaning, which just as easily could have been something like, "MOVIE-ORDERED:IP-ADDRESS".* The important thing to understand here is that, because SYMBOL begins with the letter '***O***' in this example, the Tuple emitted for it by Spout01 will be to "stream2", so Bolt02 logic will handle it. On the other hand, the Tuple emitted by Spout01 for the message "***INET:325***", will be to "stream1", so Bolt01 logic will handle it. In either case, the Tuple emitted will be comprised of two fields, like so: <SYMBOL>, <QTY>. In other words, Spout01 receives the message, tokenizes it based on the colon ":" separator, and creates and emits a 2-Tuple. Continuing with these example messages, the Bol01 task that first receives the "<INET>,<325>" Tuple, will receive all subsequent Tuples whose field-1 is <INET>. Similarly, the Bol02 task that first receives the "<ORAC>,<250>" Tuple, will receive all subsequent Tuples whose field-1 is <ORAC>.

That's the theory and hope anyway, which brings us back to our core test-case, and our use of Redis again to help: *Since tasks often carry state, we want to observe Storm's fields grouping behavior across fail/restarts of Bolt tasks under simulated failure .*

To help us track changes to fields group mappings, a Redis HASH is again employed: (**key: boltOutputHash**). Here, we use the same strategy we used for "spoutOutput**Set**": Since field names in a HASH are essentially a SET, we'll design a field string format for "boltOutputHash" such that entries for a particular SYMBOL can **only ever appear once!** To do this, let's revisit the Tuple, "<INET>,<325>". Because of our streams grouping, we know that some Bolt01 task (and not Bolt02) will process this. Next, because of fields grouping, we also know that if that Bolt01 task was, say, Task-01, then all subsequent Tuples having the pattern, "<INET> <QTY>", will also be processed by that same Task-01. Stated succinctly, INET will always be processed by the specific, Fully Qualified Storm bolt task, "bolt01.01"; and this relationship is what we have stamped as the first entry in the **boltOutputHash** diagram to the left (enclosed with a red dashed line). In the entire set of boltOutputHash's HASH fields, INET can only appear once, and exactly as the string: "*INET :: bolt01.01*". If INET appears more than once then, because of the SET nature of HASH fields, it means an INET Tuple was serviced by some other boltXX.YY combination. But why do we need the Hash values (the right hand side)? Couldn't we have just used a Redis SET? Besides knowing that we're tokenizing incoming messages into proper Tuple fields, the reason we need Hash values is to be able to observe that changes are happening at all. Sending a message "<INET>,<325>" followed by the message "<INET>,<-200>" (say, after a simulated failure), would change the value from "325" to "-200", letting us know that the INET entry remains unique, not because there was never another INET-related message, but because Storm is behaving as we desire (sticky/persistent fields grouping).

Finally, the size of this data structure is based on the number of SYMBOLS, so not very big.

DRAFT v0.81 (WIP)



- A ***supervisor***, which is the root JVM on a Storm-cluster node, can be a parent to multiple ***workers*** JVMs beneath it (i.e. supervisees). A worker belongs to a specific Storm topology, and the same supervisor can have workers from multiple topologies. As the RED color scheme above implies, the three workers on supervisor-1, supervisor-2, and supervisor-3 (nine workers in total) all perform work for the same Topology: the REDIS Topology.

- A ***worker*** JVM can be a parent to multiple ***executor*** threads beneath it (despite the one-to-one relationships shown above). An executor thread runs code for a specific Spout/Bolt topology component, and executor threads sharing the same parent worker don't necessary run code for the same topology component.

- A ***executor*** thread can run one or more ***tasks,*** each task running an instance of the same Spout/Bolt code. Tasks are not threads, so tasks that are mapped to the same executor are run sequentially/serially by that executor. Though not required, it is generally a best-practice to define topologies that result in a 1-to-1, task-to-executor ratio, as was done in the REDIS topology above. The other reason for the 1-to-1 ratios above, has to do with our test goal: At the UNIX process level, only workers can be killed (SIGTERM), so having a *1-to-1-to-1*, *worker-to-executor-to-task* ratio, allows us the ability to finely control the specific Spout or Bolt task we destroy. This, in turn, allows us to observe how and where in the code that specific Spout or Bolt task instance recovers.

- A ***task*** is a specific instance of a topology component that performs a unit of work, and is completely independent of other task instances of the same component.

The following illustration shows how data in the ***Redis*** topology is routed from Spout task-instances to Bolt task-instances, with the right-most column providing an example for a specific inputs stream:

comp.Index	Consumes	Receives	Emits	Example **received** stream and routing
spout01.00	Redis key: *dataSourceQueue* ***01***	Formatted string: *SYM:QTY*	2-Tuple: <*SYM, QTY*> to a *stream*:	INET:325, YHOO:200, ..., NDAQ:-500, SUNW:75
spout01.01	Redis key: *dataSourceQueue* ***02***	Formatted string: *SYM:QTY*	> to *stream1* if SYM begins w/ [A-M]	NDAQ, 600, DELL:200, ..., MSFT:122, YHOO:-150
spout01.02	Redis key: *dataSourceQueue* ***01***	Formatted string: *SYM:QTY*	> to *stream2* if SYM begins w/ [N-Z]	SUNW:125, MSFT:650, ..., INET:-200, DELL:150
bolt01.00	*stream1* @spout01	2-Tuple: <f_1 f_2>, fields grouped by f_1	N/A	<INET, 325>, ..., <INET, -200>, ...
bolt01.01	*stream1* @spout01	2-Tuple: <f_1 f_2>, fields grouped by f_1	N/A	<MSFT, 650>, ..., <MSFT, 112>, ...
bolt01.02	*stream1* @spout01	2-Tuple: <f_1 f_2>, fields grouped by f_1	N/A	<DELL, 200>, ..., <DELL, 150>, ...
bolt02.00	*stream2* @spout01	2-Tuple: <f_1 f_2>, fields grouped by f_1	N/A	<SUNW, 125>, ..., <SUNW, 75>, ...
bolt02.01	*stream2* @spout01	2-Tuple: <f_1 f_2>, fields grouped by f_1	N/A	<NDAQ, 600>, ..., <NDAQ, -500>, ...
bolt02.02	*stream2* @spout01	2-Tuple: <f_1 f_2>, fields grouped by f_1	N/A	<YHOO, 200>, ..., <YHOO, -150>, ...

As shown above and discussed throughout, one major goal of this topology design is to observe how Storm behaves under simulated failures. Specific questions are:

1) *When a Bolt instance (i.e. bolt01.xx or bolt02.yy) fails, does the RedisBolt() constructor and/or RedisBolt.prepare() method get re-executed? Or does it resume with the RedisBolt.execute() method? Similarly, when a Spout instance (i.e. Spout01.xx) fails, does the RedisSpout() constructor and/or RedisBolt.open() method get re-executed? Or does it resume with the RedisSpout.nextTuple() method? The underlying question here is: By what mechanisms are I/O sessions/connections to external resources recovered?*

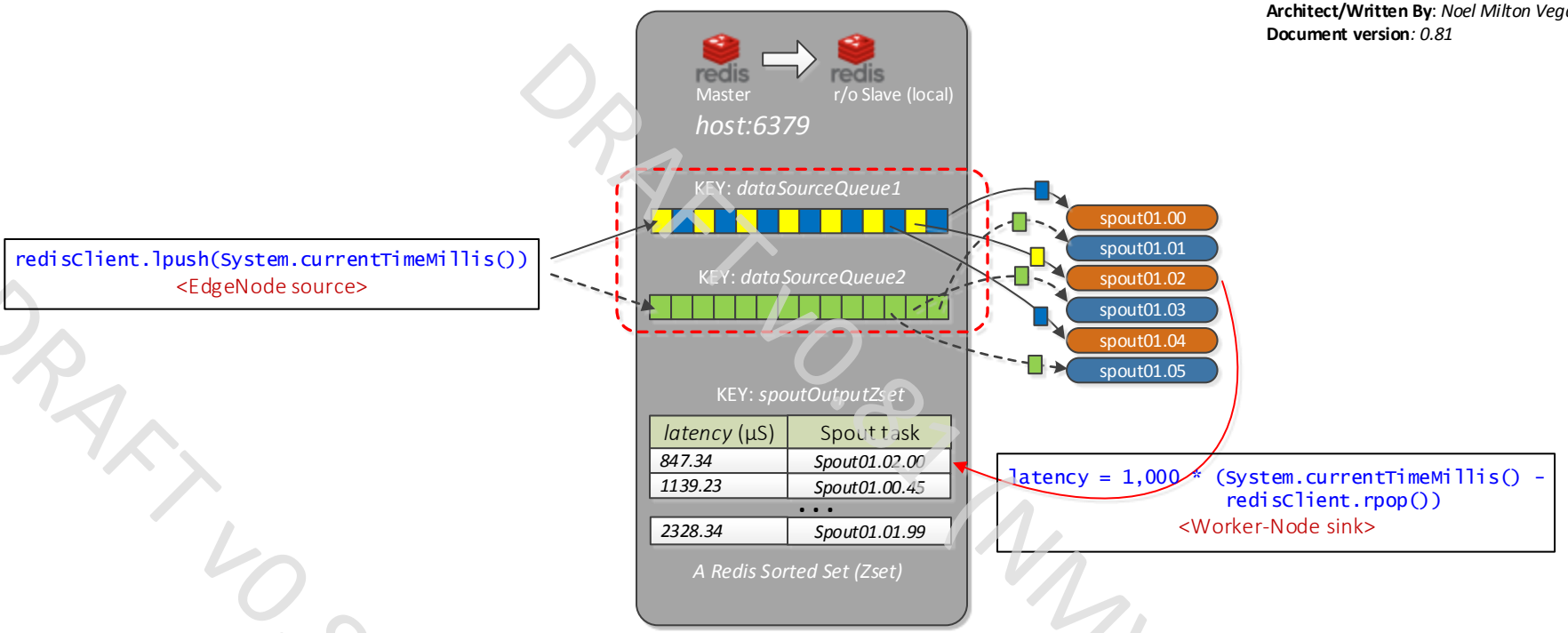
*3) Since Spout or Bolt taskIndex numbers can be used to invoke task-specific behaviors, does a task that fails & restarts maintain the same taskIndex number? For example, the remainder of the calculation, '**taskIndex % N**' (N = 2,3,4...), is used to steer Spout tasks to consume specific Redis keys (queues). It's therefore relevant to confirm taskIndex number persistence; and also to confirm whether task indices always maintain the range [**0 - (nbrComponentTasks – 1)**], irrespective of failures.*

4) When a Bolt instance (i.e. bolt01.xx or bolt02.yy) fails, does the replacement Bolt instance continue to receive the same subset of fields-grouped input data?

To answer some of these questions, at least anecdotally, I performed a before & after behavior comparison of the spout01.02 task after killing it's parent worker, and then did similarly for the bolt02.02 task. Both tasks were chosen arbitrarily and, given the symmetric routing scheme shown above, they are collectively necessary and sufficient. The pages the follow show a partial running dialog or results.

DRAFT v0.81 NMVEGA

Architect/Written By: *Noel Milton Vega*
Document version: *0.81*



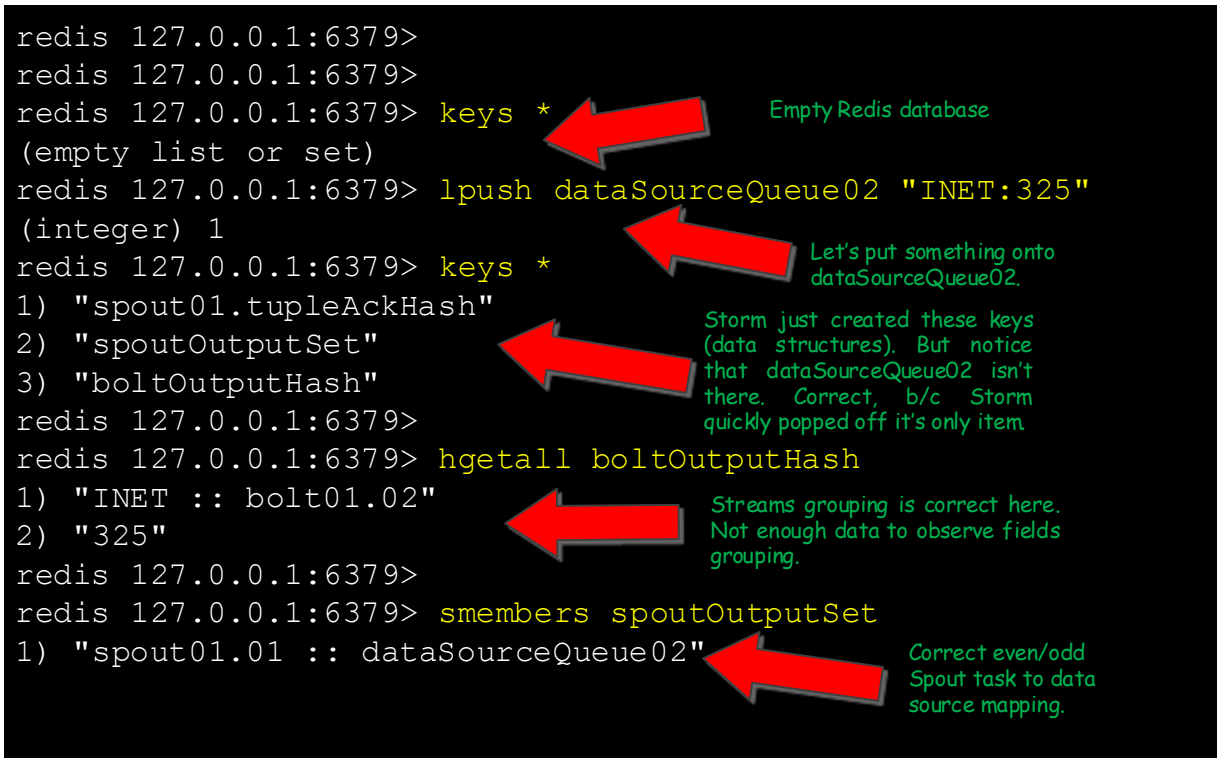
DRAFT V0.81 NMVEGA

Design goal(s):
- To observe Redis latency characteristics. There are many configurations possible (such as single or multiple queues; master/slave replication; persistence types and frequency; number of task instances; and so on), that interpolation/extrapolation of results from a few configurations will initially be necessary.

Implementation:
Redis will run on the EdgeNode where a co-located injector will continuously L-push the current NTP-synchronized system-time (`System.currentTimeMillis()`) onto one or more queues (Redis Lists). The requirement for having the Redis instance and it's injector be co-located, is to eliminate the additional network latency that would accompany use of a remote injector. That additional latency does not offer educational value since we are interested in Redis-IN / Redis-OUT differential timestamp values. The receiving Spout task will R-pop the timestamp value from the queue and subtract it from it's own, currently generated system-time. The difference is the *"Redis IN/OUT latency"* + "*EdgeNode-to-WorkerNode network latency*". This egress network latency cannot be avoided because we need to employ Storm in DISTRIBUTED MODE; however, we can adjust for this by either (1) subtract a constant value that ***approximates*** this A-to-B network overhead or, more accurately (2) by subtracting the ***measured*** average values of this latency by timing a blocking call of a write back to a Redis Sorted Set (more on that next).

On each call to Spout.nextTuple(), the Spout task calculates the differential in system-time values and record it to a Redis Sorted Set (or ZSET). A ZSET data structure was selected here because it's fields are automatically sorted, giving us the ability to easily obtain **Top-*N*** like results for the recorded latencies. Each Spout task instance will record 100 of it's own latency values (so for example, three Spout tasks would produce a total of 300 latency values in the ZSET), and the **top-*N*** latencies from that collection will bubble up to the top in a sliding window fashion.

The design decision to use one target ZSET to record all Spout task latency calculations – versus using a dedicated ZSET for each – certainly gives us a *single-pane-of-glass* view into latencies across the board. But there was another reason: Intuition suggests that multiple simultaneous writers (Spout tasks) to the same ZSET would incur more serialization and sorting load on the Redis instance, as a whole. This additional processing burden could, of course, add to Redis-IN/Redis-OUT queue latencies; so the values we obtain for them under this setup, would be a worst-case scenario like analysis… i.e. things should improve from there. At least that's what intuition suggests… which is sometimes wrong, but it's necessary to start somewhere.

Finally, as will other tests, by using the remainder of the modulo calculation, "**this.TaskIndex % 2**", EVEN numbered Spout tasks are steered to dataSourceQueue01; while ODD numbered Spout tasks are steered to dataSourceQueue02. For ever increasing throughput rates from canonical sources and/or or increases in the number of data sources themselves, this strategy can adjust to meet big-data-**V**olume increases by increasing the number of Spout tasks and using a larger modulo operand (3, 4, and so on).

Sample Storm/Redis CLI interaction, starting with an empty database.