Replication Schemes to Support Failure Resilient Processing of Real Time Data Streams

Kathleen Ericson and Shrideep Pallickara
Computer Science Department
Colorado State University
Fort Collins, CO USA
{ericson,shrideep}@cs.colostate.edu

Abstract— In this paper we explore the use of replication for fault tolerant processing of streams. We perform these experiments in the context of the Granules stream processing system that is designed for real time processing of data streams generated by devices and instruments. In this paper we explore well-known replication schemes for fault tolerant processing of data streams. We analyze two basic approaches to replication: active and passive; within the context of the Granules cloud runtime. As a proof of concept, we work with a medical dataset from a patient monitoring system that keeps track of patient respiration.

Keywords — Granules, MapReduce, Streaming Data, Replication, Fault Tolerance

I. INTRODUCTION

In distributed systems, data and computations are often replicated to increase availability. This is done by either allowing clients to make use of any replica in a set — providing concurrent access, or by allowing a replica to take over computation when failures occur. In distributed streaming databases, it is more common to see the latter case. In such an environment, multiple copies of a processing step are replicated across many machines, and communications are rerouted as needed when failures occur.

In this paper we develop a scheme which allows us to perform real-time processing of streams in a fault tolerant fashion. Distributed streaming databases are designed to handle queries on streaming data, and can often be found in the area of stock analysis [1]. While both distributed streaming databases and Granules share the ability to operate upon constantly streaming datasets, the types of supported operations are quite different. Distributed streaming databases are built to perform SQL operations on streaming data such as windowed joins. Granules is a generalized stream processing system and allows arbitrary processing of streaming data (not just query evaluation). Computations in Granules can be developed in languages such as: C, C++, C#, python, R, and Java. These computations can be expressed either as MapReduce computations or directed graphs that may have feedback loops in them. Granules is expected to process data in real time in mission-critical settings such as monitoring systems [2] and EEG analysis [3]. In these settings accurate and timely processing of data despite failures that may take place is important.

To maximize resource utilizations, Granules interleaves multiple computations on a given machine. Should a machine fail, any computations hosted on that machine should continue to be available elsewhere. Since Granules is a generalized stream processing system, it has several distinct characteristics that make fault tolerance difficult. One difficulty lies in the characteristics of the data streams that we are dealing with.

First, streaming data produces several unique challenges. Granules streams in particular are characterized as being bursty, relatively small in size, and can occur around the clock (24/7). Even a short outage can result in the loss of a large quantity of data, and it can be difficult to quantify the cost of system downtime. The replication schemes discussed here all have a unique impact on the number of messages lost. One approach to avoid dropped inputs is to store incoming data to ‘replay’ if a computation fails. In Granules, however, this approach is not feasible. As data is being produced continuously, it is simply too voluminous to store everything.

Since Granules interleaves 100s of computations on the same resource, the loss of even a single machine affects these computations. In a typical distributed streaming database, only a single computation needs to be recovered when a single machine fails. In Granules, this is again further complicated because Granules supports generalized stream processing. This means that there may be an arbitrary, stateful, computation performed as part of processing. For example: in [3], we trained a neural network in each mapper, which was then responsible for classifying EEG data in real time. Training a neural network is very time consuming, so we would need to handle transferring the neural network state in case of failure.

We are working with the Granules cloud runtime to enable support for both active and passive replication approaches. State transfer is beyond the scope of this work, so with respect to consistency, we are viewing active replication as consistent — every replica sees all the data, so they will arrive at the same state, while the passive replication approaches are either inconsistent or eventually consistent, depending on whether state is cumulative or windowed. These approaches need to be able to mesh with the Map Reduce paradigm [4], and not hinder the ability to process streaming data in real time. We first look at active [5] and passive replication [6] within the context of a simple computation. From these proof-of-concept experiments, we are able to outline the tradeoffs between...
components actively and passively replicated within Granules. We then move on to benchmark a respiration monitoring system where data is constantly being observed from 100s of users simultaneously. In this experiment, we are using a mixture of both passive and active replication schemes to explore how scaling up communications effects replica and system performance. The goal of this exercise is to analyze this respiratory dataset in real time, while ensuring resilience.

A. Research Challenges

There are several challenges to this research, centered on the idea of tradeoff spaces. The challenges are:

Streaming Data – Streaming data sources require nodes to be constantly online in order to handle data, which may come at any time. While some streams may be regulated in size, duration, and interval between bursts, Granules data streams cannot be expected to always conform to this ideal. This means that we need to be able to quickly detect and recover from failures to ensure that a minimal amount of data is lost.

Stateful Computations – Granules computations are capable of building state across successive rounds of execution (as more inputs are received). As Granules provides this unique capability, unlike other MapReduce (Hadoop) or graph based (Dryad) cloud computing frameworks, many Granules computations take advantage of this trait – such as epidemiological simulations and brain computer interface applications. When working with fault-tolerance in stateful environments, it is necessary to consider consistency guarantees. Computations that share the same code but have different internal state can produce different outputs for the same input data.

Consistency – As with any replication approach, consistency is a known problem. In the event of a failure, we need to ensure that a backup replica will produce an equivalent output to the original. Due to the stateful nature of Granules computations, as well as its ability to support generalized computations, we may need to support varying levels of consistency on a per computation basis.

Real-time Analysis – Several deployments of Granules require real-time guarantees [3]. As we introduce fault-tolerance, we want to make sure we continue to support real-time analysis of streaming data. This means we cannot waste time rebuilding state after a failure, and we also need to make sure that network resources are not flooded to the point where we can no longer process data in real time.

Interleaved Computations – In Granules, hundreds of computations may be hosted (and the processing corresponding to their input streams interleaved) on a single machine. This means that all these hosted computations will be directly affected by a single failure. We need to make sure that failures of machines do not have a cascading effect that may render the entire system unusable. We need to ensure that replicas are brought up in a timely manner to replace primaries that were running on the failed machine.

Extensibility – Granules is an open source project, allowing users to set up their distributed stream processing clusters and perform arbitrary computations on streaming datasets. This work needs to provide a flexible and simple interface, which allows users to easily define custom replication schemes to fit their specific computation needs. Providing the correct abstraction to allow a fully customizable fault-tolerance experience is a difficult challenge.

For each of these challenges we are looking at a tradeoff space. Networking bandwidth is limited, as is processing time. There are also tradeoffs between consistency and loss of data, customizability, and supportability.

B. Research Questions

This paper attempts to solve the following research questions:

How can we leverage existing research in the field of fault tolerance and apply this to a different type of processing system? – Both traditional distributed systems and distributed streaming databases contain a wealth of research in the field of fault tolerance. While Granules has unique characteristics that make direct application infeasible, we can still explore ways in which these proven approaches can be applied within this different paradigm.

What tradeoffs are inherent when using basic replication schemes? – Before we move on to develop a complex replication strategy to handle all possible computation needs in Granules, it is important to understand the tradeoffs inherent in basic replication schemes. Once we understand the simple models, it will be easier to understand the tradeoff space found in more complex approaches.

Can we effectively use replication to achieve fault tolerance in a generalized stream processing system? – Previously, little work has been done regarding replication in a generalized stream processing system. Given the characteristics of our streaming data, as well as the characteristics of the system, is replication a feasible approach to fault tolerance?

C. Paper Contributions

This paper provides several unique contributions to the field of fault tolerance through replication. It explores replication in the context of generalized, distributed stream processing where multiple computations are interleaved on a single machine to maximize resource utilizations. This is a significantly more complex problem than found in Distributed Streaming Databases (DSBs), which assume consistent datastreams and standard SQL queries, as well as a 1 to 1 mapping of processes and machines.

While the traditional heartbeat approach is well known in the field of distributed systems, the majority of these approaches use a single, centralized server to monitor health of the system. Our approach offers a decentralized solution that tries to balance processing and network overheads with the need to quickly and accurately determine machine failure.

We work with our heartbeat approach to derive a relationship between recovery rates and heartbeat intervals. This allows us to calculate best and worst-case recovery scenarios based off of the base values chosen for the heartbeat – we can use this information to help choose replication
scheme even before we have knowledge of the characteristics of the streaming data.

We have also performed a failure analysis of the system, showing the link between machine failures and the probability of a computation failing completely. This analysis, measures the impact of machine failures on hosted computations (in the order of 100s per machine), and allows us to evaluate our placement strategy while providing some feedback into our choice of replication level.

D. Paper Organization

The rest of the paper is organized as follows: Section II contains background information on both replication and granules, as well as how replication is implemented in Granules. Section III discusses related work, while section IV and V detail replication in Granules and our initial failover benchmarks. Section VI contains our experiments with a real-world application at a very large scale. We perform our failure analysis in section VII, we then conclude and discuss future work in section VIII.

II. BACKGROUND

Most approaches to replication can be broadly grouped into the fields of active and passive replication. These two approaches are essentially at either end of the consistency spectrum. With active replication, all replicas receive all data, so it is a fully consistent system. Passive replication ranges from inconsistent – where a backup copy will only start to receive data after the failure of the primary; eventual consistency – which can be achieved by replaying data on the node that just took over for the primary; or partial consistency – some state information was lost, while others can be rebuilt to a state which would be consistent with the primary had it not failed.

Many systems expand upon these basic approaches to develop a wide array of hybrid approaches – these help to find a balance between the pros and cons of each approach. While future versions of Granules will explore hybrid approaches, that is beyond the scope of this paper.

A. Replication Schemes

In this section we lay out the foundations for active and passive replication within the Granules generalized stream processing system. Computations will be replicated to achieve fault tolerance. For this discussion, we will be referring to the computation expected to produce output as the primary. The backup computations are replicas that are not currently responsible for producing output. Preserving a replication level of \( r \) implies that there are \( r-1 \) backups and 1 primary.

1) Active Replication

In active replication, the primary as well as all backups receive all data, and are expected to fully process all data. The only output which is sent to the next stage in the computation (or returned to the user) is the output of the primary. Active replication has a heavy processing cost, since each node is responsible for fully processing all data, as well as an increased network load, since each node needs to receive all inputs. On the other hand, active replication is very useful for computations where nodes need to maintain state – the backup nodes build their current state at the same time as the primary. Additionally, systems with active replication can switch to a backup node with very little overhead/switchover time. It is merely a matter of redirecting outputs, and can be done as soon as the system is notified that the primary has failed.

2) Passive Replication

Passive replication has a much lower processing overhead than active replication. In this approach, only a single node is active at a time (the primary) while the other nodes in the system remain dormant until notified that the primary has failed. Since the backups do not need to receive all inputs along with the primary, it also has a much lower networking overhead than an active approach. Passive replication is a great choice for computations which do not maintain state, and do not care if there is an additional ramp-up time to switch over to a replica.

B. Granules

Granules \([2, 7]\) is a lightweight runtime for cloud computing and is designed to orchestrate a large number of computations on a set of available machines. The runtime is designed specifically to support processing of data produced by sensors. Granules supports two of the most dominant models for cloud computing MapReduce \([4]\) and dataflow graphs \([8]\). In Granules individual computations have a finite state machine associated with them. Computations change state depending on the availability of data on any of their input datasets or as a result of external triggers. When the processing is complete, computations become dormant awaiting data on any of their input datasets.

In Granules, computations specify a scheduling strategy, which in turn govern their lifetimes. Computations specify their scheduling strategy along three dimensions: counts, data driven and periodicity. The counts axis specifies limits on the number of times a computation task needs to be executed. The data driven axis specifies that a computation task needs to be scheduled for execution whenever data is available on any one of its constituent datasets, which could be either streams or files. The periodicity axis specifies that computations be scheduled for execution at predefined intervals. One can also specify a custom scheduling strategy that is a combination along these three dimensions; for example, limit a computation to be executed 500 times either when data is available or at regular intervals. A computation can change its scheduling strategy during execution, and Granules enforces the newly established scheduling strategy during the next round of execution. Computations in Granules build state over successive rounds of execution. Though the typical CPU burst time for computations during a given execution is short (seconds to a few minutes), these computations can be long-running with computations toggling between activations and dormancy. Domains that Granules is being deployed in include Brain Computer Interfaces, handwriting recognition, and epidemiological simulations.

In our experiments with Brain Computer Interfaces \([3]\) we were able to process electroencephalogram (EEG) signals generated by electrodes placed on a user’s scalp in real time. Running concurrently over 10 machines, Granules was able to classify EEG signals generated at the rate of once every 250
milliseconds from 150 concurrent users in less than 70 milliseconds. Specifically, we were able to classify signals at the rate of 1 GB of EEG data every 83 seconds and about 1 TB every 23 hours.

III. RELATED WORK

Replication schemes have been explored in distributed streaming databases, such as Borealis [9] and Aurora [10]. Additionally, MapReduce frameworks such as Hadoop also implement replication. In a purely distributed systems approach, replicas are generally used to allow concurrent access. For example a user may have a working copy of a document on a mobile device which is often disconnected from the network, while a replica of the document remains on a server for other users to access [11].

In this work, we are focusing on replication as a means to achieve fault-tolerance. To achieve fault tolerance, every replicated process has a number of replicas started in different failure independent zones to ensure that at least one replica survives despite the loss of an entire zone.

MapReduce, Hadoop, and Dryad all support fault-tolerance through passive replication. Generally, once a node has failed, all processes running on that node are restarted on a new node. This process is orchestrated by a master node responsible for scheduling all computations. When a process is restarted, it starts with the original input, and replays all processing from the beginning. Essentially, this is a passive replication scheme – one in which processing power is conserved further by not even instantiating replicas until necessary.

While our approach uses slightly more resources by instantiating passive replicas, this is mitigated by a shorter failover time – no new processes need to be ramped up. Additionally, our approach leaves the opportunity open for implementing checkpointing – an approach which allows passive replicas to get state information from the primary. This means a shorter replay period is necessary before receiving meaningful output from a failed process.

IV. BASIS FOR GRANULES REPLICATION

The backbone for replication in Granules is a heartbeat system. This system constantly monitors the state of all registered machines in the cluster. In the current implementation, only aliveness is monitored. While more information, such as load and networking delays may also be useful and help to provide a robust experience, it would also lead to an increase in heartbeat message size. This means increasing the network load, increasing memory usage as every node tries to keep track of system vitals at every other node, and an increase in processing power required to generate and receive these more complex messages.

In this initial implementation, all nodes in the fault-tolerant cluster need to be added at startup – it currently does not support (re)adding nodes at a later time. This allows the system to function in a fully distributed fashion – no single node needs to orchestrate communications.

A. HeartBeat basics

For the Granules HeartBeat scheme, we introduce the idea of heartbeat groups. A heartbeat group is a subcluster of machines which sends heartbeats together, as well as checks for machine aliveness in sync. For example: in a system with two heartbeat groups, A and B, all machines in group A will send heartbeats to group B together. By implementing this heartbeat system within Granules, we can take advantage of its communications system which allows multiple machines to subscribe to a single stream of data, such as “heartbeats/groupB”, thus allowing our communications scheme to be easily implemented.

At every step, every group pushes heartbeat data to the next group, and one group is responsible for checking aliveness of the whole system. This means that while every machine is checked for aliveness every T, not every machine in the cluster is checking aliveness at timestep T.

This concept is shown in more detail in TABLE I. In this example we have six heartbeat groups – numbered 0-5. This table walks through 6 timesteps, showing where messages are sent for each timestep. For example, in timestep T3, group 4 is sending heartbeats to group 2, while group 2 is sending heartbeats to group 0. After a group sends heartbeats to group 0, it performs a check to make sure that all the nodes it has previously received heartbeats from has sent a heartbeat in the last 6 timesteps – since the last time this check took place.

An additional variable is S, the amount of heartbeats in which a node is in a state of “failure suspicion”. In this state, the machine has missed some number of heartbeats (up to $S$), but the system has not yet declared the node dead. This allows drift in clocks – a node may miss sending a heartbeat by a fraction of a second – as well as possible network congestion. As the number of nodes in the system grows, more messages need to be sent, resulting in the possibility of network congestion, which may cause delays to heartbeats.

As an example, in a cluster of $N$ machines with $M$ heartbeat groups, which has an update rate of $T$ and failure suspicion count of $S$, in a best case scenario it will take $TSM$ time in order to identify failures. In a worst case scenario, it could take $(M-1)T + TSM$ time to detect failures.

### TABLE I. THIS TABLE DESCRIBES THE HEARTBEAT APPROACH IN GRANULES WITH 6 HEARTBEAT GROUPS. FOR EACH TIMESTEP (T*), EVERY GROUP SENDS A HEARTBEAT TO ONE OTHER GROUP. AFTER SENDING A HEARTBEAT TO GROUP 0 (BOLD AND ITALICIZED), IT PERFORMS A CHECK TO MAKE SURE ALL EXPECTED HEARTBEATS WERE RECEIVED.

<table>
<thead>
<tr>
<th>$T_0$</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-&gt;0</td>
<td>0-&gt;2</td>
<td>0-&gt;3</td>
<td>0-&gt;4</td>
<td>0-&gt;5</td>
</tr>
<tr>
<td>1-&gt;2</td>
<td>1-&gt;3</td>
<td>1-&gt;4</td>
<td>1-&gt;5</td>
<td>1-&gt;6</td>
</tr>
<tr>
<td>2-&gt;3</td>
<td>2-&gt;4</td>
<td>2-&gt;5</td>
<td>2-&gt;6</td>
<td>2-&gt;7</td>
</tr>
<tr>
<td>3-&gt;4</td>
<td>3-&gt;5</td>
<td>3-&gt;6</td>
<td>3-&gt;7</td>
<td>3-&gt;8</td>
</tr>
<tr>
<td>4-&gt;5</td>
<td>4-&gt;6</td>
<td>4-&gt;7</td>
<td>4-&gt;8</td>
<td>4-&gt;9</td>
</tr>
<tr>
<td>5-&gt;0</td>
<td>5-&gt;1</td>
<td>5-&gt;2</td>
<td>5-&gt;3</td>
<td>5-&gt;4</td>
</tr>
<tr>
<td>$S&gt;$0</td>
<td>$S&gt;$1</td>
<td>$S&gt;$2</td>
<td>$S&gt;$3</td>
<td>$S&gt;$4</td>
</tr>
</tbody>
</table>

B. Effects on Replication

The HeartBeat scheme underlies all computation communications in a fault-tolerant environment. Not only can a poorly configured HeartBeat scheme impair all communications across the cluster, but the HeartBeat timestep and duration of the failure suspicion state define the amount of...
time the system needs to identify failures, and only then can fail-over steps be started.

As $T$ decreases, we check aliveness more often, which while it means the system will recognize failure and recover from it faster, it also means the network is more likely to become congested with heartbeat messages. Should the congestion rise enough to interfere with the messages getting though, delays could cause the system to emit false positives – deciding that a machine has failed when the messages were only delayed.

If we increase $T$, the amount of heartbeat messages sent throughout the cluster is decreased, which keeps the system from becoming congested and leads to less false negatives. On the other hand, it also has the obvious drawback of resulting in a proportional delay in recognizing failed machines, leading to delays in fail-over actions.

$S$ also has a strong impact on the speed of failure detection. Where $T$ determines how often HeartBeats are sent, $S$ determines how many iterations of $T$ are allowed to pass before a node is officially declared dead. The impact of $S$ is actually dependent upon $N$, the number of heartbeat groups. To clarify, a given group will do a full check of the system every $N$ timesteps. When a node has failed to send a heartbeat within those $N$ timesteps, it enters the failure suspicion state. Once within this state, it has $S$ full system checks to start responding to be declared dead. This means that it will take at least $SN$ timesteps before the node is officially declared dead. In walltime, this results in a delay of $SNT$ before fail-over actions can be taken.

V. EXPERIMENTS WITH HEARTBEAT SETTINGS

To fully understand the HeartBeat settings discussed in the previous section, we set up a simplified environment to explore HeartBeat settings with Active and Passive replication schemes.

A. Sample Computation

The sample computation for our experiments with HeartBeat settings is a simple Flip computation. In this computation, an input string is flipped from lowercase to uppercase. This sample computation has no state, and is a computation, an input string is flipped from lowercase to uppercase.  This sample computation has no state, and is a computation, an input string is flipped from lowercase to uppercase.

The heartbeat settings we are testing are an $M$ of 6 (with 24 nodes this means 4 nodes per group), $T$ of 2s, and $S$ of 2. In the best case, the failover time should be 24s, with a worst case time of 34s. We will be generating messages once every 10ms, to give us a good sense of timing. In our experiments we kill up to 5 nodes, showing that the cluster can survive over 20% failure and continue running.

C. HeartBeat Results

For these tests, we generated inputs every 10ms and pushed them out to be processed. When a message has been processed, it generates a reply, which contains the message sequence number. In our results below, we show the number of messages missed when a machine has failed, which directly correlates to the amount of time it took to notice and handle failure. Messages are being sent every 10ms, so 100 missed messages amounts to approximately 1s of missed data.

![Figure 1. Number of messages missed by Active and Passive replication schemes with 1-5 failed nodes.](image)

The results from our base experiments are shown above in Figure 1. We have additionally shown both best and worst case scenario values as well. An interesting note about the worst case is that once 4 nodes have been removed from the network, it is possible that an entire heartbeat group has failed, meaning that there is a potential extra $T$ to wait for the failure to be noticed.

| TABLE II. AVERAGE LOST MESSAGES WITH 1-5 FAILURES INITIAL BENCHMARKS |
|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|
|                        | 1 Failed              | 2 Failed              | 3 Failed              | 4 Failed              | 5 Failed              |
| Active                 | 2859.2                | 2799.8                | 2782.5                | 2851.2                | 2857.3                |
| Passive                | 2859.5                | 2800.1                | 2782.6                | 2851.3                | 2857.8                |
| Best                   | 2400                  | 2400                  | 2400                  | 2400                  | 2400                  |
| Worst                  | 3400                  | 3400                  | 3400                  | 3400                  | 3600                  |

While the active and passive failovers are very close on average, about 20% of the time, the passive failover missed one extra message than its active counterpart. This is more clearly shown above in TABLE II. This small difference in the average number of lost messages that we are seeing is the cost of redirecting streams. Where an active replica is already listening for inputs, a passive replica needs to set up streaming channels.
A. Dataset

For our full-scale experiments with replication in Granules, we work with a respiration dataset [12, 13] which was collected by Dr. J. Rittweger, at Institute for Physiology, Free University of Berlin. In this dataset, thorax extension was monitored at 10Hz. This data is a series of doubles, which our system streams every 100ms, to match the original 10Hz frequency.

Thorax extension directly relates to breathing rates – it monitors the rise and fall of the patient’s chest while breathing. This information can be used on both the larger scale – is the patient still breathing on their own? To a more refined scale – is the patient awake or asleep? By analyzing thorax extension, it is possible to determine which stage of sleep the patient is in.

This type of data can be used to monitor patients just out of surgery, or even to conduct sleep studies. Thorax extension data can be used to infer not only well-being, but state of mind of the patient as well. It is important to note that in these tests we want to ensure that responses are returned in real time (before the next set of output is pushed out, or within 100ms) as well as ensure that the computations continue to run even in the face of failure.

B. Experimental Setup

For this set of experiments, we used the same settings for N, M, T, and S as before: 24 nodes (N), 6 groups (M), a timestep (T) of 2s, and a death suspicion count (S) of 2. By using these well-understood values, we can focus on the interactions of replicas in a large-scale setting.

Our full-scale experiments are run with 2.4 GHz quad-core machines running Fedora 14 with 12GB RAM and a gigabit network connection. We are pushing 2400 computations to these machines and then shutting down communications to one machine at a time. We are not only looking for the number of lost transmissions (each is numbered), but also for the amount of additional time the passive instances need to open up communications channels as well as rebuild state.

In this example, we are working under the assumption that a respiratory monitoring system has been deployed in a hospital using Granules as the cloud computing runtime. We are using a very simple computation which simply stores recent values (such as if it was generating a continuous graph), as well as statistical information such as a moving average, min and max values observed, which could be useful for a graphing front-end. Because of the nature of the state information we are keeping, the passive approaches are almost eventually consistent. Given enough time, the recent values and moving average will eventually converge – and be exactly the same as an uninterrupted computation, while the maximum and minimum seen values may never reach a converged state.

In this hypothetical hospital situation, a different replication level may need to be specified on a per-patient basis. For example: a patient just out of surgery may require and active replication scheme, while it may be acceptable to have a passive replication scheme for a patient in for overnight observation. To keep to a realistic setting, each instance of the computation has its own designated input stream.

VI. EXPERIMENTS WITH RESPIRATORY DATA

A. Dataset

For these experiments we set up 800 monitoring applications, 10% of which were active. This means that there are 100 computations on every machine in the cluster, for a total of 2400 computations. In these tests we again kill 5 machines and determine the effects on replication, primarily how much data was lost in the recovery process. While this is a simple example, it is a realistic simulation, it is run at full-scale. With 100 respiratory computations on each machine, as well as the HeartBeat system, we are forcing Granules to interleave computations to handle processing 8,000 inputs per second.

C. Respiratory Results

With our setting of 10% active calculations, we have a system with 80 active deployments and 720 passive deployments. These deployments are spread among 24 machines, with our deployment operation ensuring that a computation is on a separate machine from its replicas.

With data being streamed only 10 times a second (as opposed to our small-scale test), our resolution for overheads is a little lower. The average number of lost messages is displayed above in TABLE III. As you can see, the passive versions are more likely to have variation due to the need to set up the communications.

<table>
<thead>
<tr>
<th>AVERAGE LOST MESSAGES WITH 1-5 FAILURES</th>
<th>RESPIRATORY DATASET</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Failed</td>
<td>2 Failed</td>
</tr>
<tr>
<td>Active</td>
<td>278.0</td>
</tr>
<tr>
<td>Passive</td>
<td>287.4</td>
</tr>
<tr>
<td>Best</td>
<td>240</td>
</tr>
<tr>
<td>Worst</td>
<td>340</td>
</tr>
</tbody>
</table>

The above table displays the number of lost messages with failures going up to just over 20% of the machines in the group. It is important to note that the recovery of a single machine means not the failure of a single computation, but the failure of 100 different computations.

The thorax extension data is sent out every 10 seconds, so to convert from messages missed to recovery overheads, we simply need to divide by 10. In the table above, we see that at most we lost 30.7 seconds to recovery – which is still below our worst case scenario of 34 seconds. One interesting feature is the fact that we saw increasing overheads up to 4 failures, but with the fifth failure we actually had an improved recovery time. As we saw in TABLE I. there is a rotating schedule to failure checks. With the fifth failure, we have an increased response time simply due to the time it takes to secure communications.

While these results show that the amount of time to secure the communications channel with passive replication is a rare event, there is an extra penalty to passive approaches not shown in these results. In our experimental computation, we are maintaining state in the form of the last 10 seconds of input. The active replicas, on the other hand, are receiving input the entire time, and continue to build state even if it is not passing on the results.

From these results we also learned a bit about other options to help promote fault tolerance. For example, if we opted to store the last few seconds of output to ‘replay’ in the face of failure, our experiments show that we would need to store the
last 30 or 31 seconds of output to ensure absolutely no inputs are lost.

VII. FAILURE ANALYSIS

With 800 simultaneous computations, it is difficult to keep track of how many replicas from a single computation have failed. This section is devoted to a probabilistic discussion of the probability of losing all replicas of a computation. Below we look at both the probability of a computation failing entirely given an individual machine failure rate, as well as the number of lost computations as machines fail.

A. Individual Computation Failure

For the purposes of this discussion, we are going to assume that the probability of machine failure is independent. Essentially, we are saying that the probability of machine A failing does not relate to the probability of machine B failing. While this is not necessarily true in a live situation: if machine A and machine B are in the same rack, or were purchased at the same time, there is usually a higher relation between failures. While rack-awareness can reduce the occurrence of the first problem, the second is harder to systematically avoid. If, however, the cloud is large enough the probability of replicas being hosted on a set of machines bought at the same time approaches zero.

If our computation has a replication level of 3, it will be spread across 3 machines: A, B, and C. For this experiment, let’s assume that each machine has an X% chance of failure – this includes all hardware, as well as network connections. We are also assuming that the failure probabilities are independent between machines. This means that the overall probability of failure of the entire computations is: \( P(A_{\text{fail}}) * P(B_{\text{fail}}) * P(C_{\text{fail}}) \).

To illustrate this point, we have created a table below to show how computation failure relies on X.

<table>
<thead>
<tr>
<th>X</th>
<th>1%</th>
<th>5%</th>
<th>20%</th>
<th>50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure Rate</td>
<td>0.0001%</td>
<td>0.0125%</td>
<td>8%</td>
<td>12.5%</td>
</tr>
</tbody>
</table>

From TABLE IV, we can see that even with catastrophic machine failure rates, we see a very low probability of losing a specific computation entirely. While this is a relatively simplistic view, through rack awareness and sheer numbers, it is possible to assume that we can reduce this problem to the form of \( P(A_{\text{fail}}) * P(B_{\text{fail}}) * P(C_{\text{fail}}) \), where every probability is independent. Once we get to this point, the probability of complete failure quickly decreases. Should we implement the ability of the system to re-replicate, e.g. add a replica on machine D should machine A fail, then the probability of complete failure should decrease even further.

B. Computation Failure Rate

Above we explored the probability of losing a specific computation given machine failures, now we look at the probability that computations will fail should X machines fail. For this section, we are assuming that there are U unique computations, and N total machines. We are again assuming a replication level of 3.

While there are U unique computations, there are actually 3U computations in the system total (due to the replicas). This means that each machine will have about 3U/N computations on it. As a note, we are assuming that the machines are equally loaded. For both a best and worst-case scenario, replicas for at least 3U/N computations are all co-located – essentially, machines A, B, and C would be loaded with exactly the same set of computations. In the worst-case scenario, 3U/N computations would fail after just 3 machines failed. In a best-case scenario, 2U computations could be lost before a unique computation failed entirely. Since there are 3U/N computations per machine, this means that we can still run when: 3U/N * X = 2U, or when X = 2/3 * N machines fail. This means that when the 2/3N + 1 machine fails, we will lose computations completely.

Looking into this behavior a bit more deeply, we can come up with the expected average failure given X, N and U. For X failed machines, any given computation will be failed with a chance of X/N. Since each unique computation is replicated 3 times, this needs to be raised to the cube – since all 3 replicas need to fail for the computation to fail, the complete computation failure is \( (X/N)^3 \). To find the average number of failures, we multiply by the number of unique computations: \( (X/N)^3 * U \).

To test our theory, we reran our experiments with the respiratory dataset. This time, we are looking at the number of failed unique computations given the number of failed machines. As a reminder, we have 800 unique computations (U), and 24 machines (N), meaning each machine has 3U/N, or 100 computations running on it. In this experiment, we are looking at the number of failed unique computations after killing 5, 8, 12, and 18 machines. Using the equations above, we can show the theoretical best, average, and worst cases alongside the experimental, best, average and worst cases. We ran each test 10 times, and recorded the number of computations lost.

<table>
<thead>
<tr>
<th>Machines Failed</th>
<th>Predicted</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Average</td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>7.23</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>29.63</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>100.00</td>
</tr>
<tr>
<td>18</td>
<td>200</td>
<td>337.50</td>
</tr>
</tbody>
</table>

From TABLE V, we can see that we manage to hit the best case scenario in every experiment, and we usually fall below the worst case. We had almost no losses when almost a quarter of the machines had failed, and we still maintained 50% functionality even after 75% of the cluster had died. Even in the case of catastrophic failure, we are seeing functional behavior. We are seeing increasing standard deviation in failure amounts up to the point where we lose 50% of the network – this seems to be a combination of the fact that computations are usually lost on the order of 100s at a time, as well as the fact that there is an increasing gap between best and worst cases, leaving more room for variation.

VIII. CONCLUSIONS AND FUTURE WORK

We have implemented a scheme for the fault tolerant processing of real time data streams. We have abstracted the scheme that underpins this failure resiliency. A developer
should be able to simply specify a replication scheme without needing to implement it from scratch.

Our tests in this work scaled up to 800 computations with a replication level of 3 across 24 machines and we had 100 computations per machine. In this work we were able to estimate the number of lost messages for a given replication scheme based on the characteristics of the heartbeat interval. Our experiments have shown that this derivation is flexible enough to work with both active and passive replication schemes. As we build on this work and incorporate support for more complex recovery schemes, we will be able to use this algorithm to define the upper and lower bounds on the performance of each new approach.

We have additionally developed a method to estimate best, average, and worst case scenarios for computations given machine failures. This allows us to make informed decisions about the number machines to supply in a cluster, as well as replication rates. In our experiments we found that even given catastrophic failure rates (up to 75% of the cluster was shut down) our approach allowed half of the computations to continue functioning normally.

One avenue of future work is to implement checkpointing within the passive replication scheme. This would cut down on the amount of time needed to rebuild state after failure within the passive scheme. While this would cut down on recovery time, checkpointing can potentially slow down the primary – the primary would need to pause in operations, pass along state information to the replicas, and wait until all nodes have confirmed receipt of state information. This also has the potential to lead to network congestion if all passive primaries attempt to send state information to their replicas all at once.

Another approach is to implement replaying. This is a process where we temporarily store inputs. From our experiments, it looks like we would need to store at least the last 30s of data to ensure no inputs were lost in the face of failure. While this is a straightforward solution with our respiratory dataset, it is unclear how we could define this buffer with a dataset which is more bursty in nature. How much space would we need to reserve for a buffer when the data is not generated at a regular interval?

We also plan to work towards making active replication more efficient. With a stateless, deterministic operation, it is possible to use active replication to process data in parallel. Since each replica already needs to process all data, we can split the input between the replicas and process the data in parallel. If one node fails, the data can be redirected to the remaining nodes. This would help alleviate the processing costs of active replication.

Additionally, we are assessing the possibility of developing a hybrid approach. This approach would allow a user to specify different replication schemes for different stages of a computation. For example, a mission critical stage of the computation may be actively replicated, while an analysis component may be passively replicated. This could lead to more efficient resource usage, and could allow a greater number of concurrent users to be supported.

Another avenue for further research is to look into leveraging the approach most often found in traditional distributed systems. With active replication, and the right type of computation, there’s no reason we couldn’t use the active replicas in parallel to improve processing speeds.

While extending the replication approaches available to the user is a major goal of our future work, we also plan to look into applying machine learning approaches in order to implement autonomous tuning of the parameter space. While we have explored part of the parameter space in this work, there is still a lot of area we have not covered. Once we have implemented checkpointing, there will be an even more variables for fine-tuning. There is also a very good chance that certain types of computations benefit most from slightly different parameter tweaks. If we allow autonomous tuning, then it is possible that a live cluster can adopt different replica parameters to adapt to the set of computations currently running on it, and modify these parameters, if needed, as more computations are added.

REFERENCES