Failure-Resilient Real-Time Processing of Health Streams

Kathleen Ericson, Shrideep Pallickara, C. W. Anderson*

Computer Science Department
Colorado State University
Colorado, USA

Abstract

The ability to analyze streaming data in real-time is vital in systems that process data from health sensors. These systems need to build and maintain state, as well as preserve this state in the face of system failures. In this work, we introduce a fault-tolerance scheme designed for the Granules stream processing system. We work with two distinct health stream datasets: thorax extension and electroencephalogram (EEG) signal analysis. We have developed a monitoring program to track trends in the thorax extension dataset, and a classification system for the EEG dataset which allows us to determine user intent from EEG signals. Using these two motivating applications, we have explored several approaches to fault-tolerance through replication, developing a hybrid approach that is particularly suited to health streams.

Keywords: Sensors, Health Monitoring, EEG, BCI, Distributed Systems, Granules, Replication, Fault-tolerance

1 Introduction

Sensor networks are becoming ubiquitous as sensors become smaller, cheaper, and more power efficient. These sensors are deployed across a wide range of environments, from monitoring for early earthquake detection [1] to smart condos [2] and personal health monitoring [3–9]. Sensors produce data intermittently or at regular intervals, creating *data streams* that are a set of correlated packets. The data generation rates may spike due to external stimuli, e.g. increased tectonic activity or a person entering a room, sensors may generate data at a higher rate, leading to bursty behavior.

Sensors designed to monitor health are a standard in the diagnosis and monitoring of many health conditions. As the cost and size of these sensors decreases, the feasibility of monitoring patients as they go about their daily lives increases. Should a patient have multiple live sensors, the data gathered from these disparate devices can be combined to gain a clearer picture of overall patient status. As we have come to rely on sensors and technology for our well-being, we must also cope with failures in the backend machines where the streams are processed.

Health sensors generate data streams that must be processed in real time; typically, packets need to be processed faster than the rates at which they are produced. Processing such data streams is challenging because the generation may be periodic, intermittent, or bursty. Inability to process streams in a timely and accurate fashion can lead to injury, so it is important that we continue processing streams even in the face of failures. Care must be taken to avoid *overprovisioning* where the majority of the resources would idle most of the time, or *underprovisioning* where processing requirements outpace available resources. The data generation in several cases is *long-running* (taking days or weeks) meaning that the computations that operate on them must also be long running.

Our runtime, Granules is designed for processing such data streams generated by sensors [10,11]. To maximize resource utilization, the runtime interleaves the execution of several stream processing computations on the same machine. The runtime allows each computation to have a dynamic scheduling strategy where they can be scheduled for execution periodically (at intervals specified in milliseconds) or when data is available. A given stream processing computation can have multiple rounds of execution and retain *state* across each round of execution. In each

*Email: {ericson, shrideep, anderson}@cs.colostate.edu

round of execution, the computation operates on the available data, updates its state, and then becomes dormant awaiting activation when data is available.

Issues relating to state retention and interleaving of stream processing computations at a resource compound the difficulty of failure resilient processing of data streams. All data is lost from the time a computation fails to the time a new computation can be instantiated and connected to the appropriate streams. Even temporary failures can mean the loss of irreplaceable data. Computations build state over time and the outcome of processing a stream packet relies on this retained state – the same packet may result in different outputs depending on the retained state. For example, consider a computation monitoring electrocardiogram (ECG) signals. Where a steady increase in heartrate while exercising is normal, a sudden increase in heartrate may be a sign of a heart attack. Should such an ECG monitoring computation lose state while a user was exercising, an erroneous heart attack alert may be generated. The problem of losing data is only compounded as several stream processing computations are interleaved on any given machine, a failure at any one machine impacts **all** its hosted computations.

One approach to cope with failures is to build redundancy into the system via *replication*. The trade-off space for any scheme that relies on replication for processing these streams in the presence of failures involves the following dimensions: (1) network I/O, (2) processing overheads, (3) memory utilization, (4) system throughput per cluster measured in terms of packets that were processed per unit of time excluding any duplicate processing that might be performed on copies of a packet, (5) speed of failure recovery, and (6) the amount of state loss.

Consider the extreme ends of this trade-off spectrum. A brute force, or *active* approach that relies on maintaining r active replicas of a computation results in an r-fold increase in the network and processing footprint; this approach provides the fastest recovery with no loss of state. At the other end of the spectrum, we could have an approach that trades off state for network and processing efficiency; a *passive* approach. Upon failure detection a computation is launched without any built-in state with the expectation that over time state will build up and eventually converge as a result of the stream processing.

In this paper, we explore the trade-off space that accompanies fault-tolerant stream processing. We present an empirical evaluation of our schemes using real health stream datasets.

1.1 Challenges

Failure resilient real time stream processing is challenging because:

- Streams arrive continually and the arrival rates may be bursty. Processing must be timely and faster than the rate at which data is generated.
- Stream processing computations are stateful, with processing decisions being made based on the state built over time. For a given input, the output may be different depending on the state that has been built up within the computation.
- State loss may be inevitable when failures occur. Mitigating such state losses is important.
- Given the behavior of streams, we interleave multiple computations on the same machine to maximize resource utilizations. This also means that the failure of a single node will impact multiple computations and lead to corresponding state loss.

Health streams tend to be long-running and the arrival patterns over streams can be dynamic. While these are long-running streams, the typical amount of data which needs to be processed at each timestep is relatively small. Given their arrival patterns, setting aside a resource per stream would lead to underutilization of that resource. This necessitates the need to interleave multiple computations on a single machine.

To accurately process health streams, it is important to build and maintain state. For example, consider a patient who is wearing a gyroscope reporting whether the wearer is standing upright or not. Any computation processing the data from this sensor needs to be able to differentiate between a person taking a nap and someone who has just fallen down a flight of stairs. Data sent to a failed node for processing is lost, so quickly detecting and recovering from failures is necessary to ensure the safety of those relying on our system.

To achieve scalability, we need to interleave multiple computations on a single machine. This means that the failure of a single machine will directly affect all these computations. Failures

may also have rippling affects, where downstream computations will be affected by the loss of data from failed upstream nodes. Furthermore, if recovery measures are not well-orchestrated, failure recovery measures may end up overloading other nodes, resulting in cascading failures.

To the best of our knowledge, no system has tried to address recovery in such an environment. Other distributed databases [12–16] do not rely on stateful computations for processing, and previous work in the realm of health stream processing [3–9] do not explore solutions at scale, and do not address fault-tolerance concerns at all.

1.2 Paper Contributions

This work brings several unique contributions to the field of fault-tolerance in stateful stream processing systems.

- We have developed several novel approaches to support fault-tolerant behavior.
- We have explored the trade off space in failure resilient stream processing.
- We have developed a framework to mitigate state loss during failures in situations where the processing is built on online-based classifications.
- We allow users of the system to instrument the degree of failure security.
- Our empirical evaluations are done with real health stream data.
- Our benchmarks demonstrate the feasibility of using our system for a variety of health streams.

The proposed schemes represent an exploration of the trade-off space involved in failure resilient processing of data streams. Our approach is aligned with the characteristics of stream processing that involve inputs arriving continually either in bursts or at regular intervals over a long period of time. Retrofitting the traditional file-based checkpointing schemes to such stream processing computations would have involved writing state to a network file system; the accompanying I/O operations not only introduce additional processing overheads but also introduce a bottleneck in the form of the file server that is responsible for managing such checkpoints.

We first devised a hybrid scheme where replicas of a computation are dispersed over a collection of machines. The replicas for a computation are passive in the sense that they do not receive copies of data stream packets on which they perform redundant stream processing. However, these replicas perform state synchronization at regular intervals allowing any one of the replicas to seamlessly take over primary processing responsibilities during a resource failure affecting the primary computation. Our benchmarks demonstrate that even with a large number of computations hosted per machine, we can sustain multiple random failures in the cluster before a stream processing computation becomes unavailable.

Our approach allows systems architects to choose a strategy that best fit their needs. If some loss of state is acceptable then the hybrid scheme provides the best trade-off in terms of the network and processing footprint. Our refinement of the hybrid replication scheme is suitable for computations that encompass both state and online learning from the data streams.

Unlike other approaches where fault-tolerance decisions are not accessible to developers, our scheme is fully configurable, allowing users to configure the number of replicas for a computation, the length of the periods between state synchronization, and the heartbeat interval to detect failures as a precursor to promoting a passive replica.

We have evaluated the efficacy of our failure resilient stream processing in the realm of health stream processing with several health stream datasets. We first use a thorax extension dataset [17, 18] collected by Dr. J. Rittweger, at Institute for Physiology, Free University of Berlin to explore this trade-off space. Thorax extension data can be used to monitor respiration rates, which is useful for sleep studies as well as the monitoring of pre- and post-operative patients. This dataset is representative of a class of single output health streams, where a relatively small amount of data is generated at a steady rate. While this is a trivial problem with only a single computation running on a single machine, it quickly becomes complex as computations for a hundred users are interleaved on a single machine.

We also look at a much more complex problem: analyzing electroencephalogram (EEG) streams in real time. This task represents computations that encompass online learning systems

i.e. systems that learn from the data they see. A broad class of health care applications require state build-up and perform assorted online learning on the streams to deliver targeted care and improved outcomes. For this set of experiments we focus on analyzing EEG streams, a major component of Brain Computer Interfaces (BCIs). BCIs allow users to interact with their environment using only their thoughts. BCIs can be used for computer interactions, such as a speller [19], moving a cursor [20], or even to navigate a wheelchair through a crowded room [21].

EEG streams are exemplars of complex health streams. For example, EEG streams are generally very noisy. EEGs are gathered in a non-invasive manner, so signals are only captured after traveling through the meninges and scalp. This means they are very weak and dispersed when first collected. External noise can also be a problem: when recording indoors, it is possible to see a relatively high magnitude 60 Hz component due to electrical currents in the environment (in the US). EEG signals can also be completely overwhelmed by noise from muscle movements: larger movements such as eye blinks, and even less obvious rapid eye movements may generate noise. The amount of data produced can also vary drastically depending on the amplifier used to collect the signals. High end amplifiers may have as many as 64 or 128 electrodes and sample at a rate of 1024 Hz. While these amplifiers are currently very expensive (roughly $60,000), we can expect these costs to decrease. Other amplifiers, such as g.Tec's g.mobilab+ are designed to be small, mobile, and inexpensive. They may have only 8 electrodes and a sampling rate of 256 Hz.

One goal of Colorado State University's Brain Computer Interfaces Lab (`http://www.cs.colostate.edu/eeg`) is to provide a cost-effective BCI framework driven by analyzing raw EEG data. We build on this idea by pushing EEG analysis to a cluster with multiple users being analyzed concurrently. Not only is this a more efficient use of resources than having a single dedicated system per user, but it allows allows aggregation of raw EEG data. Such aggregation has the potential for analysis on a much broader scale over a much shorter time period than any single lab would be able to collect. While our empirical evaluations are in the context of a Brain Computer Interface application that relies on Artificial Neural Networks for classifying actions performed by a user, our approach is general enough to be applicable to computations that rely on other schemes for online learning such as SVMs, Random Forests, and Hidden Markov Models.

Our approach is flexible enough to handle a variety of health streams, as shown by our experiments with both thorax extension and EEG datasets. The rate at which the sensors produce data, as well as the amount of data produced for each timestep varies greatly between these two datasets. Health stream monitoring systems need to support not only one extreme or the other, but a mixture of such data streams in a single cluster.

While interleaving computations means our approach is significantly more scalible than previous approaches, it also introduces a new problem: interference. Collocated computations are vying for the same set of limited resources, and this problem is compounded when collocated computations activate at the same time. We have extended our replication scheme to detect interference, and use this to alleviate performance problems such as increased turnaround times due to resource contention and overutilization leading to delays in processing. It is also dynamic in the sense that the roles (active or passive) taken on by the replicas continue to evolve in response to system conditions such as performance hotspots, resource overloading, and failures.

1.3 Paper Organization

The rest of this paper is organized as follows: in section 2, we describe all the tools used in this work, as well as introduce the EEG dataset in more detail. Section 3 describes the framework of our fault-tolerance schemes, along with tests outlining failure detection performance and an analysis of failure probabilities using our thorax extension dataset. We then examine our fault-tolerance approach with the more complex EEG dataset in section 4. After this we discuss EEG/health stream specific fault tolerance approaches in section 5 before exploring interference in 6. We then move on to related works in section 7 and then concluding in section 8.

2 Background

2.1 Granules

Granules [10, 11] is a distributed stream processing system designed to perform arbitrary computations on streaming data. Computations build and maintain state across multiple rounds of execution, entering a dormant state with a reduced resource footprint between these rounds of execution. With Granules users may specify computation specific scheduling constraints across three dimensions: number of iterations, time interval, or as data is available. This is particularly useful for monitoring health sensor streams where a computation should activate whenever the sensor sends data to be processed. If no updates are received within a given interval, possibly due to sensor failure or user emergency, an alternate branch of the computation can be accessed either alerting of a sensor failure or initiating an emergency response to aid a user.

Due to the ability to enter a dormant state between rounds of execution, Granules is capable of interleaving hundreds of computations on a single machine. While this means better utilization of resources, this also means that the failure of a single machine impacts multiple computations, making fault-tolerance a necessary and challenging task.

2.2 Fault-Tolerance in Granules

Here we propose fault-tolerance schemes for use in the Granules runtime. Our current focus is fault-tolerance through the use of replication. In systems that do not process streaming data it is possible to simply restart a computation in the face of failure. If this approach is applied to a streaming environment there is a risk of: *(1)* losing data while detecting failure and starting up a new computation and *(2)* loss of state. In many cases it is pointless to restart a sensor-based computation from a blank state. To reduce the amount of data lost, backup computations may be started beforehand to maintain a stateful copy of the computation on different machines. The cost of such an approach is an additional strain on the cluster as we now need to keep extra copies, or *replicas*, of a computation running at all times. Depending on the type of replication used, we may be increasing network, memory, and CPU load with every replica.

An alternate approach to replication is *checkpointing*. In this approach, processing is periodically frozen as state information is saved. While this can be accomplished in a streaming environment [16] by buffering data that arrives during the checkpointing process, it is not a good solution for health stream monitoring where delayed responses may mean injury to a user.

2.3 R Programming Language

R is an interpreted language designed to perform matrix computations efficiently, making it an ideal language for neural network code. The code we use here has been well-tested and used in previous BCI experiments [22].

We have two distinct R applications: a user instance and a generic trainer instance. There is one user instance for every user connected to the system. User instances are responsible for classifying streaming EEG signals and updating the model when new training data is sent to it. The generic trainer instance performs initial batch training from disk, as well as further batch processing of any newly submitted user data on a regular schedule. The trainer provides its generic group of experts to new users as they join the cluster for classification.

In order to handle communications between R and the Java-based Granules framework, we use a bridging framework developed for Granules [23].

2.4 Artificial Neural Networks

Artificial Neural Networks (ANNs) are biologically inspired machine learning algorithms. They are capable of modeling and classifying complex signals, so they are a good choice for analyzing raw EEG signals. Here we use a group of experts approach where we train several ANNs independently, then use their combined results to determine a final prediction. Since the networks are trained independently, there is a good chance they have all 'learned' to model slightly different aspects of the problem space. We combine the predictions of this group of ANNs to achieve a better overall prediction than a single neural network.

To avoid over-fitting, we limit the amount of training each network receives. While this seems counter-intuitive, we can actually achieve better results by doing this as it ensures that the network does not develop a bias towards the training data and can still process new, unseen data. This approach, called early stopping, also allows us to train the networks more quickly and efficiently, cutting down on the time it takes to initially train a group of experts as well as the amount of time it takes to update when a user sends new training data.

2.5 Thorax and EEG Datasets

In our experiments with the thorax extension dataset, all computations are performed in Java. We keep a running queue containing the last 10 seconds of data, as well as general statistics about the data seen so far. To explore checkpointing needs, we assume that the statistical information needs to be propagated to all replicas, while the last 10 seconds of data are an acceptable loss in the face of failure.

We classify EEG signals generated by an able-bodied adult male, gathered by the CSU BCI lab. Four tasks are recorded from the user: imagined right hand movement, imagined left leg movement, counting backwards from 100 by threes, and imagining a spinning computer. The dataset is separated into five different recording sessions, where each recording session has 10 five second recordings for each of the four tasks. We use four of our five datasets for training, reserving the fifth for testing. This allows us to evaluate the accuracy of our trained networks by ensuring that only unseen data is sent to be classified.

The EEG data was gathered using a NeuroPulse Mindset 24R amplifier with 19 electrodes arranged in the international 10-20 specifications and a sampling rate of 512 Hz. In general, more electrodes can lead to a more fine-grained view of brain activity, which in turn makes it easier to determine exactly where EEG signals are originating and how the signals are propagating across the scalp. On the other hand, more electrodes also means more data that needs to be processed at every timestep, which can potentially slow down the classification process and lead to an overburdening of the communications network.

For our EEG analysis, we use an R-based [24] classifier as the backbone of the system. While R is a good language for our classification tasks, it is designed to be run in a single thread, which complicates attempts to host multiple user computations on a single node. If we only support one user per node, nodes are underutilized and we cannot scale well. R contains several packages to distribute computations [25, 26], but we previously [22] found that the time to initiate distributed computations precludes the ability to process data in real-time.

A further disadvantage of R is the lack of support for tablet devices. Amplifiers such as the g.mobilab+ rely on a bluetooth connection to transmit data from the amplifier to a recording device such as a tablet computer or smartphone. This design allows a much more mobile EEG collection experience than previously possible, a benefit for BCI applications such as a wheelchair navigation. Ideally, a user would only need a tablet or smartphone to handle EEG collection and classification. R is currently unsupported on Android, iPhones, and iPads. This leaves the majority of tablet devices useless in this setting. These devices are all capable of running Narada Brokering [27], allowing them to connect to a Granules cluster for classification.

3 Fault-Tolerant Stream Processing

A heartbeat system underpins replication in Granules. This system monitors the state of all registered machines in the cluster, allowing the system to detect failures in a fully distributed fashion – no single node needs to orchestrate communications. While information such as processing load and networking delays may also be useful and help to provide robust behavior, we are currently only monitoring liveness, ensuring a compact heartbeat message size to control the CPU, memory, and networking footprint of the HeartBeat scheme.

3.1 HeartBeat Groups

In our HeartBeat scheme, we introduce the notion of heartbeat groups. A heartbeat group is a subcluster of machines that send heartbeats together, as well as checks for machine liveness in sync. For example: in a system with two heartbeat groups, A and B, all machines in group A

Table 1: 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.

T0	**T1**	**T2**	**T3**	**T4**	**T5**
$0 \rightarrow 1$	$0 \rightarrow 2$	$0 \rightarrow 3$	$0 \rightarrow 4$	$0 \rightarrow 5$	$\boldsymbol{0 \rightarrow 0}$
$1 \rightarrow 2$	$1 \rightarrow 3$	$1 \rightarrow 4$	$1 \rightarrow 5$	$\boldsymbol{1 \rightarrow 0}$	$1 \rightarrow 1$
$2 \rightarrow 3$	$2 \rightarrow 4$	$2 \rightarrow 5$	$\boldsymbol{2 \rightarrow 0}$	$2 \rightarrow 1$	$2 \rightarrow 2$
$3 \rightarrow 4$	$3 \rightarrow 5$	$\boldsymbol{3 \rightarrow 0}$	$3 \rightarrow 1$	$3 \rightarrow 2$	$3 \rightarrow 3$
$4 \rightarrow 5$	$\boldsymbol{4 \rightarrow 0}$	$4 \rightarrow 1$	$4 \rightarrow 2$	$4 \rightarrow 3$	$4 \rightarrow 4$
$\boldsymbol{5 \rightarrow 0}$	$5 \rightarrow 1$	$5 \rightarrow 2$	$5 \rightarrow 3$	$5 \rightarrow 4$	$5 \rightarrow 5$

will send heartbeats to group B in the same timestep. This approach is particularly suited to 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".

At every timestep T, each group pushes heartbeat data to the next group, and one group is responsible for checking liveness of the whole system. While every machine is checked for liveness every T, not every machine in the cluster is checking liveness at the same time t.

This concept is shown in more detail in Table 1. 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 timesteps 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 for drift in clocks, where a node may miss sending a heartbeat by a fraction of a second, as well as possible network congestion. This also helps to limit the number of false positives, or erroneous failure notifications, generated by the cluster.

Consider 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.

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.

As T decreases, liveness checks are performed more often. While this does mean the system will recognize failure and recover from it faster, but it also means the network is more likely to become congested with heartbeat messages. Should the congestion interfere with the messages getting through, 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 drawback of 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 before being 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.

3.2 Failure Analysis

This section is devoted to an analysis of failure rates given our HeartBeat scheme. 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.

3.2.1 Individual Computation Failure

For the purposes of this discussion, we are going to assume that the probability of machine failure is independent; i.e., the probability of machine A failing does not relate to the probability of machine B failing. This is not necessarily true in cases where machines on a rack share the same power strip. Rack-awareness in replica placements can alleviate this.

Our computations have a replication level of 3, spread across 3 machines: A, B, and C. For this experiment, we assume that each machine has an $X\%$ chance of failure. This includes all hardware, as well as network connections. The overall probability of failure of an entire computations is: $P(A_{fail}) * P(B_{fail}) * P(C_{fail})$. For a machine failure rate of 1%, the complete failure of a computation has a probability of only 0.0001%.

Even with high machine failure rates (such as 50%), we see a very low probability of losing a specific computation entirely (only 12.5%). 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 $P(A_{fail}) * P(B_{fail}) * P(C_{fail})$, where every probability is independent. Once we get to this point, the probability of complete failure quickly decreases. Implementing the ability of the system to re-replicate, e.g. add a replica on machine D should machine A fail, reduces the probability of complete failure even further.

3.2.2 Computation Failure Rate

We now look at the probability that computations will fail should Y machines fail. For this section, we are assuming that there are U unique computations, and N total machines with a replication level of 3.

With U unique computations, there are actually $3U$ computations in the system in total (due to the replicas). Assuming that the machines are equally loaded, each machine will have about $\frac{3U}{N}$ computations on it. For both a best and worst-case scenario, replicas for at least $\frac{3U}{N}$ 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, these $\frac{3U}{N}$ 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 $\frac{3U}{N}$ computations per machine, this means that we can still run when $\frac{3U}{N} * Y = 2U$, or when $Y = \frac{2}{3} * N$ machines fail. This means that when the $\frac{2}{3}N + 1$ machine fails, we will lose unique computations completely.

Looking further into this behavior, we can derive the expected average failure given Y, N and U. For Y failed machines, the probability of any given computation failing is $\frac{Y}{N}$. Since each unique computation is replicated 3 times, all 3 replicas need to fail for the computation to fail. The probability for the complete failure of a computation is $3\frac{Y}{N}$. To find the average number of failures, we multiply by the number of unique computations: $3\frac{Y}{N} * U$.

To test this, we work with a respiration dataset [17, 18] which was collected by Dr. J. Rittweger, at Institute for Physiology, Free University of Berlin. This dataset monitors thorax extension at 10Hz. To simulate a live dataset, we stream the inputs every 100ms, matching the original 10Hz frequency.

Thorax extension directly relates to breathing rates, monitoring the rise and fall of a patients chest while breathing. This information can be used on both a large scale "is the patient still breathing on their own?" to a more refined scale "is the patient awake or asleep?" It is further 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. In these tests we want to ensure that responses are returned in real time (before the next set of output is pushed out,

Table 2: Predicted and actual computation losses as machines fail

	Predicted			Actual			
Machines Failed	**Best**	**Average**	**Worst**	**Best**	**Average**	**Worst**	**SD**
5	0	7.23	100	0	0.10	1	0.32
8	0	29.63	200	0	40	200	69.92
12	0	100.00	400	0	90	200	73.79
18	200	337.50	600	200	330	400	67.49

or within 100ms) as well as ensure that the computations continue to run even in the face of failure.

Thorax extension data is shares several characteristics with EEG analysis. We need to build state over time to properly analyze thorax extension, and timeliness is an important factor when determining patient status. We use thorax extension data in these experiments because the data sent is much smaller than the typical EEG packet, allowing us to more easily test replication guarantees in large deployments.

In this set of experiments, we are looking at the number of failed unique computations given the number of failed machines. We use 24 nodes (N), 6 groups (M), a timestep (T) of 2s, and a death suspicion count (S) of 2. We deploy 800 unique computations (U) to this cluster, meaning each machine has $\frac{3U}{N}$, 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 randomly selected 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, recording the number of computations lost.

From Table 2, we see that we managed to hit the best case scenario in every experiment, and we usually stay 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. An interesting trend we found was an increase in standard deviation up to losing 50% of the network, after which the standard deviation began to decrease again. This seems to be a combination of losing computations on the order of 100s at a time, as well as the increasing gap between best and worst case scenarios, leaving more room for variation. By the time 75% of the network has been lost, the likelihood of all replicas of a computation being lost are much higher – this means that there is a chance no burst of communications can take place to activate a passive replica.

4 EEG Experiments

After an initial analysis of our fault-tolerance approach with the respiratory dataset, we move on to working with our EEG dataset. EEG data is more complex than the thorax extension dataset, involving many more sensors simultaneously generating data. EEG data is also typically produced at a much higher rate. The data we are working with is generated at 256Hz, while the thorax extension dataset is generated at only 10Hz. In order to gain more temporal insight from the EEG data (as well as to avoid overloading the network), we send out EEG data for processing every 250ms. While this slows down the rate of transmission of EEG signals below that of the respiratory dataset, it also increases the amount of data sent out for processing.

4.1 Small Cluster Testing

It is important to determine our maximum support capabilities at a smaller scale before introducing the extra communications overheads inherent in larger clusters. First, we determine the maximum number of users we can stably support on a single machine, then we move to a small cluster of 3 machines to test our ability to support fault-tolerance on this scale.

Table 3: Response Times for 30 Concurrent Users on a Single Node (ms)

Mean (ms)	**Min (ms)**	**Max (ms)**	**SD (ms)**
23.600	7.026	484.669	15.419

Table 4: Response Times for 35 Concurrent Users on a Single Node (ms)

	Mean (ms)	**Min (ms)**	**Max (ms)**	**SD (ms)**
Overall	26.292	6.993	9564.874	97.953
Passing	23.225	6.993	249.852	17.172
Failing	1283.038	250.638	9564.874	1497.543

4.1.1 Experimental Setup

For these experiments, we are using nodes with 2.4 GHz quad-core processors and 12 GB of RAM. Each node hosts a Granules Resource [10, 11] which manages all computations on the machine. The resource is connected to a stream routing broker [27] which resides on another identical machine. EEG signals are pseudo-streamed from a third identical machine which is responsible for recording round-trip classification times.

The generic trainer is deployed first, and immediately begins batch training from the training sets. Once it has finished, the user computations are launched and receive an initial, generic group of experts from the trainer. For our fault-tolerance experiments, we additionally launched a `HeartBeat Listener` on each node. With only 3 nodes, we set M to 3. This means that each node is within its own heartbeat group. In these tests we are focusing on a best-case scenario, so no further training is performed. Our goal is to determine the maximum number of users we can support while ensuring that classifications are returned in a timely manner.

A user cannot be supported when classifications fail to return before the next segment of EEG signals are sent out. As we are classifying 250ms streams, when responses take longer than 250ms we consider them failed classifications.

We launch a number of individual users who independently stream EEG signals and record round-trip classification times. These signals are sent out every 250ms, but only after a classification for the previous timestep has been returned. This means that if a classification fails (takes more than 250ms to return to the user), the next signal is sent only after receiving the previous classification. While this is not what would occur in a live scenario this helps to prevent network congestion and makes round-trip timing of individual EEG streams easier to analyze, both of which are beneficial when stress-testing the system. Each user submits 5000 EEG streams for classification, roughly 21 minutes of continuous EEG signals.

4.1.2 Single Machine Stress Tests

For this test we focus on a single machine which has a single generic trainer instance. We are attempting to determine the maximum number of concurrent users we can support before we start to breach our 250ms real-time guarantee. Initially, we attempted to support 30 individual users – a significant increase over the maximum 17 users found in our previous experiments [22]. With these settings, we found that only one message out of the 150,000 recorded failed to return to the user in time (Table 3).

On further analysis, we found that this failed message was among the first ones sent by a user. This message was probably delayed due to initialization overheads, a very likely case given that it was not a lasting problem. In the next test we added 5 extra users; with 35 users, we saw an increase in the number of failed messages, from 1 to 426 failed messages, or a failure rate of 0.2% instead of 0.0006%. Overall statistics can be seen in Table 4. In our best case, we can return results in just under 7ms, but in the worst case, responses took over 9.5 seconds. In the same table, we also show a break down for the passing and failing responses.

Analyzing the probability density of the passing response times (Figure 1), we do see some promising trends. The majority of passed classifications return to the user in under 50ms. While we do have a worst-case scenario of 9.5 seconds, most of our computations are well within the

Table 5: Response Times for 40 Concurrent Users on a Single Node (ms)

	Mean (ms)	**Min (ms)**	**Max (ms)**	**SD (ms)**
Overall	33.982	6.762	30565.040	298.482
Passing	23.487	6.762	249.961	18.605
Failing	2112.003	250.176	30565.040	3651.625

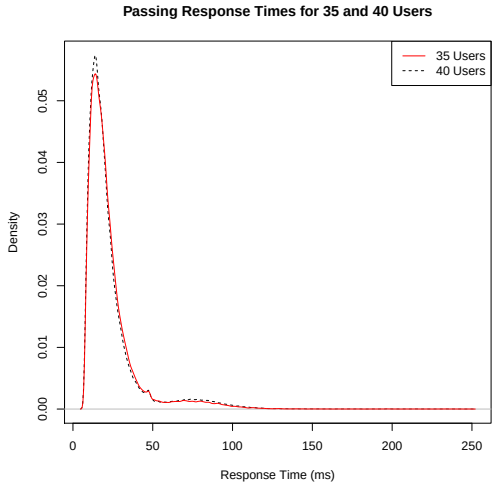

Figure 1: Density functions of passing response times in milliseconds for 35 and 40 users on a single node (ms)

passing range.

Our overall failed classification rate was still relatively small, at only 0.2%. To stress the system even further, we decided to run one more test with 40 users per machine. The results for this experiment are shown in Table 5. We again saw an increase in failures: from 426 to 1005 failed messages, while the failure rate increased from 0.2% to 0.5%. While this is still a very small value, the number of failures has more than doubled when only adding 5 additional users. We again decided to look at a breakdown of the response times of the failed and passing computations, shown in Table 5.

Figure 1 shows the probability densities of passing response times for both 35 and 40 users. Looking closely, we see very similar response times. With 40 users, there seems to be more of a smaller, secondary clustering of response times between 50ms and 100ms, possibly showing a small secondary wave of classifications. This could be a sign that incoming data streams are getting clustered together, leading to computations being processed in waves.

We saw a small increase in the percentage of failures between 35 and 40 users, but also a drastic increase in the amount of time it takes for these delayed messages to get back to the sender; increasing from 9.5 seconds to 30 seconds. Performing a closer analysis of these failed messages, it becomes clear that they are occurring in waves: all clients report overdue communications at approximately the same time. In the 35 user case, these waves occur less often than in the 40 user case. There is obviously some recurring event causing messages to be overdue.

Analyzing resource usage on the machine, it becomes clear that delayed messages occur when data needs to be shifted in and out of swap space. The node is utilizing all 12GB RAM maintaining the dedicated R instances, and has needed to start storing data which is actively needed in swap.

Looking at only the percentage of failed messages, it seems likely that we would be able to support even more concurrent users. In the case of BCI applications, however, timeliness is a priority. For example, you would not want to be using a system which may have a 30 second delay when using EEG signals to drive a wheelchair. Being stuck in the middle of a street for

half a minute could be disastrous. Due to the drastic difference in response times as swap needs to be utilized, we decided to set a cap at 35 users on a single machine. Our current bottleneck is memory usage, so in future work we can look into ways to decrease the footprint of R.

4.1.3 Passive and Active Fault-Tolerance Schemes

For our initial experiments in fault-tolerance for EEG streams, we first looked at the simplest possible case: 3 resources with 30 users hosted on each. While this is a bit below the maximum support case we found for small clusters (35 users), we are introducing extra communications in the form of the heartbeat approach. Each resource was in its own failure group ($M = 3$), has a failure suspicion level (S) of 2, and a transmission rate (T) of 2 seconds. Based on this information, and the algorithms defined in section 3, we can predict best, worst and average case scenarios with respect to how long it takes to recognize and recover from failures.

Full Passive Replication Experiments

In a fully passive approach, only the primary replica receives inputs and generates outputs. The other replicas simply remain dormant until failure of the primary has been detected. At that time one of the remaining replicas is promoted to primary status and inputs are then redirected to the new primary. While this approach has the lowest cost to maintain with respect to resource usage, it also has the highest cost with respect to the amount of time it takes to recover. Once a passive replica detects failure of the primary, it needs to initiate communications streams. For this experiment, we implemented a resend message which allows the replica to request a resend of the last data from the user. This allowed us to measure the time it takes to recognize, recover, and start processing new data after failures.

As we can see from Table 6, the passive replication recovery rate is trending towards the theoretical worst case scenario. In one instance, we are even exceeding this worst case scenario. Our approach limits the potential for flagging false positives (labeling that a failure has occurred when one has not), at the cost of increasing the time to notice failure. In a fully passive approach, new channels of communication need to be set up in order to resume the processing of data, leading to potential recovery times above the theoretical worst case scenario.

Full Active Replication Experiments

For this set of experiments, we are setting all computations in the cluster as active replicas. In our implementation, this means that all replicas receive all inputs, but only the primary is responsible for processing the data and generating a result to pass on to the user. In short, we are pushing three times as much data for inputs as we would in an unreplicated environment. With 30 users, we would originally be generating data at a rate of 2.3MB/s, but since all replicas need to see all inputs, we are instead generating data at a rate of 7MB/s.

In our initial experiments, we relied on the replicas saving state from the previous inputs to recover from failures. This should have resulted in an increased recovery time from failure, since they do not need to request a resend. With this approach, we actually found our recovery time to be far worse than the worst case scenario of 16 seconds. By having all replicas store the last EEG signal sent to it, the node was forced to store computations in swap space, leading to much larger overheads when a failure occurred. We switched to the model we used in the passive replication approaches, where a replica requests a resend of data from a client when it is promoted to primary. As seen in Table 6, this new approach leads to a recovery time right around the theoretical average.

While this approach can offer the strongest fault-tolerance guarantees, it also incurs the greatest overheads. In a live system, a fully active replication approach is too naive, as we begin to hinder our scaling capabilities.

4.2 Full Scale Cluster Stress Tests

Previously, we found our network to be an effective bottleneck at 150 concurrent users. In section 4.1, we found that memory started to became the bottleneck on a single node without replication at 35 users. This section determines the maximum number of concurrent users we can

Table 6: Time to recover from failure in a small cluster with 30 concurrent users (ms)

	Mean (ms)	**Min (ms)**	**Max (ms)**	**SD (ms)**
Active	14740.98	14609.79	14864.52	72.657
Passive	15898.75	15794.50	16023.19	67.414
Theoretical	14000	12000.00	16000.00	—

support as we scale up the number of nodes in our cluster. As the number of users increases, we increase the chances that the network becomes a bottleneck as communications increase. Ideally, we should be able to maintain the rate of 35 users per machine.

4.2.1 Changes in Approach

For these experiments, we needed to switch to a more streamlined approach to generating EEG signals due to a lack of system resources. Using a threaded approach, data is pushed every 250ms regardless of whether or not a previous response has been returned. When the delay is long enough, this can lead to lost messages. In such situations, we are unable to properly asses worst case scenarios.

4.2.2 Full Scale Stress Tests

As a baseline starting point we first look to the question of supporting 1000 concurrent users. Due to the current cost of amplifiers, this is far beyond the rate at which EEG signals are typically gathered by a single lab. Supporting this many concurrent users means that we can support multiple EEG labs and their user base on a single cluster. Amalgamating this much data from disparate users could allow us to learn much more about raw EEG data than ever before.

We spread these 1000 computations across 40 machines. While this will undershoot our findings from the previous section (25 users per machine instead of the 35 maximum we found before), this allows us some leeway to take into account any problems that may arise from a networking standpoint as we scale up.

We found that we could support 1000 users with a minimal failure rate (0.005%). A closer analysis of the data revealed that our worst-case scenario involved a maximum delay of just over 1 second. As we discussed in section 4.1, this length delay is unlikely to be noticed by a user, so falls within an acceptable limit. Looking at the probability density of passing responses in Figure 2, we see that there is a significant shift in response times from our tests with a single node. This makes it apparent that we are starting to see a problem with communications overheads as we scale up our experiments.

The next step we took was to see if we could support the 35 users per machine we saw in the previous tests. We used the same setup for this test: 40 resources, a dedicated broker, and external machines to generate EEG streams. We distributed 1400 computations, so each resource hosted 35 computations.

In this test case, we saw a significant increase in the number of failed classifications, reaching 0.8% failure rate. This includes lost messages as well as messages received in over 250ms. Looking at the 1400 user case in Figure 2, we can see that there has been a drastic shift in the mean response times. We are obviously hitting a communications threshold as we scale up to a full cluster.

Even on a full scale cluster, our new approach can support a much larger number of concurrent users than our initial approach. We also found that we are hitting a communications bottleneck as we increase the size of our cluster. The main problem appears to be with the underlying communications framework. One avenue of expansion is to develop a more efficient method of content distribution.

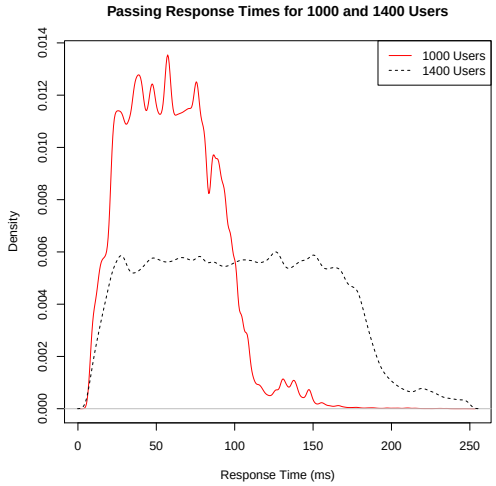

Figure 2: Density functions of passing response times in milliseconds for 1000 and 1400 users on a cluster of 40 nodes (ms)

4.2.3 Replication in a large Cluster

For the last set of experiments in this section, we decided to analyze how our replication approach scales. We hosted 450 unique user computations. With a replication level of 3, this means 1350 computations in total. We hosted this on a cluster of 45 nodes, so each node hosted 30 computations. It is important to remember at this point that while Java can recognize the difference between a primary and a replica, R cannot. Even passive replicas can take up a significant portion of memory keeping their designated R instance alive.

With respect to our HeartBeat framework, we set up our test environment to have 5 heartbeat groups each containing 9 machines. This leaves our best and worst case scenarios at a 20 and 28 seconds respectively. While we could have used only 3 message groups as in our small-scale experiments, this would add to the probability of encountering communications bottlenecks, leading to erroneous failure announcements.

In this experiment we utilize an 80/20 split of passive and active replication schemes (360 passive and 90 active computations) as a cluster containing only active replicas causes too much stress on the underlying communications framework to be feasible. The ability to support multiple replication schemes in a single instance of a framework is not common ([14, 28–30]), but a strength we have built into the Granules framework.

In the context of BCI, it is clear that not all computations are created equally. A computation performing classifications for a BCI controlled wheelchair should have stronger failure guarantees than a BCI speller. Granules' ability to host various replication approaches in a single cluster allows for much more flexibility in deployments.

As in our previous stress tests we needed to use a modified generator with a threaded approach. In the case of failures, we can only see the number of missed messages. As messages are sent every 250ms, every second spent detecting failure means 4 lost messages. This means in the best case scenario we will lose 80, while in the worst case we could lose 112 messages.

In our tests we found that our passive and active approaches both lost the same number of messages, right at the theoretical average of 96 lost messages. This is most likely caused by the lockstep nature of our generator approach. The cost to set up communications in a passive approach only takes tens of milliseconds to occur. Unless the timing is perfectly aligned, the odds of missing one of the regular 250ms bursts is relatively low.

5 EEG Specific Failure Resilient Stream Processing Schemes

While our HeartBeat scheme does allow us to accurately determine whether or not a machine has failed, the cost of this approach is readily apparent in the amount of time needed to recover from a failure. The purpose of this section is to explore several different approaches to help reduce the amount of time a client is left without any new classifications. A key to these approaches is the ability of the client to raise an alert if they are having problems contacting their computation. Giving users the power to determine failure suspicion opens up several complex fault-tolerance schemes, allows for basic load-balancing techniques, and even opens up the possibility of detecting a new class of failures.

5.1 Multi-User Classifiers

Our current approach involves each user training up an individualized group of experts in their own R instance. Previously, we looked at a much more generalized approach where all users shared a single group of experts. While the accuracy of a generic group of experts will be lower than an individually trained group of experts, it should be better than leaving the user without any classifications.

This approach involves starting a single, unreplicated generic classifier on every machine in the cluster. The generic classifiers would be tuned to listen for computations which do not have any replicas hosted on the same node, as that would provide no additional support. While evaluating various approaches to host groups of experts for BCI applications, we considered using generic classifiers to host multiple users. Our experiments showed that we could potentially support up to 40 users within a single generic classifier.

When a user has begun to miss classifications, they can begin to simultaneously transmit data to a generic classifier. This way a user will not be left without any classifications while waiting for the failure of a node to be confirmed with the HeartBeat approach.

5.2 Dual Processing of Inputs

In stream processing systems some research has gone into exploring how replicas can reduce latency [13], this approach could also be adapted to our process of classifying EEG signals. This is a resource-intensive approach where multiple replicas receive, process, and generate outputs. Clients are required to keep track of sent messages, so they can determine whether they are receiving a straggler or new classification.

Overall, this is a simple fix with a high return – a user never even notices when a replica has failed as they continue to receive classifications from the other replicas. This would be an ideal strategy for a BCI such as the wheelchair where even small outages may result in disastrous consequences. Whether or not this outweighs the extra cost in resource usage can be a more difficult question to answer. Using our replication strategy of hosting 3 replicas, this would lead to lowering a clusters capacity to $\frac{1}{3}$ of what it could otherwise host.

One possible solution to this resource usage is to use a hybrid approach: 2 replicas are concurrently processing inputs and returning results, while the third exists solely as a passive replica. Should one of the active replicas fail, the passive one would then be promoted to an active role. All this could occur without the user even noticing that a failure has occurred.

A big problem in this approach is the slim but not non-existent possibility that a second failure could occur causing the user to lose both active replicas before the passive replica can be instantiated. This problem can be solved by the addition of the generic classifier approach introduced above.

While waiting for the passive replica to acknowledge the failure and be promoted to active status, the user can make use of a generic classifier (possibly giving preference to results returned from the remaining active replica). Should both active replicas fail before the passive replica can be promoted, the user would be able to rely entirely upon the generic classifier – ensuring some processing of EEG data is occurring.

5.3 Toggling Replicas

An alternative approach is to give users even more control over their replicas by revealing computation hosting options. Instead of staying with a single replica until failure, users switch between replicas periodically. Based on performance and user requirements, the user can choose which replica to send the bulk of their data for processing. This approach has several advantages:

- **Load Balancing** – over time, users will settle down to primarily use replicas residing on nodes with the lowest loads. This will allow the system to keep itself load balanced over time.
- **Meeting User Needs** – based on BCI application, users may have very different requirements. Some may prioritize lower latency, while others need to limit variations in response times. Users can make informed decisions about which replica to rely on based on previous behavior.
- **Reduced Overheads** – this approach uses less resources than the dual processing approach, yet should still allow a user to detect failure more quickly than a naive approach.
- **Increased Knowledge Dispersal** – as users are regularly switching between replicas, different nodes in the cluster will be acting as primary over time. This means that different generic trainers will have access to new training data from this user over time. Each node in the cluster will obtain broader access to training data, potentially increasing the capabilities of the models developed by the generic trainers.

This approach does, however, require a lot of processing and memory usage on the client side. An approach this complex may not be a good choice for a mobile device such as a smartphone with limited resources to begin with. This approach should only be implemented after careful consideration of various parameters. Should the toggle function be timed incorrectly, there is a chance that the system will never reach a stable state if all users sync up unfortunately.

5.4 Determining Computation Level Failures

The HeartBeat system has been designed to detect failures at a machine level. It is tuned to avoid false positives, and so it tends to err on the side of caution taking seconds to ensure a failure has actually occurred. While we have been focusing on the task of detecting failure more quickly, we have been avoiding the more difficult to detect computation level failure.

This is a failure which does not effect other computations on the machine (such as the HeartBeat computation), meaning it can never be detected by anyone other than the single user connected to that instance. By allowing a user to independently switch to a different replica when responses fall below an acceptable threshold, we solve the problem of partial failures.

6 Interference

In our stress tests we overloaded the cluster with identical computations and continuously pushed data to the cluster for classification. This is a worst-case scenario with respect to interference: computations with identical footprints and resource needs are being activated continuously very closely together.

From our previous experiments, we see congestion occurring as we overload the system, leading to failed responses. While some of this is undoubtedly due to network congestion, interference between computations is likely exacerbating the situation. To relieve interference, computations which are activating at the same time can be shifted to different machines. This can be easily accomplished using our passive replication scheme. If the machine that the primary is on is overloaded, one of the other replicas can be promoted to primary and shift the load to a different machine. This is very similar to the idea of allowing a user to toggle replicas. The main difference is that instead of switching between replicas regularly, we are essentially migrating the primary replica between machines as needed.

6.1 Interference Exploration

In this experiment we pushed 24 user computations with a replication level of 3 to a small cluster of 3 machines. All computations were passive in order to take full advantage of interference detection and relief. We are currently only looking at machine load and queue delays. On the active replica, we gather information about the time data spends waiting in a queue for processing. All replicas have the ability to query the host machine as to current CPU load. When the user notices responses falling below a predefined threshold, it first checks the queue delay on the primary – this helps determine if the problem is because the system is overloaded, in which case shifting processing to another machine would be beneficial; or if the problem is simply in network congestion, in which case a shift in processing may cause more problems. Once it has determined that a shift in processing should occur, the user can then has all replicas query for the status of their host machines. The user then chooses the replica with the lowest overheads to host the primary computation.

We first ran our basic stress test to determine a baseline passing rate. Given that all the primaries are collocated on a single machine, we expected to see some failures of computations. We saw a failure rate of 8.26%, a rate much worse than we saw in our previous experiments. Next, we ran a round of classification with our new user code, with interference detection and relief abilities. We then ran a final round of classification with the original stress testing code, finding that we had reduced the failure rate to 6.22%, a decrease of over 2%. By providing more information to the user, we should be able to expand on this code to achieve greater improvements more efficiently.

7 Related Work

Replication schemes have been explored in distributed streaming databases, such as Borealis [12] and Aurora [14]. MapReduce frameworks such as Hadoop also implement replication. In distributed systems, replicas may also 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 [13].

In this work, we focus on replication as a means to achieve fault-tolerance. Every replicated process has replicas located in different failure independent zones to ensure that at least one replica survives even catastrophic failure.

MapReduce [29], Hadoop [28], and Dryad [30] all support fault-tolerance through active replication. Once a node has failed all processes it was hosting 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 at the beginning with the original inputs. 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. Reducing failover time is very important when dealing with streaming inputs, particularly bursty inputs where data is not entering the system at a consistent rate.

Replication may also be leveraged for correctness [31] in a scheme where a central node awaits outputs from all replicas to decide on a consensus result of a computation. This reduces the impact of malicious or erroneous results, and is a good way to handle computing in an untrusted environment.

Databases typically make use of replication to allow quick recovery in the case of failure conditions. There are two main approaches to replication updates: *Eager*: all nodes need to apply an update before it is considered completed, or *Lazy*: only the primary applies the update before reporting success. These approaches can be loosely classified as active and passive approaches, respectively [32]

Classifying different mental tasks is a common approach for BCI applications [21,33]. There have been several studies [34] to determine appropriate mental tasks. In this work we develop ANNs that determine which task a user is most likely performing. Other work attempts to simulate EEG data for each task and then compare these simulations to actual data to determine which task a user is currently performing [35].

Other approaches for BCI include identifying Event Related Potentials (ERPs) such as the P300 [19]. A P300 is a very strong signal which can be seen in an EEG recording that occurs

roughly 300ms after an uncommon target stimulus has been seen. This approach is often used in spellers, where random letters are flashed as the user attempts to spell a word.

Error-related potentials (ErrP) have been found to be beneficial to BCI applications as well [36]. These occur about 250 ms after the 'wrong' action has occurred. For example, a BCI user may attempt to move a cursor from one side of the screen to the other. If an incorrect classification is chosen by the BCI system (e.g. the cursor moves left instead of right), an ErrP can be detected in the users EEG. Utilizing this signal, it is possible to allow a user to give real-time feedback on the accuracy of an underlying classifier.

8 Conclusions and Future Work

Our experiments represent the first time that real-time EEG classifications have been performed for such a large number of concurrent users. In this work, we have presented a scalable approach to EEG analysis for BCI applications. This approach allows for the aggregation of massive amounts of EEG data, opening the way for new analyses. Furthermore, we do so without limiting the ability for users to customize their classifiers to obtain greater accuracy on demand. We managed to obtain throughput rate of 1TB every 4 hours.

In distributed systems, failures are common. When working with health sensor data, it is important to be able to detect and recover from failure in a timely manner. Here we developed a failure detection scheme with low false positive rates. We have designed approaches to failure detection and recovery that allow the user to play an active role in failure detection. Allowing users to enter a failure suspicion mode makes it possible to take action to mitigate possible failures before the system can reliably determine that a failure has occurred. This allows us to detect single computation failures where most computations on the node are still functioning. Such failures may go undetected on a system which only determines complete node failures.

Our small scale results showed a bottleneck due to the memory requirements of R. One avenue of future work involves trying to reduce this footprint. Alternatively, we may move to a different backend language for EEG classification. Python's NumPy library [37] offers comparable classification overheads, and has reduced communications overheads with Granules Bridges [23].

Acknowledgement

This research is supported by a grant from the US National Science Foundation's Computer Systems Research Program (CNS-1253908).

References

[1] T. Rui, X. Guoliang, C. Jinzhu, S. Wen-Zhan, and H. Renjie, "Quality-driven volcanic earthquake detection using wireless sensor networks," in *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, 2010, pp. 271–280.

[2] E. Stroulia, D. Chodos, N. M. Boers, H. Jianzhao, P. Gburzynski, and I. Nikolaidis, "Software engineering for health education and care delivery systems: The smart condo project," in *Software Engineering in Health Care, 2009. SEHC '09. ICSE Workshop on*, 2009, pp. 20–28.

[3] F. Camous, D. McCann, and M. Roantree, "Capturing personal health data from wearable sensors," in *Applications and the Internet, 2008. SAINT 2008. International Symposium on*, 2008, pp. 153–156.

[4] C. Chung-Min, H. Agrawal, M. Cochinwala, and D. Rosenbluth, "Stream query processing for healthcare bio-sensor applications," in *Data Engineering, 2004. Proceedings. 20th International Conference on*, 2004, pp. 791–794.

[5] H. Fei, X. Yang, and H. Qi, "Congestion-aware, loss-resilient bio-monitoring sensor networking for mobile health applications," *Selected Areas in Communications, IEEE Journal on*, vol. 27, no. 4, pp. 450–465, 2009.

[6] I. homed, A. Misra, M. Ebling, and W. Jerome, "Harmoni: Context-aware filtering of sensor data for continuous remote health monitoring," in *Pervasive Computing and Communications, 2008. PerCom 2008. Sixth Annual IEEE International Conference on*, 2008, pp. 248–251.

[7] A. Milenkovi, C. Otto, and E. Jovanov, "Wireless sensor networks for personal health monitoring: Issues and an implementation," *Computer Communications*, vol. 29, no. 1314, pp. 2521–2533, 2006.

[8] H. Schuldt and G. Brettlecker, "Sensor data stream processing in health monitoring," ETH Zrich, Technical Report 422, October 2003 2003.

[9] A. Wood, J. Stankovic, G. Virone, L. Selavo, H. Zhimin, C. Qiuhua, D. Thao, W. Yafeng, F. Lei, and R. Stoleru, "Context-aware wireless sensor networks for assisted living and residential monitoring," *Network, IEEE*, vol. 22, no. 4, pp. 26–33, 2008.

[10] S. Pallickara, J. Ekanayake, and G. Fox, "An overview of the granules runtime for cloud computing," in *IEEE International Conference on e-Science*, Indianapolis, USA, 2008.

[11] ——, "Granules: A lightweight, streaming runtime for cloud computing with support for map-reduce," in *IEEE International Conference on Cluster Computing*, New Orleans, LA, 2009.

[12] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the borealis stream processing engine," in *Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, 2005.

[13] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly available, fault-tolerant, parallel dataflows," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. Paris, France: ACM, 2004, pp. 827–838.

[14] D. J. Abadi, D. Carney, U. etintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[15] J. G. Elerath, A. P. Wood, D. Christiansen, and M. Hurst-Hopf, "Reliability management and engineering in a commercial computer environment," in *Reliability and Maintainability Symposium, 1999. Proceedings. Annual*, 1999, pp. 323–329.

[16] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik, "Scalable distributed stream processing," in *Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA USA, 2003.

[17] J. Rittweger, "physiodata," 2000, ed: Institute for Physiology, Free University of Berlin.

[18] K. Eamonn, "Hot sax: Efficiently finding the most unusual time series subsequence," L. Jessica and F. Ada, Eds., vol. 0, 2003, pp. 226–233.

[19] D. J. Krusienski, E. W. Sellers, F. Cabestaing, S. Bayoudh, D. J. McFarland, T. M. Vaughan, and J. R. Wolpaw, "A comparison of classification techniques for the p300 speller," *Journal of Neural Engineering*, vol. 3, no. 4, p. 299, 2006.

[20] G. E. Fabiani, D. J. McFarland, J. R. Wolpaw, and G. Pfurtscheller, "Conversion of eeg activity into cursor movement by a brain-computer interface (bci)," *Neural Systems and Rehabilitation Engineering, IEEE Transactions on*, vol. 12, no. 3, pp. 331–338, 2004.

[21] F. Galan, M. Nuttin, E. Lew, P. Ferrez, G. Vanacker, J. Philips, and J. d. R. Millan, "A brain-actuated wheelchair: Asynchronous and non-invasive brain-computer interfaces for continuous control of robots," *Clinical Neurophysiology*, vol. 119, no. 9, pp. 2159–2169, 2008.

[22] K. Ericson, S. Pallickara, and C. W. Anderson, "Analyzing electroencephalograms using cloud computing techniques," in *IEEE Conference on Cloud Computing Technology and Science*, Indianapolis, USA, 2010.

[23] K. Ericson and S. Pallickara, "Adaptive heterogeneous language support within a cloud runtime," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 128–135, 2012.

[24] "The r project for statistical computing," p. R project homepage, 2010. [Online]. Available: http://www.r-project.org

[25] L. Tierney, A. J. Rossini, N. Li, and H. Sevcikova, "Snow: Simple network of workstations," 2009.

[26] J. Knaus, "snowfall: Easier cluster computing (based on snow)," 2010.

[27] S. Pallickara and G. Fox, "Naradabrokering: a distributed middleware framework and architecture for enabling durable peer-to-peer grids," pp. 41–61, 2003.

[28] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, 2009.

[29] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *ACM Commun.*, vol. 51, pp. 107–113, 2008.

[30] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, Lisbon, Portugal, 2007.

[31] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, ser. GRID '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–10. [Online]. Available: http://dx.doi.org/10.1109/GRID.2004.14

[32] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Understanding replication in databases and distributed systems," in *20th International Conference on Distributed Computing Systems, 2000. Proceedings.*, 2000, pp. 464 –474.

[33] J. R. Millan and J. Mourino, "Asynchronous bci and local neural classifiers: an overview of the adaptive brain interface project," *Neural Systems and Rehabilitation Engineering, IEEE Transactions on*, vol. 11, no. 2, pp. 159–161, 2003.

[34] M. C. Dobrea and D. M. Dobrea, "The selection of proper discriminative cognitive tasks – a necessary prerequisite in high-quality bci applications," in *Applied Sciences in Biomedical and Communication Technologies, 2009. ISABEL 2009. 2nd International Symposium on*, 2009, pp. 1–6.

[35] E. M. Forney and C. W. Anderson, "Classification of eeg during imagined mental tasks by forecasting with elman recurrent neural networks," in *Neural Networks (IJCNN), The 2011 International Joint Conference on*, 2011, pp. 2749–2755.

[36] P. W. Ferrez and J. del R. Millan, "Error-related eeg potentials generated during simulated brain-computer interaction," *Biomedical Engineering, IEEE Transactions on*, vol. 55, no. 3, pp. 923–929, 2008.

[37] T. E. Oliphant, "Python for scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 10–20, 2007. [Online]. Available: http://link.aip.org/link/?CSX/9/10/1