Dissertation

Robust Health Stream Processing

Submitted by

Kathleen Ericson

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall 2014

Doctoral Committee:

Advisor: Shrideep Pallickara

Daniel Massey
Daniel Turk
Charles Anderson

Abstract

Robust Health Stream Processing

As the cost of personal health sensors decrease along with improvements in battery life and connectivity, it becomes more feasible to allow patients to leave full-time care environments sooner. Such devices could lead to greater independence for the elderly, as well as for others who would normally require full-time care. It would also allow surgery patients to spend less time in the hospital, both pre- and post-operation, as all data could be gathered via remote sensors in the patients home. While sensor technology is rapidly approaching the point where this is a feasible option, we still lack in processing frameworks which would make such a leap not only feasible but safe.

This work focuses on developing a framework which is robust to both failures of processing elements as well as interference from other computations processing health sensor data. We work with 3 disparate data streams and accompanying computations: electroencephalogram (EEG) data gathered for a brain-computer interface (BCI) application, electrocardiogram (ECG) data gathered for arrhythmia detection, and thorax data gathered from monitoring patient sleep status.

Acknowledgements

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

This dissertation is typset in LaTeX using a document class designed by Leif Anderson.

Table of Contents

Abstract ii

Acknowledgements iii

List of Tables vii

List of Figures viii

Chapter 1. Introduction 1

1.1. Monitoring Health Stream Data 1

1.2. Challenges in Robust Processing of Health Streams 2

1.3. Research Contributions 4

1.4. Dissertation Organization 6

Chapter 2. Background 7

2.1. Evaluating Replication Schemes Across Distributed Systems 7

2.2. Replication in Distributed File Systems 14

2.3. Replication in Cloud Runtimes 23

2.4. Replication in Distributed Hash Tables 35

2.5. Replication in Distributed Stream Processing Systems 42

2.6. Discussion 52

Chapter 3. Requirements 56

3.1. Computation Support 56

3.2. Real-time Requirements 57

3.3. Robust to Failures 58

3.4. Robust to Interference 60

3.5. Throughput 61

Chapter 4. Components 63

4.1. Datasets 63

4.2. Sample Computations 64

4.3. System Elements 67

4.4. Communications 70

4.5. Summary 72

Chapter 5. Detecting Failures 74

5.1. Introduction 74

5.2. Approach Synopsis 79

5.3. Fault-Tolerant Stream Processing 79

Chapter 6. Replication 83

6.1. Failure Analysis 83

6.2. BCI Experiments 86

6.3. Leveraging Replicas for BCI 98

6.4. Summary 102

Chapter 7. Detecting Interference 104

7.1. Defining Interference 104

7.2. Clustering Interference 105

7.3. Clustering Overheads 107

7.4. Intelligent Placement 110

Chapter 8. Migration 112

8.1. Soft Migration 112
8.2. Hard Migration 114
8.3. Summary 116
Chapter 9. Contributions and Future Work 118
9.1. Conclusions 118
9.2. Contributions 120
9.3. Future Work 123
Bibliography 127

List of Tables

2.1 A summary of the distributed systems discussed in this paper 52

5.1 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. 81

6.1 Predicted and actual computation losses as machines fail 86

6.2 Response Times for 30 Concurrent Users on a Single Node (ms) 88

6.3 Response Times for 35 Concurrent Users on a Single Node (ms) 89

6.4 Response Times for 40 Concurrent Users on a Single Node (ms) 89

6.5 Time to recover from failure in a small cluster with 30 concurrent users (ms) 93

6.6 Re-Replication Overheads 94

7.1 Small-Scale Clustering as Hosted Computations Increase (ms) 109

7.2 Large-Scale Interference Clustering as Hosted Computations Increases (ms) 110

8.1 Small Scale BCI Migration Overheads 116

List of Figures

2.1 Flow of communications for GFS 18

2.2 Flow of communications for HDFS 20

2.3 Flow of communications for MapReduce, Hadoop, and Dryad 26

2.4 Flow of communications for BOINC 34

2.5 Flow of communications for latency-reducing active replication 48

2.6 Flow of communications for active standby 48

2.7 Flow of communications for amnesia approach 51

2.8 Flow of communications for a checkpointing DSPS 51

4.1 This figure shows components deployed on two distinct machines. The dashed lines denote *control traffic*; here, we see this as bidirectional communications between the HeartBeat components, from the hosted computations to the `ResourceMonitors`, and from the `ResourceMonitors` to the `CoordinatorNode`. The solid lines denote *data traffic*; We show one hosted computation both receiving data from an external sensor, and then passing results out to an actuator. Our framework puts no limits on either the source or sync of incoming data. We also show a hosted computation sharing state information with a replica. 72

6.1 Density functions of passing response times in milliseconds for 35 and 40 users on a single node (ms) 90

6.2 Probability density functions of passing response times in milliseconds for 1000 and 1400 users on a cluster of 40 nodes (ms) 96

7.1 Small-scale clustering overheads across 2 machines with increasing computation loads (ms). 109

7.2 Small-scale clustering overheads across 2 machines with increasing computation loads (ms). 110

CHAPTER 1

Introduction

The idea of remote health monitoring has been gaining traction over the last few years [1–8]. Health sensors are becoming cheaper and lighter, with greater connectivity and longer battery life. As these devices become more readily available to the average consumer, the possibility of remote health monitoring grows.

Remote health monitoring offers a variety of advantages over traditional health care. Primarily, it allows patients to spend less time in a hospital setting, as well as reducing the amount of man-hours needed to provide a safe environment. This reduces costs to both the patients and care facilities, as well as reducing the stress of extended hospital stays, and also the chance of contracting new diseases in the hospital itself.

Remote health monitoring also has the potential to provide more independence to those who would otherwise need full-time care. Instead of living in a full-time care facility, if their health status could be reliably processed, it would be possible to continue to live in their own homes. This again can result in substantial savings in medical expenditures by both the healthcare system and its patients.

1.1. Monitoring Health Stream Data

Several studies so far have explored the possibility of monitoring health and environment sensors to assist in patient monitoring. In [8], several experiments were conducted where patients were monitored in their own condos, while still meeting regularly with their doctors. One finding was that computer models could often detect small changes in patient behavior that indicated the onset of new symptoms often before the patients themselves felt that

something was awry. Their main focus was developing a system which could aid doctors in the diagnosis and treatment of their patients.

In [3], programmers worked with doctors to produce a framework for computations that more actively monitors patient status. This framework is designed to allow doctors to use already existent monitoring code to actively monitor streaming data from sensors and raise alerts once patient status has breached predefined thresholds.

These previous efforts have shown us several things. First, this is a field that is rapidly growing, and one which will become increasingly necessary as the health care industry becomes more and more stressed. Second, such approaches are actually useful in the diagnosing and treatment of patients; and from [7] we know that customization is important, as is the ability to process health sensor data in real time, without waiting to run batch processes on previously collected data. For many illnesses, early detection and treatment can greatly impact quality of life. Processing data in real time to raise emergency alerts is vital in the timely detection of illnesses.

The goal of this work is to provide a framework to perform custom computations on health stream data in real time. This framework needs to be able to process incoming data at least as quickly as it is generated, and needs to be safe and reliable for patients to use. In order to provide any guarantee of safety for patients, it is important to implement robust behavior through fault-tolerance.

1.2. Challenges in Robust Processing of Health Streams

Health stream data can be difficult to handle for several reasons. First and foremost, patients are relying on their data being processed correctly and in a timely manner. Loss of data or delays in processing could lead to user injury. A framework which supports health

stream computations needs to be robust to both failures and interference—it needs to be able to adapt to changes in the state of the resource pool as well as to changes in data arrival patterns on each and every stream that it is responsible for.

The data itself is the second major challenge. Health sensors typically generate data at sub-second rates. In order to prevent a backlog of data, this data needs to be processed at least as quickly as it is generated. This means we need to be able to process data despite any fluctuations in data volume and arrival rates, failures of either whole machines or communications between machines, and any interference between collocated computations.

1.2.1. Stream Fluctuations. We must expect health streams to change over time. While this may occur slowly, over a period of weeks or months, for example in response to changes in when the patient is most active as a response to changes in daylight hours, it can also occur quite rapidly. For example, we can expect to see bursts during periods of interest, sometimes unexpectedly triggered by the data itself. For example, sampling rates may be increased for a cardiac patient if there is a chance that a cardiac event is occurring. This holds true for other, non-health, streaming datasets as well. For example, the sampling rates of sensors designed to track tectonic activity should be able to increase their sampling rate if seismic activity has been detected.

1.2.2. Machine Failures. In distributed systems, as the number of machines comprising the system increases, the likelihood of one of these machines failing increases significantly. Since data is streaming in real time, we cannot simply wait for a machine to come back up after failure. First, failures may not be transient, i.e., it is not guaranteed that that the machine will come back up. Second, any data destined for a failed computation will be lost. Since this is streaming data, and voluminous in nature, it is not amenable to buffering, so

data cannot simply be 'replayed' in the event of a failure. This also means that any computations which rely on state will have a similar problem – once that computation fails, its state is lost with it. Even if data has been stored to help 'replay' inputs in order to rebuild state after a failure, such an approach is still infeasible, as the computation would need to stay on top of incoming data, as well a processing previous data in order to rebuild state. In such a situation, it is unlikely that the computation would ever manage to catch up on its processing.

1.2.3. Interference between Computations. Computations would suffer no adverse affects from interference if each machine hosted only a single computation. This would also, however, be an incredible waste of resources as well as pose serious scaling problems. In order to make more efficient use of our resource pool, multiple computations – sometimes hundreds – are placed on the same machine. These computations need to share the same limited set of resources, and any one computation which is using more resources than it was originally allocated could cause delays in the processing of other computations. In a health stream processing environment, such delays could lead to dangerous situations for patients, and must be avoided.

1.3. Research Contributions

This dissertation contributes to both the field of distributed stream processing systems, as well as the field of health stream processing. With respect to distributed stream processing systems, we have incorporated support for failure-resilient real-time stream processing. This includes a novel failure-detection scheme that is scalable while amortizing the control-traffic costs associated with failure detection. Our failure resiliency relies on using replication for redundancy. The replication levels are configurable on a per-computation basis, as are the

replication schemes. We include support for both passive and active schemes, alongside a hybrid approach that strikes a balance in achieving fast failovers while minimizing duplicate processing.

We have also designed a novel approach to interference detection and avoidance, leveraging machine learning techniques to not only infer computation placements with minimal interference, but to try to predict future interference and take measures to avoid it. Current approaches [2–8] have focused on much smaller scales, typically with more lax processing constraints as well; where processing needs to be done on the order of seconds, not subseconds.

The field of health stream processing has seen a boom in recent years, mostly focusing on case studies proving that remote monitoring can be beneficial for patients [8, 1]. These studies happen at extremely small scales, usually evaluating the performance of a single computation for a single user, nowhere near the scale that we are working to support. Such studies have usually been run from a health care background, meaning better knowledge of what needs to be supported, as well as access to applications which are currently being used to monitor health data, and have been vetted by professionals. We expand on previous work by exploring the problem from a computing background – how can we leverage what we have learned so far to support larger numbers of patients? How can we ensure that this is a safe transition for patients? How much of this process can be automated?

1.3.1. Contributions to Computer Science. This work extends the field of computer science in several ways. First, it adds fault-tolerance capabilities to Granules [9, 10] stream processing system. This is the first fault-tolerant system which supports arbitrary computations.

Aside from fault-tolerance for arbitrary computations, we have also developed a system which detects interference between collocated computations. Our system takes into account data arrival rates at sub-second intervals and uses this information to not only help detect current interference, but also predict future interference.

We use a novel approach which blends both machine learning techniques as well as traditional distributed systems responses to fault-tolerance. This work not only expands the field of health stream processing, it also represents an important step forwards for generic stream processing as well. Our framework is suited to any stream processing paradigm, and can provide support for any system that needs to process data in real time.

1.4. Dissertation Organization

The rest of this dissertation is organized as follows. In Chapter 2, we survey the use of replication across various distributed systems. Chapter 3 outlines the requirements of a system that needs to support real time health stream processing, the goal we are trying to accomplish with this work, while Chapter 4 introduces the components we have developed to meet these requirements. In Chapter 5 we describe our failure detection and recovery system. Chapter 6 describes our replication system and how it works with the failure detection system to prevent data loss during failures. Chapter 7 introduces our interference detection system, while Chapter 8 focuses on how we mitigate detected interference. In Chapter 9 we conclude, discussing the contributions of this work and its place in the current literature, as well as our plans for future work.

CHAPTER 2

Background

Many distributed systems solve problems with fault-tolerance and load balancing through the use of replicas. We also use replication as the basis of our robust framework. This chapter is devoted to an exploration of replication schemes across distributed systems.

2.1. Evaluating Replication Schemes Across Distributed Systems

Distributed systems harness many machines together to perform a task which would be either impossible or impractical on a single machine. In such a system, failure is expected, so steps must be taken to ensure failures are not catastrophic and do not violate service level agreements. The predominant approach to solving this problem of fault tolerance is to create copies of processes and data across multiple machines, a technique called *replication*. With this approach, if the machine which is running the initial or primary version fails, one of the copies, or *replicas*, can take over and either restart or finish the task. This approach is utilized across many types of distributed systems. In this paper we take a close look at various replication schemes across traditional distributed systems, such as distributed file systems (DFSs), cloud runtimes such as Hadoop [11] and Dryad [12], as well as explore several approaches found in distributed stream processing systems (DSPSs).

DFSs, cloud runtimes and DSPSs have different baseline requirements and service needs, and tend to leverage their replicas in slightly different ways. For example, a DFS would need to handle multiple accesses to a single file, but can generally handle short periods of interruption. A DSPS, on the other hand, needs to deal with continuously streaming data, and can suffer drastically from even a short failure. Systems such as Hadoop and Dryad share some similarities with both of these approaches.

Replication approaches to fault tolerance have been developed independently in these systems, with little work studying their crossover [13] applications. Here we perform a deeper study focused on specific implementations of replication-based fault tolerance.

Eric Brewer's CAP theorem [14] is an important factor to consider when comparing distributed systems and their tradeoffs. In short, this theorem states that large-scale distributed systems need to sacrifice in either Consistency or Availability when there is a network partition. Databases choose consistency and make the entire system unavailable during a network partition. Cloud scale systems choose availability and compromise on consistency by relying on what is referred to as the *eventual consistency* model. This is a very important concept when analyzing the effectiveness of various replication approaches.

Previously, Wiesmann, et al. ,[13] performed a brief study comparing basic replication schemes from both replicated databases and distributed systems. In this paper, the authors developed a *neutral model* to help explain replication schemes at a high level in order to classify replication schemes from different fields. Their work assumes a traditional replicated database, and a distributed system such as a DFS.

In a replicated environment, there is a *primary instance* which is responsible for all transactions. The primary instance then has a number of replicas, placed on different machines so they can continue running even if the primary dies. In *active* replication, the primary as well as all backup nodes are constantly active. Alternatively, in *passive* replication only the primary is active at a given time.

2.1.1. Active Replication. Within the broad definition of active replication, several variations can be found. For example, while every replica generates results, some implementations may only send results from the primary replica. In other situations, generated results may be compared before achieving a consensus and deciding on a result. Active replication

allows state to be built easily among all replicas as inputs arrive, which means it is a good choice for stateful computations where the increased resource footprint balances out the cost of losing state.

In general, active replication will always have a larger resource footprint than passive approaches. At the minimum, an active replication scheme requires the memory, CPU, and inbound networking bandwidth to be scaled up with the replication level. For a replication level of three, essentially three times the resources are required. In the cases where not all replicas forward results to the producer, the outbound networking bandwidth is reduced, but this may be relatively small gain. Active replication tends to have much quicker *fail-overs* – the time until the system can detect and recover from a failure – than passive approaches; where passive approaches need to start up a replica and then redirect input and outputs, the active approach generally takes no longer to fail-over than the initial failure detection.

Active replication is a good choice for computations where state needs to be built up over time. This is a trait which is most often found in distributed stream processing systems, where data is constantly streaming into the network. By letting all replicas see all inputs, we should not have any consistency problems after a failure. The one caveat to this is the case where processing involves a non-deterministic or stochastic operation at each replica. In this case a passive replication scheme which can pass state information to replicas may be the best way to achieve near-consistent results.

2.1.2. Passive Replication. Passive replication has a much smaller resource footprint than active replication schemes. In these approaches, only one replica is active at a time: the primary. There are two main ways in which passive replication is implemented. In the first, all replicas are instantiated, yet remain dormant until needed. Once the failure has been detected, inputs and outputs are redirected to point to and from the replica which takes the

primary's place. In the second case, the replicas may not even be instantiated until needed. This approach has the lowest overheads, but the highest cost with respect to failover time.

To help reduce failover time, passive replication schemes can transfer state from the primary to the replicas, in a process commonly called *distributed synchronization*. It is important to realize that by performing this synchronization operation we are bringing the replicas up to a full processing footprint for a portion of time. In this case we are again creating a bigger memory, CPU, and network I/O footprint, losing some of the benefits of using passive replication.

Passive replication is well-suited for stateless operations in environments where resources are at a premium and some delay in processing is acceptable. Delays occur during synchronization (if implemented), as well as when recovering from failure. Unlike an active approach, where the replicas are all fully active and have been receiving inputs and generating outputs all along, passive replicas need to spend some time recovering functionality – as they are usually in a dormant state – and all inputs and outputs need to be redirected to point to the new primary. Passive replication schemes can support stateful operations if checkpointing is implemented, but this comes at the cost of an increased resource footprint.

2.1.3. A Framework to Compare Replication Schemes. [13] discuss a method for analyzing distributed file systems (DFSs) and distributed databases. By abstracting the idea of replication into a neutral model, we can see a direct and fair comparison of these two types of systems. Here, we expand on the work first discussed by [13] to look at specific, current DFSs, the cloud computing frameworks which run on these DFSs, as well as several replication schemes implemented in DSPSs.

In the original work, operations were broken down into a five-phase process:

: Request – the client submits an operation to the replicas.

: **Server Coordination** – the replicas coordinate to synchronize execution (to preserve order of concurrent operations).

: **Execution** – the operation is executed.

: **Agreement Coordination** – operation outcome is agreed upon (to preserve atomicity).

: **Response** – the outcome is transmitted back to the client.

The authors claim that the order in which these processes execute helps to define the replication approach of the distributed file system or distributed database operation. With several approaches, some of these phases are iterated over, while other phases may be skipped entirely. For example, in a purely passive replication scheme where only one replica is active at a time, there is no need for the Server Coordination or Agreement Coordination phases.

While we can use this notion of an abstract execution of phases which makes up replication, [13] were interested in comparing distributed file systems and distributed databases. We are focusing on distributed file systems, cloud computing frameworks, and distributed stream processing systems. While these phases do not directly translate into the distributed systems we discuss, we have used this idea to develop a new set of phases to help describe the communication patterns in these systems. The biggest differences arise because the original work assumes a much stronger consistency model than we see in the distributed systems discussed here (hence the multiple, distinct coordination phases), as well as assumed client interaction. The client interaction phases are most confusing in the context of DSPSs where data is constantly streamed. To remove some of this ambiguity we have developed a slightly different set of stages:

: **Data Entry** – Data enters the replicated process. This data may be supplied by a user, an orchestration process, an upstream replicated unit, or some other producer of data.

: **Coordination** – Replicas may contact each other and share information. In some implementations, this stage may not appear at all, while in others it may appear multiple times.

: **Execution** – Data is processed. With a cloud runtime or distributed stream processing system, this means performing the specified operation on the data. With the distributed file systems, this means writing/reading the data.

: **Results/Response Generated** – In the case of a write in a DFS, a confirmation is sent signifying the write proceeded correctly. In the case of a read, the data is returned to the user. With a cloud runtime or DSPS, this generally means that the results are then passed on to the next stage of the computation or some other consumer of the results.

In the rest of this chapter, we use this framework to help analyze and classify replication schemes across distributed systems. For each replication scheme we discuss, we will be investigating the following specific points:

: **Replication Goal** – What is the ultimate goal of replication in this system? Replication may be used to provide increased availability, fault-tolerance, correctness, or to increase processing speed. An important aspect of each replication approach is understanding the primary purpose of the replication scheme.

: **Replica Placement** – How does this approach place replicas? Replicas should not all reside on the same machine, as this would negate any fault-tolerance gained by

instantiating replicas. Additionally, the system load needs to remain balanced. If a single machine is overloaded, a weak point has just been introduced into the system.

: Replication Level – How many copies are made of the process/data? While a larger number of copies means a smaller chance of complete, unrecoverable failure, maintaining excessive copies means that less data/computations overall can be stored in the system. Finding a balance of resilience while avoiding redundancy is a difficult problem.

: Synchronization – When stateful information needs to be passed between replicas, the synchronization policy becomes vital. Replication is not effective if a backup does not produce correct results. Synchronization schemes help to ensure that failures do not affect correctness. This is directly related to the coordination phase described above.

For each distributed system analyzed in this paper, we explore how these four factors interact. By understanding how these factors work together, we reveal the various tradeoffs inherent to each approach. The goal of this work is to provide a comparison across a broad selection of distributed systems, providing a unique insight into the strengths and weaknesses of these systems. As we analyze each approach to replication we will also look at the cross-cutting tradeoffs of:

- Memory Utilization
- CPU Utilization
- Network I/O
- Consistency and Availability in the presence of failures

2.2. Replication in Distributed File Systems

Distributed file systems have several unique characteristics. They manage files on disk, but also need to handle incoming requests from multiple users in real time. Each system may have very different ideas about what user interaction will look like, such as long, sequential reads and writes vs. short, random reads and writes. Each system is then designed around these assumptions. They also may need to handle situations where there are partial network disconnects – can a user continue modifying a file while they are disconnected from the network? An example where this behavior is desirable is when working with a file on a mobile device. Even if a mobile device holding the primary copy of a file is disconnected from the network, users connected to the network should be able to view even modify the file. In this section, we will be focusing on writes, the more challenging process in DFSs. Reads are usually a much simpler operation, and often involve only communicating with a single node, and does not generally create conflicts.

In [15], Gray, et. al. explore the challenges of replication in a distributed file system. To understand the tradeoff space of a distributed file system, they work with a fictitious file system with naively implemented replication schemes. Most importantly, the focus was on replication schemes where all replicas had very strong consistency needs. This means that all replicas need to reply that the changes have been successfully performed before any modifications can be finalized. The authors are assuming that these operations have strong ACID (atomicity, consistency, isolation, durability) guarantees, which are normally relaxed in a distributed environment due to the difficulty of maintaining consistency [16].

While there is the obvious problem of making headway while users are concurrently connected and attempting to update the same pool of files, the slightly subtler problem occurs with an explosion of networking overhead seen in this type of environment. If the

primary and all replicas need to pass messages before and after any change is made to a file, each transaction cost is multiplied by the number of replicas. If transactions are expected to come at a steady rate, the number of deadlocks arising from attempts at concurrent writes will grow at the same speed. From [15]: "A ten-fold increase in nodes and traffic gives a thousand fold increase in deadlocks or reconciliations." The bottom line is that fully active replication in a DFS is impractical and doesn't scale.

2.2.1. Google File System. The Google File System (GFS) [17] has been designed to handle long, sequential reads and writes. This means the system may have reduced performance when trying to perform many short, random reads and writes. GFS was also designed under the assumption that the primary users would be applications attempting to read and write files, instead of humans. The advantage of this approach is that there is some built-in leniency for consistency. Where a file which needs to be read by humans can have no extra or *garbage* bits, this is an acceptable approach when applications are expected to be the primary users of a file system. It is simply then a requirement that any application using the file system is capable of recognizing and skipping over any garbage bits which are left as the result of inconsistencies when writing data.

GFS is able to store extremely large files by breaking these files into pieces, or *chunks*. This means that even an extraordinarily large (as in multiple petabyte) file can be easily stored in GFS. These chunks are also more easily loaded in memory, meaning that heavily accessed files can remain in memory, reducing the amount of time needed to obtain a file from GFS by cutting out the bottleneck of reading from disk. While replication levels are set at the folder level, each chunk of every file is independently replicated across multiple machines. This dispersion allows for concurrent parsing of files using MapReduce [18].

Replication Goals. The primary goals of replication in GFS are fault-tolerance and availability for concurrent processing with MapReduce. In the event that a machine fails or a replica becomes corrupted, there is another copy somewhere else in the system. When one replica is lost, the remaining copies are responsible for making sure another replica is introduced to the system to return to the appropriate replication level.

Replication is prioritized based on whether the chunk belonged to a live file that is being actively read/written to and how far the replication level for the chunk is from its desired replication level.

While it is not the main goal, replicas in GFS can also help provide decreased latency. Replicas are spread between geographically disparate data centers, so the primary can be chosen as the replica closest to the client. This is possible due to Google's internal IP assignment scheme, which allows the system to infer location from IP address.

Replica Placement. Google assumes that their datacenters are safe and controlled environments. Part of this means that they can trust the IP addresses of the servers to accurately reflect server placement. Google makes use of rack awareness to ensure that all replicas are not lost with a single failure. While the exact behavior of this mechanism is not detailed by [17], it is clear that: a) replicas are never located on the same machine, and b) some effort is put into making sure that replicas are located in different failure zones – meaning they may be placed in different data centers entirely. This approach ensures that no data is lost for small, localized failures – such as loss of power to an entire rack; as well as large-scale failures – such as the loss of an entire data center.

Replication Level. By default, GFS has a replication level of three for every file. This replication level is maintained across all chunks which comprise this file. GFS also allows users to set a default replication level on a per-directory basis. This allows users to designate

a `/tmp` directory which is not replicated. This helps to avoid unnecessary replication of files for which replication guarantees are not needed.

Synchronization. When a user is writing to GFS, a single replica is chosen as the primary. This one node is then responsible for synchronizing file modification across the group of replicas. GFS streamlines communications by having all interactions between replicas occur in a chain which leverages internal network topology information. The client only sends data to the closest replica, which then forwards any data to the other replicas. This streamlined approach allows large amounts of data to be written quickly, while having a single replica take the lead for synchronizing writes simplifies the task of concurrent writes.

GFS also provides availability by reducing the constraints on consistency. GFS will not block writes, so there is the chance that concurrent write requests may append data in an inconsistent manner. Because of this, replicas in GFS are never guaranteed to be byte-wise identical.

Supporting Concurrent Writes. At first glance, GFS appears to be an active replication scheme – all replicas are constantly active, and a user may communicate directly with any of the replicas. This view is a bit simplistic, however. In GFS, a client only ever communicates with one replica at a time, the primary. This primary is responsible for ordering concurrent operations as well as ensuring that writes are successful across all replicas. This supports the model of a passive replication scheme, where the primary performs operations, and pushes state information to the replicas as necessary.

Using the stages developed by [13], we see the outline of communications shown in Figure 2.1. As we can see, the client first pushes data to one of the replicas, in the Data Entry phase. After that, we see the first Coordination phase where the data is pushed to all other replicas. Once this has happened, the client can tell the primary to write. In the second coordination

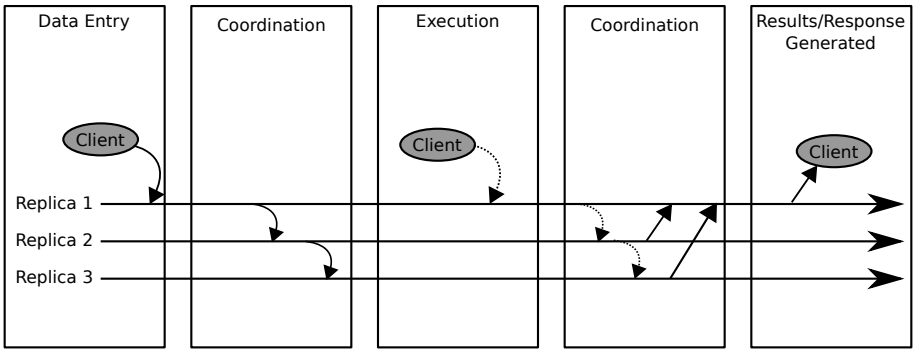

FIGURE 2.1. Flow of communications for GFS

phase, the primary passes the write command along to the other replicas. The replicas then respond to the primary once all data has been written, and the primary informs the client of the successful write. These two stages of communication make GFS stand out in this study – all other DFSs discussed here only contain a single stage of coordination among replicas. While this does mean that GFS incurs a higher communications overhead, this two-phased coordination step allows GFS to handle concurrent writes gracefully and actually helps to increase the availability of the system.

2.2.2. HADOOP DISTRIBUTED FILE SYSTEM. The Hadoop Distributed File System (HDFS) [19] is an open source implementation of GFS. While based on GFS, the fact that HDFS is expected to run in a variety of environments (instead of solely a safe, known environment), leads to several distinct differences between the two file systems. HDFS also has stricter consistency guarantees, as its clients are not expected to be able to handle garbage bits, or segments of failed concurrent writes.

Replication Goal. As with GFS, the primary goal of replication in HDFS is fault tolerance. HDFS is capable of making sure that replicas of each data chunk are kept on different physical machines to ensure that a single machine failure is not catastrophic. Given that HDFS needs to run in an unknown environment, this is a more difficult task in HDFS than in GFS.

Replica Placement. Like GFS, HDFS can make sure that replicas are not on the same machine, and it tries to make sure replicas do not all exist on the same rack. Unlike GFS, however, HDFS cannot determine machine location from IP address – since HDFS can be run by anyone anywhere, it is impossible for the system to automatically detect machine location. From [19], the general approach to replica placement (when there are 3 replicas) is to place the second replica on the same rack, but a different node than the primary. This can help to cut down on increased inter-rack communication, which may be costly. The third replica is then placed on a different rack entirely. HDFS is able to ensure this only when users supply rack information, otherwise the system has no way of knowing which machines are on which rack.

Another interesting problem HDFS faces with replica placement arises from the opportunity to run it on resources such as Amazon's Elastic Compute Cloud (EC2) resources as explored by [20]. This is particularly troublesome if Amazon's *small* resource is used [21]. In an elastic environment, where more machines are added as needed, it is difficult to ensure replicas are not stored in virtual machines (VMs) hosted by the same physical machine.

Replication Level. Like GFS, HDFS has a default replication level of 3, which may be reconfigured when starting up the cluster. HDFS allows a replication level to be specified as files are produced or even modified after the file has been created. While this seems to allow more options than discussed in GFS [17], it is not clear whether or not similar behavior is supported in GFS.

Synchronization. HDFS assumes long, sequential, write-once behavior. It also assumes that there will only ever be a single writer. This immediately reduces the number of consistency problems which will arise, as failures occurring during concurrent writes is the primary cause of consistency problems. While this strengthens consistency guarantees, it also reduces

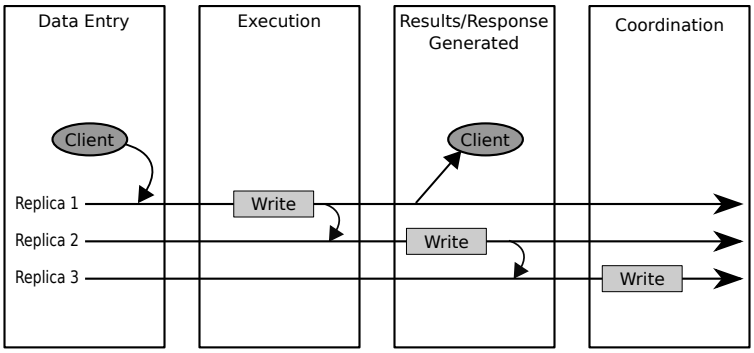

FIGURE 2.2. Flow of communications for HDFS

the availability of the cluster. If another client already has a write-lock on a file, it is simply unavailable for another user to write to the file.

HDFS also employs a pipelined approach to writing data, which is outlined in Figure 2.2: The coordination phase can be seen as occurring throughout the results/response generated phase as well as the coordination phase since the primary will report success to the client as it sends on data to the replicas. This approach appears to leave a loophole for potential errors if only the primary performs a successful write.

Sacrificing Availability for Concurrency. HDFS is more clearly an example of passive replication than GFS. Where in GFS the client may initially push data to any of the replicas, the client in HDFS only ever communicates with the primary. This again arises from HDFSs need to function in an unknown environment. Where GFS can speed up writes by allowing the client to start pushing data to the replica which is physically closest, in HDFS this isn't known so all communications must go through the primary. As clients only communicate with the primary, this is a passive replication scheme with quick state updates – data is passed on as soon as it has been processed.

As discussed above, we also see some interesting problems arising from the ability to run HDFS on almost any platform. From EC2 instances where what appear to be different machines may be the same physical device, to a hole in the write procedure – the primary

will report success back to the client before all writes have completed successfully. The approach of only writing through the primary does let HDFS provide a stronger consistency guarantee, but at the cost of availability since users are blocked from performing concurrent writes.

2.2.3. Azure. Azure [22, 23] is Microsoft's cloud computing framework. It has a tiered architecture, where the storage solution runs on top of a *fabric layer*. This underlying fabric layer is then responsible for monitoring individual system health and status. Through this monitoring system, Azure is able to recognize machine failures and work to re-replicate data as soon as failures are detected. Azure is not very well defined, so there are many open-ended questions raised in this section.

Replication Goal. In the Azure storage system, the goal of replication is purely for fault tolerance. As far as a client is aware, there is only ever one copy of the data – clients are not aware of replication on any level. Azure guarantees read-what-you-write consistency, so it seems likely that all read and write operations pass through a single, primary node.

Replica Placement. Replicas are spread across *fault domains* to ensure no data is lost entirely should replication fail. A fault domain is defined as a group of machines which will fail together should a single piece of hardware fail. From the whitepapers available [22, 23], it appears that there are multiple fault domains in a single data center, and one fabric controller per data center. Since the fabric controller is responsible for detecting failed machines and replicating processes, it appears that Azure does not support inter-datacenter replicas, meaning data may be lost if an entire data center goes offline.

Replication Levels. From [22, 23], Azure has three replicas for all data, so it has a replication level of three. This does not seem to be a modifiable value, though it is difficult to be certain given the available documentation.

Synchronization. From the read-what-you-write guarantees, it seems likely that Azure employs a pipelined write approach like in HDFS – once data has been successfully written to the primary, it is forwarded and written to all replicas. It is unclear whether or not Azure may suffer from the same loophole we see in HDFS where a write may not be able to achieve full replication.

Azure Replication. It is difficult to classify Azure with confidence given the dearth of information published so far. As discussed in the previous section, Azure's read-what-you-write guarantees seem to imply that passive replication is implemented here. The user is expected to interact with only a single node at a time, and this node is responsible for passing along information to the replicas.

Without specific details, it is difficult to fully understand the strengths and weaknesses of Azure. One key point which was hinted at in [22, 23] is that replicas may not be able to exist across different data centers. This means the fault-tolerance guarantees of Azure are limited, possibly weaker than both GFS and HDFS. Additionally, if we assume that the write strategy is closer to HDFS than GFS (limiting concurrent writes), Azure also suffers from reduced availability in favor of stronger concurrency guarantees.

2.2.4. Distributed File Systems Trending Toward Passive Replication. The distributed file systems discussed in this section are a cross-cutting example of modern DFSs. These systems have seen extended usage and have withstood production workloads. In general, we see that DFSs tend to fall into the category of passive replication solely for the purpose of fault-tolerance. While all are looking to solve essentially the same problem, we see that consistency and availability guarantees vary across approaches.

One trait which these DFSs all share is the requirement for a single node to be in charge of returning replica location to clients and choosing primaries. GFS has a single master

node, HDFS the namenode, and Azure has the fabric controller. These all represent single points of failure, and can cause delays to the system when one of these controller nodes fail. While a single point of failure can limit system availability, it also makes the system simpler to manage, and helps the systems remain partition-tolerant.

2.3. Replication in Cloud Runtimes

Cloud runtimes are generally designed to take advantage of a distributed file system, allowing computations to be run where the data is, reducing communications overhead. We see this pattern with the Google File System (GFS) and MapReduce, Hadoop Distributed file system (HDFS) and Hadoop, and Azure and Dryad.

Cloud runtimes need to handle the same situations as a DFS where machines may unexpectedly fail, as they are often running on top of DFS instances. Cloud runtimes often rely on an underlying DFS, so we see a different approach to fault-tolerance than what we have seen in distributed file systems so far. In this section we look at four different cloud runtimes: MapReduce, Hadoop, Dryad, and BOINC volunteer computing.

2.3.1. MapReduce.

MapReduce was developed by Google to handle the processing of voluminous data. While the code to process the data may be simple, such as a `grep` operation, the code to properly distribute the process, manage failures and recombine results makes otherwise simple code unwieldy and difficult to understand or produce. MapReduce builds off of primitives found in functional languages, such as Lisp, which allow the user to split input data and run in separate processes, then recombine the results – a map, then a reduce. Google's MapReduce, as presented by [18], sets the basis for the MapReduce paradigm.

When the user starts a MapReduce job, several processes are spawned. One of these processes becomes the master, and is then responsible for coordinating the MapReduce operation. The rest of the processes are then workers, which are assigned the map and reduce tasks which make up the process. The master is responsible for detecting and handling worker failure. Failure of this master means unrecoverable failure of the entire job.

Replication Goal. The primary goal of replication in MapReduce is to provide fault tolerance. Once the master decides that a worker node has failed, it will shift processing to accommodate failure. Any map task which has been completed, yet not consumed by a reduce task will be rescheduled to run, as will any task which was in progress when the failure occurred. MapReduce stores intermediate results locally, so any completed yet unconsumed tasks need to be rerun. Should a reduce node fail, only in progress tasks need to be restarted – any completed tasks are stored in GFS, so are not lost when a single machine fails. MapReduce can also make use of replicas to speed up processing as discussed below in Replication Levels.

Replica Placement. MapReduce is expected to run on top of GFS, so the master node will attempt to push computations to machines holding a copy of the input data. This is applicable to map functions, but not to reduce functions. When a map task cannot be placed on a machine which contains a copy of the data it needs to process, it will put the map task as close to the data as physically possible (a possibility since Google can infer rack location from IP address). When multiple replicas are live, the master will make sure that there are never two identical tasks running on the same physical machine.

Replication Levels. MapReduce assumes GFS is the underlying file system, so it never needs to work with a case where inputs are lost – there will always be a copy somewhere in the cloud. To speed up intermediate writes, MapReduce writes these to local disk which is

relatively fast, but will not survive node failure. This means that MapReduce can successfully run with a replication level of 1, simply restarting any failed process from the beginning.

By default, MapReduce has *backup tasks* [18] enabled, since this ability has shown that it can provide a 44% increase in completion rates. Backup tasks allows processes which are running slower on average to be scheduled across multiple workers. In this way, a faster machine may be able to pick up the slack from a machine running more slowly – possibly an effect of failing hardware, or overscheduling. This results in a replication level somewhere between 1 and `N`, where `N` is the number of backup tasks launched. This reveals a side goal of replication for MapReduce, as processing speed may be increased by introducing additional replicas to the system. This approach helps to reduce the impact of processing bottlenecks, but cannot help in an ideal environment.

Synchronization. MapReduce has no concept of synchronization. In a best-case scenario, only one replica of any process is ever active at a time. When a backup task is started, these replicas have no communication with each other and run in parallel until the task has been completed.

Active Replication and a Single Point of Failure. MapReduce is an active replication scheme – all replicas receive inputs and generate outputs. In a best-case scenario there is only ever one replica active: the primary. While this also matches a passive scheme, where only the primary receives inputs and generates outputs, when backup tasks are taken into account MapReduce is clearly an example of active replication. An example of the flow of communications in a MapReduce job can be seen Figure 2.3. The job is pushed to a single replica, which performs the requested operation then passes on results to the next stage of the computation. Should a backup task be started, the job will be pushed to a second replica for processing, with no communications taking place between the replicas.

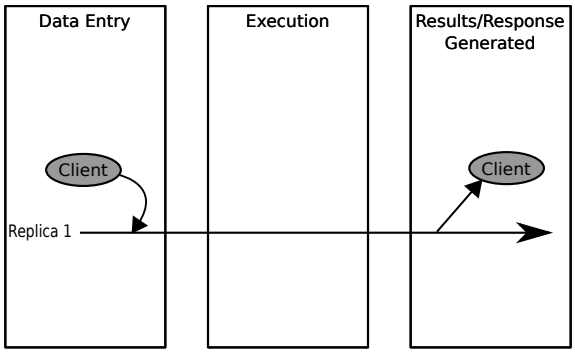

FIGURE 2.3. Flow of communications for MapReduce, Hadoop, and Dryad

One weakness with this approach is the existence of a single master node. As discussed previously, this master node is responsible for monitoring workers and orchestrating the computation. If the master fails, however, there is no other node which can take over the orchestration process – the computation simply fails. This single point of failure is a major weakness of MapReduce, but was more than likely introduced to keep the approach simple. If multiple masters are introduced to the system, the likelihood of problems due to partitioning rise significantly.

2.3.2. HADOOP. Hadoop [11] is an open-source implementation of the MapReduce paradigm, originally started by Yahoo! but now an official Apache project. While based off of MapReduce as defined by [18], Hadoop is more flexible due to its open-source nature, and may be more advanced today than MapReduce.

Replication Goal. Much like MapReduce, the primary goal of replication in Hadoop is fault tolerance. When speculative or backup tasks are enabled, Hadoop also uses replicas to speed up processing times by running replicas in parallel in a winner-takes-all approach. Results are taken from the task which finishes first, and passed on to the next stage of execution.

Replica Placement. Like MapReduce, Hadoop attempts to place replicas on a machine holding a copy of the input data. If a task fails in Hadoop's map reduce framework, a new task is simply relaunched on a different machine. While some effort is taken to find a node which is not 'overloaded', this is not a well-defined term. It is important to stress again that by default Hadoop does not launch additional replicas unless the master has determined that a machine has failed, or backup tasks are enabled and a process is running slowly. While Hadoop has a built-in speculative task launcher, there has been some research into developing a more robust solution which performs more reliably in a heterogeneous environment [24].

This problem with heterogeneity becomes apparent when trying to run Hadoop on an EC2 cluster. When using small instances, where multiple VMs may be sharing the same physical machine, the processing power and resources available may differ drastically as the VMs compete for resources. In such an environment, it is even possible that a job and its backup task may end up sharing the same physical machine, causing the addition of more backup tasks to create an even greater bottleneck.

Replication Level. Technically, Hadoop has a default replication level of 1, and a replication level somewhere between one and `N` (where `N` is the number of backup tasks launched) if a speculative task launcher is used. The default behavior is to launch a new copy of a task only after the original has failed, thus the default replication level of 1. When tasks are speculatively launched (either with the built in scheduler, or with an external scheduler such as LATE [24]), new tasks are speculatively launched once it is deemed that a previously started task is running too slowly.

Synchronization. Like MapReduce, there is no synchronization between replicas. When there is only one replica, there is nothing to synchronize with. When another replica is launched (in the case of a failed/slow task), the task is restarted from the beginning with

the original inputs. In this sense, synchronization is handled by the underlying file system, which is responsible for storing all inputs and providing them as needed.

Active Replication and Transient Failure. Like MapReduce, Hadoop is technically an active replication scheme. In Hadoop, speculative tasks are disabled by default, so the situations where speculative tasks are assigned is assumed to be in a minority. Communications in Hadoop follow the same pattern as we found in GFS, which can be seen in Figure 2.3.

White [11] states that the timeout for heartbeats (how long the system takes to detect failures) is 10 minutes. This can reflect several different design decisions: 1) Hadoop jobs are expected to be very long running – an average job is expected to take more than 10 minutes; 2) Hadoop expects short, intermittent network failures – even if the master loses contact with a worker for a few minutes, there is still a good chance it is running successfully, so the master should wait and see if it comes back before deciding that the machine has actually failed.

While the LATE scheduler has been designed to work with Hadoop, it would be interesting to see how it compares to Google's MapReduce speculative scheduler. We do not have access to Google's MapReduce code, so we cannot actually perform this comparison, but in theory the schedulers should be applicable across both Hadoop and MapReduce. Given that LATE was designed specifically for a heterogeneous environment, it is likely that it could outperform the MapReduce scheduler should they both be tested in a heterogeneous environment.

2.3.3. Dryad. Unlike MapReduce and Hadoop, Dryad does not follow the MapReduce paradigm. Instead, Drayad programs take the form of a directed acyclic graph (DAG) where the vertices are operations (possibly DAGs themselves), and the edges are the communications bridges between stages of computation. Unlike MapReduce and Hadoop, Dryad does

not require Azure to be running underneath it. Drayad does, however, expect that large inputs are broken up into pieces and spread among the machines in the cluster [12].

Replication Goal. The goal of replication in Dryad is fault tolerance, just as in both Hadoop and MapReduce. In Dryad, it is assumed that every vertex on the execution graph is a deterministic operation, so tasks can simply be restarted elsewhere in the cluster if a machine fails. After a number of attempts to re-run a job, the job scheduler assumes there is some unrecoverable error and cancels the job. From [12], it is not clear whether this limit is configurable on a per-job basis, or something which needs to be set for the cluster on startup.

Replica Placement. Much like MapReduce and Hadoop, Dryad has a single job manager which is responsible for coordinating job execution. Additionally, the Dryad job manager is expected to reside outside the cluster where the work is being performed. As such, if the job manager machine fails the whole job terminates – there is no way to recover. This is essentially the same as what happens in MapReduce and Hadoop if the master node dies.

Every program, or vertex of the job graph can specify a list of preferred machines in the cluster to run on as well as minimum resource requirements. The job manager will then perform greedy scheduling to place jobs – it works under the assumption that no other jobs will be running on the same cluster. This means that a Dryad cluster will be underutilized, and also hints that the average Dryad cluster is smaller than the average MapReduce cluster – thousands of machines would be overkill for a cluster hosting only a single job at a time.

Replication Levels. Dryad has a replication level of one. While Dryad users are capable of specifying where multiple instances of the same process should be run, they are expected to operate on different portions of the input data, and Dryad does not treat them as interchangeable entities. Following this assumption: in an ideal situation, where the programmer

has perfectly designed execution graph and programs have been placed perfectly, there is a replication level of 1. Dryad will start up replica processes in the event of failure, much like MapReduce and Hadoop, but will not perform speculative scheduling.

One unique ability of Dryad is a runtime refinement of the execution graph. These refinements take into account program placement, communication format, and graph layout. For example, Drayad may add replicas for a fan-in operation. This approach can reduce inter-rack communication costs by compressing results. In other instances, Dryad may decide to replicate an existing vertex on the execution graph for the same purpose. While these two cases are close to true replication, they are treated as unique computations by Dryad, meaning a new replica will be instantiated should one of these operations fail.

Synchronization. Dryad has a replication level of 1, so no synchronization is needed. Identical copies of a program, such as the result of a fan-out or -in operation, have no communication and operate on different portions of the input space. This justifies labeling Dryad with a replication level of 1. Again, this replication level means no excess processing in a best-case scenario, but an increased cost of needing to restart from the beginning should a process fail.

Active Replication with Underutilized Resources. Dryad is another example of active replication. While no replicas are started until a failure occurs, all replicas (the primary) receive all inputs and generate output. Due to the lack of speculative scheduling in Dryad, it is more difficult to come to this conclusion than in MapReduce and Hadoop. While the runtime refinements may appear to be replicas, they do not behave like replicas and are treated as individual computations. Dryad shares the same communications footprint as MapReduce and Hadoop, which can be seen in Figure 2.3.

One major inefficiency in Dryad is the assumption that only a single job is being run on a cluster at a time. While this does reduce the amount of information which needs to be shared between the worker nodes and the job manager, it also implies wasted resources. If you are only running one job at a time on a cluster you are wasting processing power – in MapReduce and Hadoop, it is assumed that many jobs may be concurrently running, allowing computations to be interleaved across machines. It also suggests that Dryad clusters are expected to be much smaller than MapReduce or Hadoop clusters (an average MapReduce job may use 2,000+ machines [18]).

Another concern with Dryad's approach is data loss. Dryad does not require Azure to be running underneath it, so data is not necessarily replicated throughout the cluster. While large data inputs are assumed to be partitioned across the machines, it is not clear if there is any replication associated with the underlying DFS.

2.3.4. Berkeley Open Infrastructure for Network Computing. BOINC [25], the Berkeley Open Infrastructure for Network Computing, is unique in this analysis. Unlike the other cloud runtimes discussed here, it does not have an associated file system. It is designed to allow scientists to easily distribute data and computations to a pool of volunteers, hosting projects such as SETI@home, Proteins@Home, and Climateprediction.net.

The open nature of BOINC leads to several distinct design decisions. First of all, BOINC jobs are expected to run in an untrusted environment. This means that unlike Hadoop, MapReduce, and Dryad BOINC operators have no control over the machines in their work pool. Machines operate solely on free CPU time donated by users, so machines may unexpectedly join or leave the pool of available workers. Additionally, BOINC operations may be paused, delayed, or completely halted by volunteers unexpectedly. This notion of pausing a process has no equivalent in the other runtimes we have discussed here.

Aside from a far less stable environment than other cloud runtimes, BOINC has to face the additional problems of erroneous or malicious results. Erroneous results may arise from impending hardware failures, such as a faulty hard drive corrupting data, or even flaws in the hardware. Aside from erroneous results, users may attempt to return malicious responses. Malicious responses may range from bogus results, to bogus processing times – an attempt to claim more CPU cycles were donated than actually used. Because of this, replication in BOINC plays two roles: fault tolerance to handle losses from the worker pool, and verification to handle potential erroneous or malicious results.

Replication Goal. BOINC uses replication not only for fault-tolerance, but to surpass the difficulties arising from operating in an untrusted environments. The approach of using replication to verify results makes this framework stand out. BOINC also uses replication to handle cases where clients are not able to continue processing – which can occur often in a volunteer computing environment where the volunteer computation may be abruptly dropped so that another process can use those cycles.

Replica Placement. Replicas of a specific task are never sent to the same machine, as this would invalidate any verification steps. Computations can further affect replica placement by designating a specific architecture or operating system on which to run replicas. This can help when precision is needed and different hardware may return slightly different results.

Replication Level. In BOINC, two values are specified on a per-project basis: `N` and `M`. For every unit of work which needs to be done, `N` replicas are generated and scheduled for execution. `M` is some value where $M \leq N$. After `M` results are in, the managing program will then compare results in an attempt to determine a consensus. If there is no consensus, or too many of the original `N` computations fail (such that `M` cannot be reached), another `N` set

of replicas will be scheduled for execution. This cycle will continue until either a maximum limit on the number of replicas created is hit, or the scheduling attempt is timed out.

Synchronization. Synchronization in BOINC does not occur between replicas. Replicas are simply created for one-shot computations – all data needed resides at a central server and is sent out with the computation to be run on volunteer machines. Synchronization, such as it is, occurs on the server side. When results are returning in from clients, the *validater* [25] is responsible for verifying results against other replicas, and deciding if more replicas need to be spawned off to determine a definitive result. When a definitive result has been found, the volunteers who contributed to the result are awarded participation incentives, while any still running processes are sent terminate commands.

Active Replication in a Volunteer Environment. Like the other cloud runtimes discussed in this section, BOINC is an example of active replication. All replicas receive all inputs, and are expected to run any processing in parallel. This fits the ultimate goal of BOINC – verification through replication. All replicas receive the same inputs and run the same process, so all results from the replicas should match. While variations between systems may cause small differences in results, they all should fall within a margin of error. By pruning results which fall outside this range, BOINC can identify erroneous or malicious results. The flow of communications in BOINC differs from what we saw in the other cloud runtimes, and can be seen in Figure 2.4.

BOINC is unique in expecting to run in an untrusted environment, meaning extra measures must be taken to ensure that valid results are returned, per the quorum method described above. This approach is, however, wasteful: there will at a minimum be $N - M$ wasted replicas. While setting `N` and `M` to the same value means no work is wasted, it also means that the entire job may be held back if a single replica is running more slowly than

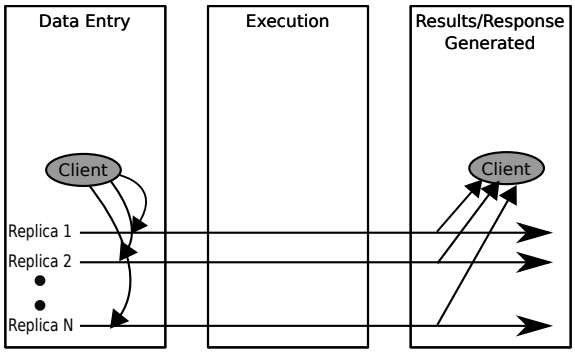

FIGURE 2.4. Flow of communications for BOINC

the others – a likely scenario given that BOINC is designed to run in idle time on personal computers. This difference between `N` and `M` is essentially the leeway that describes the speculative or backup task space we see with MapReduce and Hadoop.

2.3.5. CLOUD RUNTIMES: ACTIVE REPLICATION IN STAGES. Cloud runtimes generally rely on a replicated, external source for fault-tolerance with respect to inputs, and it is assumed that it is cheaper to restart a job than to launch multiple replicas at the start. With the exception of Dryad, all these approaches also have some leeway designed into the system to help speed up processing. In MapReduce, stragglers are common, enabling speculative or backup tasks can lead to a 40% increase in processing speed [17]. Should a similar feature be designed for Dryad it stands to reason that it could lead to an equivalent increase in performance.

An interesting trait revealed in this work is the tendency for cloud runtimes to be examples of active replication schemes, usually with a very low replication level. This is most likely due to the assumption of an underlying replicated system responsible for holding inputs as well as a result of the type of applications expected to run in this environment. In general, cloud runtimes assume that processes are one-shot computations which have a long runtime.

One other trait inherent in all these approaches is that there are assumed stages to computations. These stages do not execute in parallel, but sequentially. In the case of MapReduce and Hadoop, the map phase is run and then the reduce phase is scheduled. In the case of Dryad, the execution graph is broken down into stages which can be performed in parallel – ones which do not rely on outputs of anything currently running to execute correctly. Even in BOINC, there is never a case where programs are sent out which require output from anything else currently running. This means that these approaches do not make a good candidate for streaming data, where inputs are continuous and programs are made up of chains of individual computations which feed into each other.

2.4. Replication in Distributed Hash Tables

Distributed hash tables (DHTs) are designed for persistent storage of data. All DHTs support 2 main types of operations: a `get(key)` and a `put(key, data)`. By introducing a hashing function on all keys, it is possible to easily distribute data evenly among a pool of available servers.

DHTs are designed to allow efficient lookup of data obtained through a consistent hashing scheme. They are often used in place of traditional database schemes which do not scale well in a distributed environment.

DHTs typically rely on an active replication scheme, much like a DFS. Replicas are hosted on several disparate machines, and any can be accessed by a user for speedier lookups (double-check). Peer to Peer (P2P) systems typically take advantage of replicas to reduce the search space for objects stored in their network.

2.4.1. Chord.

Chord [26] was designed as an efficient peer-to-peer lookup scheme. It is a protocol which allows a user to quickly identify a node in a cluster given a key. This

approach is robust enough to handle lookups even when a cluster is in constant flux. By associating data with these keys, Chorrd becomes a scalable storage solution in a distributed environment even when cluster membership changes often.

Chord relies on each node knowing about a subset ($\log N$) of its neighbors that are organized in a hash ring. Any datastore built on the Chord protocol can implement a replication level `r` simply by enforcing copies of data to be stored at any nodes `r` successors. As this follows Chord's default lookup path in the face of failures, this is a natural solution to the problem of fault tolerance.

2.4.2. PNUTS. PNUTS (Platform for Nimble Universal Table Storage) is a storage solution developed by Yahoo! PNUTS allows both a hash storage function as well as a more traditional database storage with a table and row structure.

PNUTS supports replication at a data center level. This means that no single data center will ever contain replicas, replicas only exist in geographically disparate locations. By ensuring replicas are broadly dispersed, no single failure will completely destroy data. Furthermore, PNUTS defines the primary copy as the replica geographically closest to the user – through this method, PNUTS uses replication to cut down on latency. PNUTS is designed to run as a hosted cluster which supports multiple different applications concurrently. Fault-tolerance and consistency guarantees are determined on a per-application basis, so individual data items may have different numbers of replicas and consistency guarantees.

Replication Goal. The goal of replication in PNUTS is for both low-latency access and fault-tolerance. An inability to respond to a client request for either of these reasons can result in lost revenue, so it is important to avoid both. Specifically, PNUTS focuses on achieving global low-latency guarantees. Their motivating application is a social networking storage system. While the owner of information may live in one country, it is quite likely

that friends or colleagues may be trying to access this information from different continents entirely.

Replica Placement. PNUTS is designed to have a replica stored in each PNUTS region. "Backup regions" may be placed geographically close to the region they are serving, but they are typically expected to be in different datacenters, spread across the globe. This allows the system to achieve extremely low-latency reads no matter where the request originates. This approach ensures that no replicas are on the same rack, or even the same datacenter, providing excellent fault tolerance. One weakness of this approach is then the time needed to query a replica that is geographically distant, which is necessary when failures occur.

Synchronization. PNUTS is built with the belief that eventual consistency is too weak. A perfect example of this can be seen from their motivating application of a social networking backend: A user wishes to post pictures from spring break, but does not want their parents to see them. First, the user removes the parents access to their photo feed, then posts the photos.

In a system which only supports transactions with eventual consistency guarantees, it is possible that the parents will be able to see the photos before the access change is processed. For such operations, PNUTS needs to be able to ensure that updates will be applied serially at all replicas.

Fine Grained Replication Providing Stronger Consistency Guarantees. While all data in PNUTS is replicated, it is actually tracked at a per-record level. Other database-styled replication schemes typically replicate at the table (or portion of) level. At these scales, latency needs to be sacrificed in order to provide stronger consistency guarantees – a user would need to lock out the whole table until an update operation is complete. Since PNUTS tracks replication on such a small-scale, it can provide the stronger consistency guarantees

without sacrificing latency. The primary copy is automatically chosen as the one closest to the most write accesses. PNUTS utilizes this in combination with its underlying communications system, YMB (Yahoo! Message Broker). YMB ensures that all updates to a single broker will be applied in order at all other brokers. Each replica is connected to only one broker, meaning that all updates to a primary are guaranteed to occur in order at all other replicas. Using a fine grained approach to replication ensures that the majority of writes are occurring through the primary.

2.4.3. Cassandra. Cassandra was developed by Facebook, with the primary goal of providing high write throughput. This runs counter to the approach found in many distributed storage solutions, where the paradigm is more focused on read throughput.

Cassandra is designed to store highly structured data accessible by a hashed row key. Any operation on a row of data is considered atomic, even if the operation affects multiple columns in this row. Cassandra nodes are expected to undergo transient failures, so relies on a journaling system to persist data to local disk. In the event of permanent failures, replicas can be used to reconstruct lost data.

Replication Goal. Replication is used in Cassandra to ensure high availability and durability. Each row in the table is replicated across multiple data centers so that even the complete failure of a single data center will not result in outages. Cassandra assumes that most failures are temporary and requires an admin to launch rebalancing operations, so nodes rely on both data persisted on disk as well as any changes that are currently being held in memory.

Replica Placement. Each Cassandra instance has a unique replication policy. Cassandra currently supports three different policies: "Rack Unaware", "Rack Aware", and "Datacenter Aware". "Rack Unaware" schemes are the easiest to implement, and simply involve having

the primary host choose $N-1$ successors along the hash ring to host the replicas. For both the "Rack Aware" and "Datacenter Aware" approaches, Cassandra relies on a Zookeeper node to coordinate replicas.

Synchronization. Replicas are synchronized on both reads and writes. Synchronization is achieved through the simple process of "last write wins". On both read and write operations, the system will query all nodes hosting replicas. Using timestamps, it determines the latest data and will schedule a repair of data for any replica which does not have the latest timestamp. For systems which need higher throughput, Cassandra will allow asynchronous writes, which can result in more repairs of data if replicas did not successfully perform the write by the time the next synchronization check occurs.

High Write Throughput for Structured Data. Cassandra was designed to allow the storage of more traditional, structured data in a distributed fashion. While it supports the idea of tables, with rows and columns, it is stored by an order preserving hash of the row keys. This means it can be easily stored with a DHT, while remaining searchable.

Cassandra further stands out in the fact that it can be optimized for high write throughput. This is far different from the 'write once read often' paradigm we saw with distributed file systems, and can put considerable stress on the hardware in a distributed system. Cassandra gets around this problem by using a journaling approach, where all writes are sequential, minimizing wear and tear on the hardware, while greatly increasing write throughput.

2.4.4. Dynamo. Dynamo [27] is Amazon's custom DHT designed to provide highly available key-value lookups for a variety of applications. While Amazon's Shopping Cart feature is the most notable customer of this tool, it is used across a variety of applications, including the catalog, which requires vastly different guarantees and use-case scenarios. Dynamo is designed to be versatile, and customizable based on a per-application basis.

Replication Goal. The primary goal of replication in Dynamo is availability. Regardless of natural disasters destroying data centers, routing problems, or hardware failures, users need to be able to view and modify their shopping cart. Any degredation in performance can be directly linked to monetary losses as customers cannot be satisfied and will take their business elsewhere. At all times, at least one replica needs to be available so users will not simply lose all data.

Furthermore, Dynamo is designed to guarantee quick responses. A slow response time could again result in lost revenue. Replicas are also leveraged to ensure quick response times to user requests. Instead of seeking to satisfy an average response rate to all queries, Dynamo seeks to meet a response rate in milliseconds for 99.9% of all queries.

Replica Placement. In Dynamo, replication is specified on a per-instance basis, setting a replication level of N for all data. Dynamo guarantees that this data is contained on N different physical machines. Dynamo refers to these replica nodes as a *preference list* which all other nodes in the system can easily compute. This means any single node in the system can quickly find all replicas of a data item.

In order to achieve better balance of data access across the hash ring, Dynamo makes use of "Virtual Nodes", where a single physical device is actually responsible for multiple hash regions. Systems which use virtual nodes need to take this information into account to make sure that no replicas are hosted on the same physical device.

Synchronization. Dynamo allows consistency guarantees to be set on a per-application basis. For applications such as the shopping cart which needs to be always write and readable, this results in very low consistency guarantees, where reads and writes will succeed with very low quorum requirements – in the case of the shopping cart, only a single replica needs to report success before success is reported to the client. While Dynamo is capable of resolving

inconsistencies between replicas without outside intervention by simply using the 'last write wins' paradigm, this is inadequate in many situations. Dynamo's true strength lies in the ability to keep track of versions of data. When Dynamo has multiple available versions, it can pass off the decision of which version to keep to the client. While this results in more business logic on the application side, it also means client applications have much more control over their data, and can take full advantage of Dynamo's loose consistency guarantees.

Customizable Storage Solution. Dynamo's goal is to provide storage for a large number of disparate applications. This need to fulfill a variety of different roles makes Dynamo stand out in this review. Instead of being developed to solve a particular problem or use case, it is designed from the ground up for multiple use cases. One of the motivating use cases is as a catalog store. This is a write once, read often use case, where the contents are uploaded in a batch process, then remain mostly unchanged for long periods of time. Another motivating use case is the shopping cart – this application requires more writes than anything else. Yet another use case is to retain user state across sessions, which requires an equal read to write ratio. By giving clients access to more of the inner workings, such as allowing application owners to set their own read/write quorum numbers, and even implement their own synchronization logic, Dynamo becomes a powerful tool.

2.4.5. Alternative Methods to Achieve Fault Tolerance. While replication is an effective method of achieving fault tolerance, it also is a drain on resource usage. With a replication level of 3, it is only possible to utilize $\frac{1}{3}$ of your total disk space. This tradeoff of availability and resource usage is inherent in replication schemes. In order to alleviate this problem, erasure and network coding schemes have been explored in the context of DHTs.

2.4.5.1. *Erasure Coding and Network Coding.* Erasure and network coding involve different implementations, but the baseline premise is the same. If you have a replication level

of 3 for all data items you store, you can only store up to $\frac{1}{3}$ of the data you could potentially hold. Erasure and network coding ensure that data is still recoverable in the face of multiple failures, but allows a reduced storage footprint. With smaller footprints, data can be stored more efficiently, and the overall cluster utilization is improved.

Network and erasure coding can build an entire data item from a subset of pre-computed codes. These fragments can also be used to increase throughput – a user can ask the cluster for the data item, and only needs to receive *subset* data items before the original requested data can be constructed in its entireity.

2.4.5.2. *Why not in other approaches?* At first glance, erasure or network coding seem like an ideal solution for storage. These approaches, however, are only seen in the area of distributed hash tables. One problem with erasure and network coding is lack of flexibility. An entirely new encoding must be calculated if the encoded data is modified. In a distributed file system, files are written incrementally by running computations. While these systems eventually hit a stable point (they generally follow the write once, read many times paradigm) the initial creation and update process would swamp the system with continuous changes needed for erasure or network codes.

2.5. Replication in Distributed Stream Processing Systems

Where fault tolerance through replication has several concrete examples in both distributed file systems and cloud runtimes, it is more of an open problem in the field of distributed stream processing systems. Constantly incoming data adds a new dimension of difficulty to the problem of fault-tolerance. Unlike cloud runtimes where input data is saved to file, simply restarting a process in a DSPS implies a heavy loss of data. While an active approach would ensure no data is lost in the event of failure, it also implies a replication of

resource usage. This may be a problem in a system which is hosting many computations simultaneously. If a passive approach is used, there is a much smaller footprint, but extra care needs to be taken to ensure that no incoming data is missed after failure has been detected. A passive replication scheme may also need to negotiate the transfer of state information between replicas if the computation builds state while executing.

Processing in a DSPS is further complicated due to the way computations are typically staged. As mentioned previously, a DSPS computation generally consists of multiple smaller processes chained together. All these stages of a computation need to remain active at all times, to handle streaming inputs. This means that any single machine may be hosting portions of multiple computations at a single time, so the failure of a single machine may disrupt multiple computations managed by the DSPS.

DSPSs expect a continuous flow of incoming data, and have been designed to perform SQL-like queries on these streaming datasets. They have been deployed in fields such as stock analysis, telecommunications management, as well as host threat detection [28]. These approaches rely on distributed stream processing middleware such as Synergy [29], Borealis [30], or System S [31] as a basis for communications, which help forward data correctly, and can help place primary replicas. Work on replication in this environment has been extensive, with several different approaches analyzed.

In this section, we use a slightly different format for our analysis. Much of the research on replication in DSPSs focuses on a portion of the replication space, such as placement or synchronization schemes. Due to this, we have altered our analysis format accordingly.

2.5.1. Replication Goals in Distributed Stream Processing Systems. In DSPSs, the goal of replication in passive schemes is purely for fault-tolerance; should the primary fail, there is another replica willing to step up and take over processing. In active replication

schemes, however, there is usually more variation in replication goals. For example, the main goal of replication in [28] is to increase processing speed. It is able to increase processing by a factor of `N`, where `N` is the number of replicas in the system. One interesting thing to note about this approach is that it assumes a stateless computation as well as a large amount of incoming data. In this approach the incoming data is split up among the replicas and processed independently at each. Since there is no state maintained at any of the replicas, only allowing a replica to see a portion of the input will not effect later computations.

One weakness to this approach is that by using all replicas in unison we are essentially losing some measure of fault-tolerance. Instead of a typical active replication scheme where any replica can step up in case of failure, we are now in a situation where losing one replica can mean the loss of a portion of the input data.

Active replication schemes can also leverage replication to help reduce latency. Instead of waiting for the primary node to provide a result, the consumer can simply use the first result provided – this helps to alleviate any network delays. The program flow for this type of active replication is shown in Figure 2.5: The upstream client sends inputs to all replicas, and the downstream client receives output from all the replicas. This approach has the highest overheads, as all replicas are running all the time and every replica is receiving all inputs and producing outputs (for `N` replicas, `N` times the networking bandwidth is used). On the other hand, there is no cost if failure occurs – inputs and outputs are already set up and the correct state has been built up at each replica. The system then doesn't even need to recognize that one of the replicas has failed, the consumer simply continues to choose the first output produced.

There are also active replication schemes which exist solely for fault tolerance. An example of this is *active standby* as explored by [32]. While losing the ability to speed up

processing by splitting inputs across all replicas or reduce latency with a winner-takes-all approach to output, active standby maintains the ability to build consistent state at each replica. All replicas receive all inputs, but only the outputs produced by the primary are passed along to the next stage of the operation. This helps to reduce networking overheads, while maintaining the lower failover times found with active replication schemes. The program flow for active standby is shown in Figure 2.6: While all replicas receive inputs, only the primary is passing a result to the downstream client, which can add a slight delay to recovery times – once a failure has been detected, the outputs from the new primary need to be redirected.

2.5.2. Replica Placement in Distributed Stream Processing Systems. Placing replicas in a DSPS is not a simple task. Unlike DFSs or cloud runtimes, DSPSs generally have a chain of tasks which all need to be performed sequentially, in real time to successfully process incoming data. The placement of replicas for all the tasks which make up a processing chain can be vital to the performance of the overall process as well as any other processes hosted by the cluster.

This problem is discussed in detail by [33], who design a scheme where replicas are placed in a distributed manner, using a distributed hash table. This removes the need for a centralized machine which is responsible for all replica placement – an approach we see in every distributed system discussed so far in this chapter. In this approach replicas are placed in series on a per-process basis. First, the primaries of each task in the process are placed one at a time in the cluster. Once all primaries have been placed, each primary negotiates for placement of its replicas.

While this is an interesting approach which could prove valuable to many implementations of replication, there are questionable assumptions the authors make in this paper. First, the

authors base their placement algorithms on a simple experiment: processes of 3, 5, and 10 components are placed with a replication level of 2. The authors varied the number of nodes in the cluster holding tasks and calculated the application availability given all combinations of 5 nodes failing. From this experiment the authors decided that all tasks in a process should ideally be placed on the same machine. This assumption then drives their placement strategy.

If all tasks in a process are placed on the minimal amount of machines, you are actually increasing your chance of catastrophic failure – there is a greater chance that failure will remove all replicas, instead of leaving a chance for re-replication and recovery. While with only two replicas there is a good chance that the process will not survive if enough machines fail, trying to reduce the number of nodes which host a process is too simplified of an approach.

What this approach does do, however, is cut down on latency and reduce networking overhead. Instead of passing data between possibly distant machines, all tasks in a process may be hosted on a single machine. Their algorithm will additionally attempt to place tasks which make up a process as well as their replicas as physically close to each other as possible. Most distributed systems recognize the possibility of machines which are physically near failing together – from machines in a rack losing power simultaneously to the possibility of losing an entire data center due to a natural disaster. These tests are run on PlanetLab [34], which could mean that nodes are spread around the world and reduce the worry that physically near nodes will necessarily fail together. Unfortunately, the authors do not list where their PlanetLab nodes were located, so it is not clear that the machines chosen for replicas were spread out enough to ensure they were located in different failure zones.

The distributed method of placing tasks and their replicas is an interesting approach, but it is not clear that their tests support this placement scheme. From the initial assumptions, it seems like this approach would not be a good fit for a cluster within a company or institution – the replicas would be placed close to their primaries, and would then be susceptible to complete failure should a rack or data center be subject to an outage.

Another approach to replica placement is described in [35]. This approach involves first splitting up tasks into minimal logical sections, or atomic units. The idea behind this is that smaller computation units means less state per unit. As the goal is to use checkpointing to assure no state or messages are lost due to failures, lowering this cost helps to speed up the checkpointing process. By splitting up tasks, the authors are also introducing the notion that a single machine may hold parts of different tasks – this is a major difference from their previous work, as well as from that found in most DSPSs. Generally, tasks are kept as atomic units, so little work has been done to explore this idea of breaking a process into smaller pieces.

After a task has been split, the atomic query units are then placed on separate machines. Once each atomic unit has been placed, each unit then looks for other machines to use as backups. The authors claim that their system is capable of analyzing these atomic pieces, and can co-locate similar streams, while ensuring no single machine becomes a bottleneck. The placement strategy relies on servers volunteering to take on more load when under-loaded, meaning an overloaded system may struggle to maintain replication levels. From the paper, it seems that there is a possibility that a single computation may be offered hosting opportunities across the cluster, so some work was done to ensure that the atomic units are 'fairly' replicated and no one unit exists on every machine.

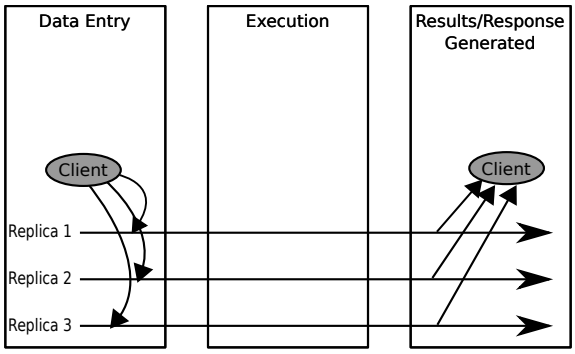

FIGURE 2.5. Flow of communications for latency-reducing active replication

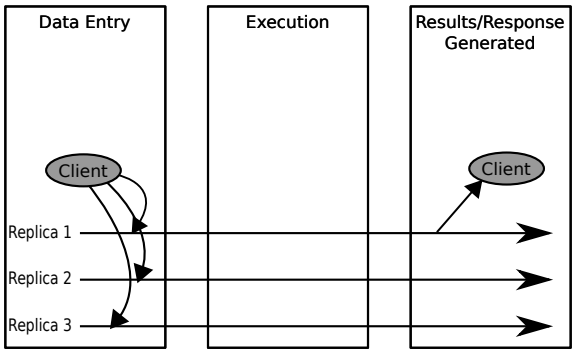

FIGURE 2.6. Flow of communications for active standby

2.5.3. REPLICATION LEVELS IN DISTRIBUTED STREAM PROCESSING SYSTEMS. Most DSPSs have a replication level of 2 [35, 32, 28], with others using a replication level of 3 [36]. This is very different from DFSs where the default level of replication is three across the board. This is most likely a result of the lower latency requirements inherent in DSPSs – data usually needs to be processed in real time and it may not be worth the gains in fault tolerance to overload the system with replicas.

One interesting approach mimics what we see with Hadoop and MapReduce, where only one replica is active at a time. This approach is discussed in [37], which presents a novel approach to fault tolerance through replication. To keep a minimal footprint, replicas are only instantiated when failure is imminent. To test this theory, the authors introduced two types of predictable failure: user error and impending machine failures such as overheating or failing hard drives.

The key to this approach is to only start a new replica when necessary (and even then only a single one). The primary can then perform a checkpoint to ensure the replica is up to the current state. All input and output streams are then redirected to utilize the replica. Once all processing has fully switched over, the old primary can be gracefully halted.

2.5.4. Synchronization in Distributed Stream Processing Systems. With active replication approaches [36, 28] in DSPSs we again see no synchronization systems in place, just as in cloud runtimes. The passive replication schemes in DSPSs, however, often involve some synchronization or checkpointing scheme in order to preserve state. Before reviewing these schemes, it is important to note that not all passive schemes implement checkpointing. An example of this is found in [32], where the approach is referred to as gap recovery, or *amnesia*. As soon as a replica detects that the primary has failed, the replica promotes itself to the primary instance and redirects inputs. The amnesia layout can be seen in Figure 2.7. There are 3 replicas in this approach, but the client is only communicating with the primary, replica 1. While replicas 2 and 3 have been instantiated, and are taking up the required memory footprint to remain at a ready state, there is no communications with these replicas reducing the I/O overhead. In the amnesia approach, replicas periodically send heartbeat messages to the primary to determine if they need to take over processing inputs. At that point in time the client input stream and results stream will be redirected to make use of the new primary. In an approach where some external mechanism is used to track whether the primary is alive, even the minimal heartbeat messages from the offline replicas can be foregone.

[32, 35] explore several checkpointing options to transfer state between replicas. Where DFSs constantly apply updates, these approaches use a delayed checkpointing scheme. In a checkpointing approach, the primary periodically pauses operations and saves state. It

then pushes the state information to all of its replicas. Once the replicas have updated, it is safe for the primary to delete the last checkpoint and start to create a new one. One of the major research areas of this problem is when to perform the checkpoints and subsequent updates. By delaying a checkpoint/update cycle, it takes significantly longer for the system to recover from failure. Alternatively, a constant update schedule leads to the reduction any gains made by using a delayed checkpoint – every replica requires a larger processing, network, and memory footprint.

The generic program flow for a passive, stateful replication scheme can be seen in Figure 2.8. While the client only communicates with the primary replica, the primary will push state information to the replicas between rounds of execution. This communication step may occur at any point in the program flow. It halts any new processing while state is saved on the primary, then continues after the save point has been made. It is then up to the primary to negotiate writes of state information to the backup replicas. Since we are assuming a constant, streaming environment, it is likely that data will be incoming while a checkpoint is taking place as well as between checkpointing breaks. Implementations can choose to either ignore any missed state from a failure between checkpoints – assuring an inconsistent state after failure, or upstream clients can be responsible for keeping a queue of inputs between checkpoints. While this may drastically increase overall recovery time, due to the need to 'replay' data, it also provides the strongest consistency guarantees.

The work in [35] is a follow-up of their previous work [32], focusing entirely on what was previously described as a passive standby approach. In this work they pay close attention to the checkpointing schedule, and compare two different approaches to checkpointing. A round-robin approach as discussed in their previous paper [32] and a new min-max approach.

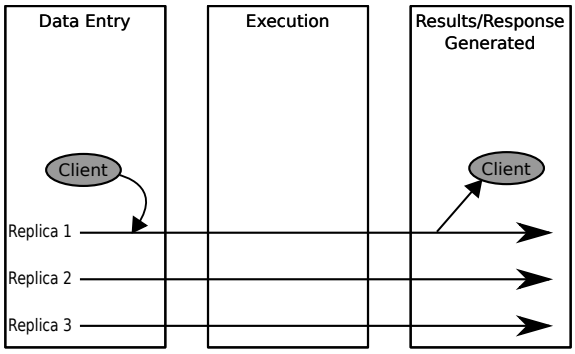

FIGURE 2.7. Flow of communications for amnesia approach

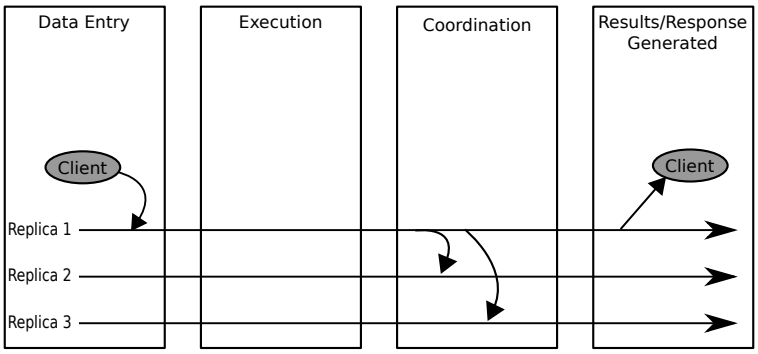

FIGURE 2.8. Flow of communications for a checkpointing DSPS

While the round-robin approach provides a fair schedule for checkpointing, it can easily lock up waiting for free cycles to create a checkpoint, and then wait for all machines hosting replicas to find the time to apply the checkpoint update. To alleviate the long delays between checkpoints seen in the round-robin approach, the authors proposed a min-max scheme. In this approach, a primary creates a checkpoint, then lets replicas update when they have free processing power. Every atomic unit further keeps a log of all outgoing data, which is periodically flushed once replicas confirm receipt and application of checkpoints. By using this approach there is a tradeoff between memory usage and delays in processing.

2.5.5. DISTRIBUTED MONITORING IN DISTRIBUTED STREAM PROCESSING SYSTEMS. In this section we have seen several different replication approaches from distributed stream processing systems. Since DSPSs have a wide variety of needs depending on application,

TABLE 2.1. A summary of the distributed systems discussed in this paper

System	Replication type	Replication Goal	Replica Placement	Replication Level	Synchronization
GFS	Passive	Fault-Tolerance and Availability	Centralized	3	Constant Checkpoints
HDFS	Passive	Fault-Tolerance	Centralized	3	Constant Checkpoints
Azure	Passive	Fault-Tolerance	Centralized	3	Constant Checkpoints
MapReduce	Active	Fault-Tolerance	Centralized	1	None
Hadoop	Active	Fault-Tolerance	Centralized	1	None
Dryad	Active	Fault-Tolerance	Centralized	1	None
BOINC	Active	Correctness and Fault-Tolerance	Controller	N	None
DSPSs	Active and Passive	Fault-Tolerance, Reduce Latency, Increase Processing Speed	Distributed and Centralized	2 or 3	None or Delayed Checkpoints

replication approaches in DSPSs are similarly varied. The need to handle continuously streaming data leads to the need of much stronger processing speed requirements than seen in DFSs and cloud runtimes. DSPSs also have a unique need to have all stages of a computation active and running constantly, leading to tighter resource constraints in general.

Another trend found in DSPSs is the need for distributed control. In Hadoop, GFS, and Dryad there is a single node which is responsible for orchestrating the computation, as well as monitoring the health of nodes in the system. Several of the DSPSs discussed here [35, 32] have a distributed monitoring system, removing the single point of failure that a single master node represents resulting in a more robust environment.

2.6. DISCUSSION

In most distributed settings, replication is used solely for fault-tolerance. In this survey we have explored additional uses, such as increased availability, decreased latency, correctness, and faster processing speeds. A summary of the distributed systems discussed here

can be seen in Table 2.1. While we can broadly classify replication schemes as belonging to either a passive or active approach, within these definitions there is a lot of room for variation. Each approach has strengths it was designed to handle, and weaknesses which my not be initially apparent.

Where many cloud compute runtimes generally forsake replication when backed by a distributed file system, this can be disastrous in a distributed stream processing system where inputs are not backed by any storage system and may be coming in at a continuous rate. Most DSPS computations are assumed to be well-defined SQL-like queries which can handle streaming data. MapReduce and Hadoop, on the other hand, are expected to handle arbitrary computations which fit the mapreduce paradigm making them more flexible frameworks, but without the ability to handle streaming data. While Dryad was designed to work with SQL-like queries, it still needs to be able to handle generic programs and again cannot support streaming data. BOINC supports completely arbitrary computations, with data and executables sent to volunteer nodes as needed. BOINC is not, however, designed to handle streaming data. It assumes all data arrives with the computation at the beginning of the processing phase.

Through this work, we have seen several trends emerge. For example, the distributed file systems discussed here all have passive replication schemes, with quick checkpoints this matches the conclusions from [15]. All the cloud runtimes we looked at have active replication schemes, generally with low levels of replication. While initially they appear to be passive replication schemes, all replicas receive all data meaning they are ultimately active replication schemes. When moving to DSPSs, on the other hand, we finally see variation in replication schemes.

This trend does seem to hint that we simply may not have performed a broad enough sweep of DFSs and cloud runtimes in order to fully capture what schemes are available. This, however, is simply not the case. With DFSs, we have looked at the premier open source solution [19] as well as Microsoft's solution [22, 23]. While there are rumors of a redesigned GFS2, no work has been officially published as of this writing. For cloud runtimes, Hadoop is widely accepted as the de facto standard for cloud runtime performance. To flesh out the section, we looked at MapReduce, which is the basis for Hadoop, as well as Dryad and BOINC which follow drastically different paradigms.

One major gap in distributed systems is an environment which supports arbitrary computations in a streaming environment. This gap is filled by Granules [9, 10, 38]. To fully support arbitrary computations, it is also important to support many different replication schemes simultaneously. For example, a system which only supported passive replication schemes would not be a good choice for any computation which cannot afford long failover times. On the other hand, if the system only supports active replication non-deterministic computations would have no consistency guarantees. As of this writing, no work has been done exploring the support of disparate replication policies in a single environment. Furthermore, such an environment cannot assume a DFS backend, so it cannot leverage the cloud computing approach of a default of only one replica. It should additionally be able to support correctness schemes such as found in BOINC: allowing functionality within an unsafe environment as well as the ability to adapt to drastic changes in resource availability.

We also found an interesting trend with respect to replication levels. DFSs tend to have three replicas, while DSPSs lean towards two replicas. One area for research would be implementing a hybrid replication scheme of three replicas. In this scheme two of the three replicas would be processing inputs in parallel (an active approach), while the third replica

waits in a dormant state. On the failure of one of the active replicas, the passive replica could promote itself to an active scheme. This means a resource overhead just slightly greater than having 2 actively replicated tasks, and we can assure quick fail-overs and stronger consistency guarantees than with a purely passive approach. While this would not work in a DFS environment, and would be overkill for a cloud runtime such as Hadoop or Dryad, it could be a worthwhile approach in a streaming environment with the need to quickly respond to failures while resources are at a premium.

Another area for expansion is using a passive or active standby approach to support migration of processes from overloaded machines. If the machine which is holding the primary copy of a replicated computation becomes overloaded, it should be possible to transfer primary responsibilities to one of the replicas. This could be a useful load balancing technique in an overloaded cluster where there is no room to migrate the replica, but moving communications responsibility away from a machine may help improve overall cluster performance.

CHAPTER 3

Requirements

In order for a stream processing system to become a viable solution to relieving stress on the health care system, such a framework needs to fulfill several requirements. These requirements need to be met in order to ensure patient safety; such as to continue processing with a minimum of lost packets in the face of failure, or preventing collocated computations from causing interference which precludes real-time processing of data. While there are many other concerns, mostly relating to patient privacy, we consider those concerns to be outside the scope of this dissertation.

3.1. Computation Support

Before discussing in detail aspects of the framework that we need to support, it is important to discuss the types of computations we are expecting to support. We expect that computations build and maintain state over time. This requirement arises due to the nature of health streams – each individual data packet is small, but the associative stream is long-running. The amount of information that can be gained from any one data packet is relatively low when compared to the information gained from observing packets over time. Take, for example, monitoring thorax extension for a sleep study. While "awake" breathing patterns are normal when a person is awake, seeing such a pattern in the middle of "deep sleep" breathing patterns may be a sign of sleep apnea and would need to be studied further. This base assumption that we need to ensure state is maintained drives our remaining requirements. An in depth look at each of the computations we support, as well as the underlying data streams for each, is provided in Chapter 4.

3.2. Real-time Requirements

One major requirement to this system is the ability to process streams in real time. Many current health stream processing systems are fully capable of batch processing large quantities of medical data after the data has already been recorded. While this can be very useful for diagnosis after the fact as well as long-term studies and trend detection, it is inadequate when it comes to timely interventions, particularly in cases where a patient has left the hospital for in-home recuperation, or elderly preferring to age at home instead of moving to a full-time care facility. If a sensor-based system cannot react to changes in patient state when they occur, this system will not be able to provide a safe alternative to full-time care.

Part of providing a safe transition from a full-time care facility where the patient may be constantly monitored by caregivers, to a more independent lifestyle backed by sensors, involves a system that can reliably process data in real time. One example of why this requirement is necessary is in the case of a stroke. With a stroke, time lost is brain lost – if EEG data is not being processed in real-time, stroke detection may not occur until after irreparable damage has already been done. Similar arguments can be made for heart attacks, or even falls in the home. Emergency situations need to be detected as soon as possible to ensure appropriate responses are launched in time to prevent patient injury.

Another problem arises due to the nature of streaming health data. Health sensors generally produce small amounts of data at sub-second rates. While each packet is generally on the small side, the rate at which packets arrive can easily overwhelm a system that is running slowly. If data cannot be processed as fast as it is being collected, there is the risk of losing data – eventually queues will overflow, resulting in dropped packets. Since this

is a streaming environment, any dropped packets are lost forever, possibly resulting in a misdiagnosis or missing a medical emergency.

3.3. Robust to Failures

Another requirement for such a system is the ability to recover quickly and gracefully from failure. This is a challenging problem in distributed systems due to the inability to detect the difference between a failed machine and a slow machine. As stated above, since we are working in a streaming environment any data that is lost is lost forever. Failures need to be detected and recovery measures need to take place as soon as possible to minimize data loss. This means that any solution needs to include two parts: (1) How do we reliably detect failures? (2) How do we respond to these failures?

3.3.1. Reliably Detecting Failures. There are two extremes which can be used to approach the first part of this challenge. First, we can be overly sensitive to failures. A failure can be declared as soon as there is any slowdown in communications. While this means that we will always respond to a true failure as fast as possible, it also means that we are wasting a lot of time on false positives, i.e., declaring a failure when one has not actually occurred. Reacting to false positives strains a cluster with needing to instantiate new computations that have not actually failed, and state synchronization messages to try to bring these new computations up to speed. In a large-scale environment, this approach is infeasible due to the heavy price of reacting to false positives.

At the other end of the spectrum, one can choose to put off declaring a failure for as long as possible. The longer the wait to declare a failure, the more certain one can be that it is a true failure, and not simply a result of network congestion or a slowdown at the machine being contacted. This can lead to instances where a computation has failed with no alert

raised. All communications to that computation are effectively lost, meaning irreplacible data is now gone forever. This is not a suitable solution for any streaming system, and far worse for a system involved in a health care setting.

Our approach balances these two concerns – generating a large number of false positives which overwhelm the system against losing important data as a failure goes undetected. Our system uses a decentralized heart beat approach, where every machine in the system is responsible for detecting failures on every other machine in the system. This is done in a way that prevents strenuous loads on the system, while also limiting false positives. It is configurable with metrics defining the sensitivity of the system to failures. This approach is discussed in detail in Chapter 5.

3.3.2. Responding to Failures. After we have detected that a failure has occurred, we need to respond to it. For this, we make use of *replicas*, or duplicate copies of computations. We fully define and analyze this approach in Chapter 6. For situations where replicas are not already processing incoming data, the first step to failure recovery is to ensure that communications are redirected. Once failure is detected, all replicas are notified that this has occurred, so another can take over the role of processing incoming data. The next step involves starting up a new computation to replace the one which was lost. This process is called *re-replication*, and preserves the redundancy levels associated with a particular computation. This ensures that the next failure will not wipe out a computation entirely.

Choosing where to place new replicas can be a difficult task and is fully explored in Chapter 7. In the most basic terms, replicas should be placed on failure independent machines. Placements should ensure that a new replica is not placed on the same machine as any other replica – this would preclude failure-independence of replicas, as it would mean that now

not one, but two replicas are lost on a machine failure. The task of effectively placing a new replica can be complex, and needs to take into consideration not only the location of existing replicas, but also the load and capabilities of all potential replica locations.

3.4. Robust to Interference

The system additionally needs to be robust to interference. Computations that are co-located should not affect each other. This is a problem which is a result of trying to fully utilize each resource. While a single machine could easily host a single computation, this is a highly inefficient use of resources. Such an approach would also preclude the ability to support a hospital or nursing home on a small cluster, with even a small number of patients requiring dozens if not hundreds of physical machines. To effectively utilize each resource, we use Granules' ability to effectively interleave hundreds of computations on a single machine. While we will not be able to scale at this rate for all computations, as each has its own profile of resource requirements, we can still hope to support dozens of co-located computations on a single machine.

In order to effectively interleave computations, we need to be able to perform three tasks: *(1)* effectively profile each computation, *(2)* use this profiling information to detect interference, and *(3)* take action to mitigate interference.

Data generation rates on different health streams may be different. Each patient will have a slightly different schedule, i.e., patient A is more active between X and Y hours of the day, resulting in higher sampling rates during these hours; while patient B needs to be monitored more closely while sleeping, between the hours of Y and Z. While patient A and B may be using the same set of computations, their usage patterns would result in computations with vastly different profiles. It is not enough to simply profile each type of computation and use

these statistics to drive all interference detection, instead we need to profile computations on a per-patient basis, with continuous monitoring. We expect these computations to be extremely long-running, on the order of months to possibly several years. Over the lifetime of these computations, usage patterns are expected to change, at the very least with the change of seasons if nothing else, as daylight hours fluctuate.

After we have profiled each computation, we next need to be able to utilize this information. By using information about past behaviors, we can extrapolate future trends, and create both current and expected future profiles of each computation. Our exact approach is outlined in Chapter 7. Using this information, we then take the next step of both reacting to current detected interference as well as avoiding predicted future interference. This is done through a process called *migration*, where a computation is moved between physical machines. This task is difficult since it needs to be performed while continuing to process incoming streaming data in real time. We describe our approach to migration in Chapter 8.

3.5. Throughput

The throughput of a stream processing system is defined as the number of packets processed in a timely fashion over a period of time. A system which cannot maintain a high throughput will not be able to meet any of the previously discussed requirements: real-time processing, robustness to failures, and robustness to interference. A failure to meet any of these requirements would result in a corresponding drop in throughput, making it both a goal to be met as well as a measure of success.

Our success in implementing a framework for robust health stream processing will be measured via throughput. For the purposes of our work, we are measuring throughput as: the number of completely processed packets per second. For computations that require a

response to be sent to a patient, we do not count a packet as fully processed until the patient has received the response. Given both the different data generation rates and different processing requirements of each computation, each computation will also have a related "best" rate, which would be the theoretical best possible throughput if network overheads are ignored.

While it is important to try to maximize the overall throughput of the system, we also need to ensure that throughput is maintained on a per-computation basis as well. For example, take a machine which is hosting multiple computations, among them computation A and computation B. The machine may have a high overall throughput, but an extremely high throughput for computation A may be masking the poor performance of computation B. This could be a result of the processing for A interfering with the processing of computation B. Because of this, we cannot rely solely on overall throughput, but need to take into account individual computation throughput as well.

CHAPTER 4

Components

This chapter provides a high-level overview of the components developed for this dissertation. These components will be described in more detail in the appropriate sections.

4.1. Datasets

We use three distinct datasets to test and motivate our framework. These datasets are all publicly available health stream datasets, and have been selected to provide a broad spectrum of data types and generation rates.

4.1.1. Electroencephalogram. The electroencephalogram (EEG) dataset we use in this work was gathered by Colorado State University's Brain Computer Interface (BCI) lab `http://www.cs.colostate.edu/eeg/`. This particular dataset was generated by an able-bodied male in his 20s, using a NeuroPulse Mindset 24R amplifier with 19 electrodes in the international 10-20 configuration, at a 512 Hz sampling rate. The dataset was gathered specifically for a BCI application – it records multiple 5 second bursts of each of 4 different tasks. These tasks involve: counting backwards from 100 by 3s, imagined right hand movement, imagined left leg movement, and visualizing a rotating computer screen in 3 dimensions. This dataset was gathered across 5 trials, where 10 sequences of each task was performed for 5 seconds. Each of these tasks should involve higher levels of activity in different sections of the brain, making them good tasks for a 4-way classification problem.

4.1.2. Electrocardiogram. We use the MIT-BIH arrhythmia dataset [39], publicly available through the physiobank database [40]. This dataset has been developed with the express purpose of evaluating arrhythmia detectors and training doctors to recognize

arrhythmias. It has also been used in several machine learning competitions. This dataset contains recordings from 47 patients, and contains 48 half-hour excerpts of two-channel ambulatory ECG recordings. This dataset has been hand annotated by trained professionals, and the digital version was originally intended to develop tools for real-time ECG rhythm analysis.

4.1.3. Thorax Extension. The thorax extension dataset [41] we use was gathered by Dr. J. Rittweger at Institute for Physiology, Free University of Berlin. Thorax extension data monitors the expansion and contraction of the chest – it monitors respiration rates. This information can be used at higher level scales such as: is the patient breathing properly on their own? It can also be used for more fine-grained information such as: What stage of sleep is the patient in? This information can be useful for both sleep studies and pre- and post-operative patients. This dataset represents a class of single-dimension output health sensors, where a relatively small amount of data is generated at a steady rate.

4.2. Sample Computations

In order to utilize our datasets for testing, we have developed a set of sample computations. Each dataset has its own computation, with unique processing and resource requirements.

4.2.1. Brain-Computer Interface Application. The EEG computation we use [42] has been developed to process raw EEG data in real-time. It assumes that this data is being processed for a BCI application, so will return a classification to the user. We are using a group of experts approach for each user. Each user has several artificial neural networks (ANNs) which have been trained on previously recorded user data. Each ANN will learn a slightly different thing about the dataset, become an expert about their portion of the input

space. The ANNs then all 'vote' on a final classification, providing a group of experts. These ANNs each have a single hidden layer, and we limit the number of training rounds that each network undergoes. This has a two-fold benefit. First, it reduces the amount of training time needed – as users will need to periodically stop and retrain a network over extended periods of use, this limits the amount of time the system needs to spend in the training process, unable to process new data. Secondly, it also prevents overfitting. Overfitting occurs when an ANN becomes unable to handle new, unseen data patterns.

Using our group of experts approach, each ANN is randomly assigned different starting weights. This means that each ANN will react to incoming data in a different way. The goal of such an approach is to allow each ANN to learn a slightly different part of the solution space. While no single ANN would be able to answer everything, working as a group we should be able to get decent results. Each ANN will 'vote' on a classification, with a final classification coming from the group consensus.

Each user has a single, unique computation that comprises multiple ANNs all implemented in the programming language R. These ANNs return a consensus, which is then returned to the user. In addition to user computations, each machine also runs a generic trainer. The generic trainer is trained on new data as soon as any user with a computation hosted on the same machine submits new data for training. This allows each machine to become a reflection of the users hosted on it – it will become more capable of providing a 'generic' set of ANNs to new users, providing a boost to both initial user startup, while also providing new insights into how the brain works. We leverage Granules bridges [43] to communicate between the Java-based framework and the R-based ANNs.

4.2.2. ECG Anomaly Detection. The ECG computation we have developed has been designed to detect arrhythmias. Upon detection, the computation will store the last

10 seconds of data to disk for later analysis, and continue to store all incoming data to disk until 10 seconds have elapsed without an arrhythmia. Such a computation would be useful in a situation where a patient is under observation. By storing data preceeding and following the event, it would allow a professional to analyze what led up to the event, helping with the diagnosis and treatment process. This computation has a single monitor per patient, and does not report data back to a user. Its primary purpose would be as a tool to help doctors diagnose patients by filtering the raw ECG data.

This computation requires that each second of data be sent to an artificial neural network (ANN) for classification as either a possible arrhythmia, or normal data. This ANN is also implemented in R, so we again make use of Granules bridges [43] for communication between our Java-based framework and R. When each computation starts, a stored ANN is loaded into its associated R computation. Each computation needs to store the last 10 seconds of data in memory, meaning that this computation has a high memory footprint. During periods where it needs to write to disk, it will also have high disk I/O accesses.

4.2.3. Respiration Monitor. The thorax monitoring computation we use here is designed to act as a backend for a visual monitoring application. It retains the last 10 seconds of data in memory for display purposes, while also maintaining a running average, minimum and maximum of the values seen so far. All this information would be useful for a visual display of the patients respiratory rates. While it also stores 10 seconds of information in memory, just like the ECG computation, it requires significantly less resources. This is primarily because the thorax dataset is one dimensional; unlike the EEG dataset where data is being gathered simultaneously from 19 electrodes, and the ECG dataset which has two electrodes generating readings at each timestep.

4.3. System Elements

Our framework relies on a series of system elements which provide both information gathering and the ability to make system-wide decisions about how the resource pool reacts to changes. We gather information about machine and computation state and then leverage this information to recover from failures, make informed migration decisions, and decide where to place new computations.

4.3.1. `HeartBeat` System. The `HeartBeat` system underpins the system's ability to identify and respond to failures. Our system is unique in that it is a completely decentralized approach. Furthermore, it is tunable as needed. For example, a power user would be able to trade off between the certainty of a failure claim and the amount of time it takes to make such a decision. This level of customization is unique in our approach, and makes it amenable to a variety of situations. For example, different guarantees may be required for a retirement home than for a hospital's surgical center.

While this was designed with monitoring health sensors in a care environment in mind, it is also generally applicable to sensor processing systems. For example, a system which connects multiple autonomous robots may use this to allow the robots to keep track of each others liveness. The system can be adjusted to allow only intermittent communication of state, allowing the robots to preserve battery power. Alternatively, it may be used to back a sensor system which monitors seismic activity. During 'interesting' periods, the failure detection system could be kicked into high gear, to ensure no data is lost. Otherwise, the system could revert to a reduced detection rate to preserve battery life.

4.3.2. Resource Monitors. Each machine in the resource pool is additionally responsible for hosting a `ResourceMonitor`. This component is responsible for tracking the current

load on a system. It gathers information from each hosted computation that is eligible for migration. This excludes the `HeartBeat Monitors`, generic trainers from the BCI application, and the Resource Monitor itself. Based on the numbers and types of computations currently hosted, it develops a load profile for itself, and self-labels as either underloaded, stable, or overloaded. An underloaded machine is willing to accept new computations and migrations, while an overloaded machine will actively try to migrate computations away from itself.

4.3.3. Computation Statistics. Each computation gathers computation statistics about itself, and periodically sends this information to the collocated `ResourceMonitor`. These statistics are used to detect interference between computations, and combine information about activation schedules and resources used. Each `ComputationStatus` gathers:

- **Queue Length**: The average amount of packets waiting for processing seen over the last sampling period
- **Run Time**: How long the computation has spent executing
- **Data Packets**: Average number of incoming data packets needed for processing seen over the last sampling period
- **Control Packets**: Average number of incoming internal communication packets seen over the last sampling period
- **Data Bytes**: Average size of data packets
- **Control Bytes**: Average size of control packets
- **Activation Times**: Tracks when the computation has been activated in 10ms intervals
- **Completed Count**: Tracks the number of data packets that have been fully processed in the last sampling interval

- **Replica Locations**: Used when working with replication levels grater than one

These statistics are updated every time an event occurs which modifies the values, and is reset after the `ResourceMonitor` requests the current status. For example, when a data packet arrives, the *Data Packets* field is updated, as well as the *Data Bytes* and *Activation Times* fields. This approach allows statistics to be gathered on a configurable basis. This affects the rate at which the system can adjust to changes in load as well as the amount of overhead generated by systems communications.

4.3.4. Interference Detector. The `InterferenceDetector` is designed to gather information from all computations to detect and predict interference. It uses the information from the `ComputationStatus` reports from each computation to develop an activation profile. Based on the current activation profile, it then predicts future execution statistics. We then perform clustering on a vector of current and predicted activation profiles. Computations which have similar activation schedules and receive similar amounts of data will be in the same cluster, meaning that they are more likely to interfere with each other. The system will attempt to migrate interfering computations away from each other to reduce overall interference.

4.3.5. Coordinator Node. The `CoordinatorNode` is responsible for coordinating interactions between system components. As it gathers information from all the components, it is also responsible for using this information to both guide initial computation placements and coordinate and orchestrate migrations. In order to ensure that no conflicting decisions are made, we ensure that there is only a single node responsible for making all these decisions. While this may be seen as a single point of failure, a system of passive replication and failover can ensure that a failure will not bring down the system.

4.4. Communications

In our system, we have two main types of communications occurring at all times: Data and Control. While the primary purpose of the system is to support the processing of incoming data packets, the additional control packets are what allow us to adjust to changes in either system state or changes to the flow in data. These types of communications are competing for limited network bandwidth, requiring a balancing of priorities to ensure the system can still maintain a high throughput of data traffic while retaining the ability to respond quickly and reliably to changes.

4.4.1. Data Traffic. Data traffic is the traffic generated by sensors, which is being fed into our system for processing. As discussed above, it may be subject to changes in data arrival rates, as dictated by an external source. Additionally this includes information which needs to be passed between replicas: both state updates and any replication in raw data and processing. We consider data traffic to be all communications needed by the computation – including communications which are needed as a result of replication. We differentiate this from control traffic, which exists solely to keep our framework running and allow intelligent placement – communications between user computations and system computations, as well as communications between system components.

The main purpose of our system is to support the processing of data traffic robustly. System performance is measured by the amount of unique data packets processed. This is a tricky measurement because it does not directly take into account either communications between replicas or replicated data processing. Communications between replicas and replicated data processing are, however, still vital data traffic which has precedence over control traffic.

4.4.2. Control Traffic. We consider all communications which are necessary for system functionality to be control traffic. The throughput of these messages is not considered directly when measuring system performance, but the flexibility of the system and speed of response to changes is. Our system has 4 main sources of control messages: deployments, heartbeats, migrations, and interference detection.

Deployment messages are sent whenever a new computation requests to join the system. It can be expected that such a request comes from an external source outside the resource pool, and is directed to the `CoordinatorNode`. The `CoordinatorNode` then uses information about the current state of the resource pool, as well as expected computation overheads, to generate an informed placement plan. It then pushes the new computation, along with all requested replicas, to the designated machines.

Heartbeats are one of the continuously running control messages that our system requires. Heartbeats are sent out regularly to determine the liveness of each machine in the resource pool. These types of messages are discussed in more detail in Chapter 5.

Migrations are performed when interference has been detected, and the system is taking steps to mitigate this interference. Depending on the type of migration, this can involve multiple messages between the `CoordinatorNode` and all computations affected by the migration. Migration messages are not continuous, but only occur when a migration begins, and will end once the migration is complete. Unlike the heartbeat messages, it does not represent a continuous load on the system. Migration details are discussed in Chapter 8.

Interference detection messages are the second type of continuously running control traffic. This includes information about each computation currently supported by the system, as well as resource usage and statistics for each machine in the resource pool. These messages require the most overhead, as they contain more information than the heartbeats, which are

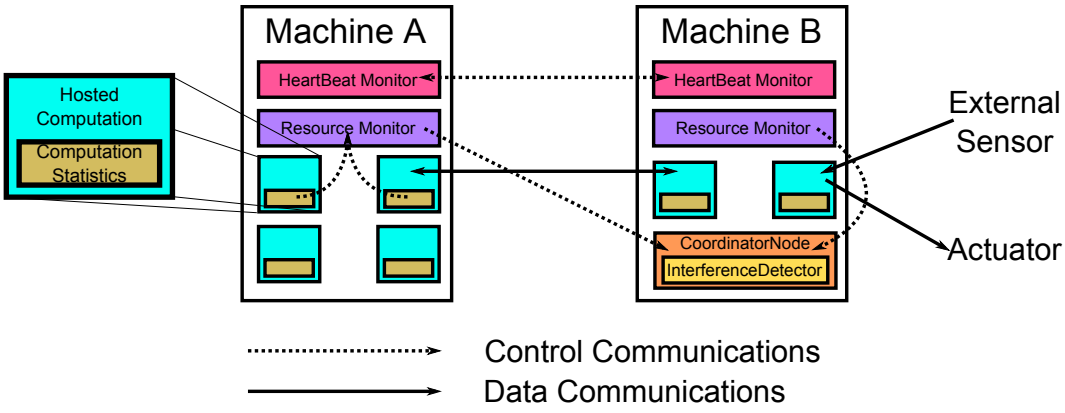

FIGURE 4.1. This figure shows components deployed on two distinct machines. The dashed lines denote *control traffic*; here, we see this as bidirectional communications between the HeartBeat components, from the hosted computations to the `ResourceMonitors`, and from the `ResourceMonitors` to the `CoordinatorNode`. The solid lines denote *data traffic*; We show one hosted computation both receiving data from an external sensor, and then passing results out to an actuator. Our framework puts no limits on either the source or sync of incoming data. We also show a hosted computation sharing state information with a replica.

essentially just pings. These messages represent another trade-off. We need to balance the amount of information sent, which could improve our decision making, with how much of our bandwidth we are willing to allow to be consumed by control traffic as opposed to data traffic.

4.5. SUMMARY

This chapter has introduced the tools we have developed to test and support the real-time processing of health stream data in a robust manner. The computations we have developed use publicly available health sensor data, and have been designed to not only represent a wide variety of application types and loads, but to do so realistically.

In order to enable robust behavior, we have developed multiple applications to monitor liveness of machines, the resource usage of each computation, and measure interference

between computations. Figure 4.1 shows a small sample of all the components working together, along with different examples of each type of communications.

CHAPTER 5

Detecting Failures

In distributed systems, it is impossible to tell the difference between a failed system and a slow system. As such, any failure detection system needs to balance swift detection with limiting false positives of failure reporting.

5.1. 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 [44] to smart condos [1] and personal health monitoring [2–8]. Sensors produce data intermittently or at regular intervals, creating *data streams* that are a set of correlated packets. Sensor data can be difficult to work with as data generation rates may spike due to external stimuli, e.g., increased tectonic activity or a person entering a room. Due to such stimuli, sensors may generate data at a higher rate, leading to bursty behavior.

Sensors designed to monitor health are 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 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 during failures of one or more machines. 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 data streams generated by sensors [9, 10]. To maximize resource utilization, Granules 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 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 heart rate while exercising is normal,

a sudden increase in heart rate may indicate cardiac problems. 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 chapter, 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.

5.1.1. 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 utilization. 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 that 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 resources. This creates the need to interleave multiple computations on a single machine.

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 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 framework.

To the best of our knowledge, no system has tried to address recovery in such an environment. Other distributed databases [30, 28, 45–47] do not rely on stateful computations for processing, and previous work in the realm of health stream processing [2–8] does not explore solutions at scale, and does not address fault-tolerance concerns at all.

5.1.2. Fault-Tolerance in Granules. 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 [47] by buffering data that arrives during the checkpointing process, it is not a good solution for health stream monitoring where delayed responses may lead to user injury.

5.2. Approach Synopsis

There are two aspects to failure resilient real-time stream processing. First, redundancy needs to be built into the system. Computations build state over time, and per-packet processing depends on this state. A significant loss of state may lead to incorrect results. Redundancy schemes try to minimize failover delays and reduce the amount of state lost during failure. Second, we need to have a fast, decentralized resource failure detection scheme in place to allow counter measures to be initiated. This chapter focuses on the task of reliably detecting failures in a decentralized fashion.

There are two inter-related issues for replication: customization and re-replication. Customization allows each computation to specify the degree of failure resilience. A higher replication level means that more failures can be sustained before a computation may be lost completely. To preserve redundancy, replication levels need to be maintained even during failures. The system achieves this using a combination of resource-level fast failure detection and re-replication that targets all affected computations. When a resource failure occurs, both primaries and secondaries hosted on that machine need to be re-replicated.

5.3. 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 cluster to detect failures in a fully distributed fashion – no single node needs to orchestrate communications. The fault-tolerant system is responsible only for monitoring liveness, ensuring a compact heartbeat message size to control the CPU, memory, and networking footprint of the HeartBeat scheme.

5.3.1. 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 cluster with two heartbeat groups, A and B, all machines in group A will send heartbeats to group B in the same timestep. This approach is particularly suited to Granules, since 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 the 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 5.1. 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 from which it has previously received heartbeats have sent a heartbeat in the last 6 timesteps – since the last time this check took place.

An additional variable is S, the number 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.

We can now analyze the detection interval in our scheme. Consider a cluster of N machines with M heartbeat groups, which has an update rate of T and failure suspicion count of S. The units for this measurement is determined by the units of T. In a best case scenario

TABLE 5.1. 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**
$0 \rightarrow 1$	$0 \rightarrow 2$	$0 \rightarrow 3$	$0 \rightarrow 4$	$0 \rightarrow 5$	***0*** $\rightarrow$ ***0***
$1 \rightarrow 2$	$1 \rightarrow 3$	$1 \rightarrow 4$	$1 \rightarrow 5$	***1*** $\rightarrow$ ***0***	$1 \rightarrow 1$
$2 \rightarrow 3$	$2 \rightarrow 4$	$2 \rightarrow 5$	***2*** $\rightarrow$ ***0***	$2 \rightarrow 1$	$2 \rightarrow 2$
$3 \rightarrow 4$	$3 \rightarrow 5$	***3*** $\rightarrow$ ***0***	$3 \rightarrow 1$	$3 \rightarrow 2$	$3 \rightarrow 3$
$4 \rightarrow 5$	***4*** $\rightarrow$ ***0***	$4 \rightarrow 1$	$4 \rightarrow 2$	$4 \rightarrow 3$	$4 \rightarrow 4$
5 $\rightarrow$ ***0***	$5 \rightarrow 1$	$5 \rightarrow 2$	$5 \rightarrow 3$	$5 \rightarrow 4$	$5 \rightarrow 5$

it will take TSM time in order to identify failures – where the failure is immediately detected upon the next heartbeat check, and enters failure suspicion for S rounds of checks. In a worst case scenario, it could take $(M-1)T + TSM$ time to detect failures. This case will occur should the system need to make a full cycle of checks. Using Table 5.1, assume that a machine in heartbeat group 5 fails and does not send a heartbeat to heartbeat group 0 (while this happens at **T0** in this table, we're assuming that the system has been running for some time, and has completed several cycles of sending and checking aliveness). Heartbeat group 0 does not again check for aliveness until **T5**, *M-1* timesteps later.

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, 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 maneuvers.

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 M, the number of heartbeat groups. To clarify, a given group will do a full check of the system every M timesteps. When a node has failed to send a heartbeat within those M 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 SM timesteps before the node is officially declared dead. In walltime, this results in a delay of SMT before fail-over actions can be taken.

CHAPTER 6

Replication

Replication is a well-known solution for circumventing failures in distributed systems. It is used in distributed file systems to help resolve inconsistencies in stored data [17], and in processing systems for a variety of purposes – from preventing data loss in the face of failures [11, 18], to increasing throughput [25].

Replication schemes broadly fall into one of three categories: *(1) Active* – where all computations actively receive and process data simultaneously; *(2) Passive* – where only one of the computations, the *primary* receives and processes data, with the *secondaries* ready to assume a primary role should failure occur at the primary; and *(3) Hybrid* – Some mix of the previous two categories.

6.1. 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.

6.1.1. Individual Computation Failure. For the purposes of this discussion, we assume that machine failures are 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.

For this analysis, our computations have a replication level of 3 spread across 3 machines: A, B, and C; and 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

computation is: $P(A_{fail}) \times P(B_{fail}) \times P(C_{fail})$. 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%). Through rack awareness and given that the probability of a particular machines failure is quite low, the probability of complete failure of a particular computation quickly decreases. 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.

6.1.2. 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 distinct computations with an arbitrary replication level of R, and N total machines supporting them. While we support replication levels set on a per-computation basis, for this analysis we are assuming that R is constant across all computations in order to simplify our calculations.

With U distinct computations, there are actually RU computations in the system in total (due to the replicas). Assuming that the machines are equally loaded, each machine will have about $\frac{RU}{N}$ computations on it. For both a best and worst-case scenario, replicas for at least $\frac{RU}{N}$ 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 $\frac{RU}{N}$ computations would fail after just R machines failed. In a best-case scenario, $(R-1)U$ computations could be lost before a distinct computation failed entirely. Since there are $\frac{RU}{N}$ computations per machine, this means that we can still run when $\frac{RU}{N} \times Y = (R-1)U$, or when $Y = \frac{R-1}{R} \times N$ machines fail. At this point, the failure of just one more machine will

guarantee the complete failure of computations. When $Y = \frac{R-1}{R}N + 1$, we will lose distinct 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 distinct computation is replicated R times, all R replicas need to fail for the computation to fail. The probability for the complete failure of a computation is $R\frac{Y}{N}$. To find the average number of failures, we multiply by the number of distinct computations: $R\frac{Y}{N} \times U$.

To test this, we work with the thorax extension dataset, which monitors respiration rates [41, 48], 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.

In this set of experiments, we are looking at the number of failed distinct computations given the number of failed machines. The `HeartBeat` settings are as follows: we use 24 nodes (N), 6 groups (M), a timestep (T) of 2s, and a failure suspicion count (S) of 2. We deploy 800 distinct computations (U) to this cluster, with a constant replication level (R) of 3. This means each machine has $\frac{RU}{N}$, or 100 computations running on it. In this experiment, we are looking at the number of failed distinct 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 6.1, 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

TABLE 6.1. 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

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.

6.2. BCI 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. 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 in each packet.

6.2.1. 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 resource pools. 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.

6.2.1.1. *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 [9, 10] which manages all computations on the machine. The resource is connected to a stream routing broker [49] which resides on another identical machine. EEG signals are fed into the system from a third identical machine which is responsible for recording round-trip classification times.

The generic trainers are deployed first, one to each node, and immediately begins batch training from the training sets. Once they have finished, the user computations are deployed in a round-robin configuration. Each computation receives an initial, generic group of experts from the trainer collocated on the same machine. 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. In the previous section we focused on failure rates given our replication scheme. Our goal in this section is to determine the effects of replication on the throughput of our system. We use the BCI computation as a baseline as it requires much more resources than the thorax computation. We determine the maximum number of users we can support while ensuring that classifications are returned in a timely manner.

TABLE 6.2. 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

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 programs which simulate 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.

6.2.1.2. *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 [42]. With these settings, we found that only one message out of the 150,000, i.e., 0.0006%, failed to return to the user in time (Table 6.2).

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 6.3. 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.

TABLE 6.3. 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

Analyzing the probability density of the passing response times (Figure 6.1), 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 passing range.

Our overall failed classification rate was still relatively small, at only 0.2%. To stress the system even further, we ran one more test with 40 users per machine. The results for this experiment are shown in Table 6.4. 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 6.4.

TABLE 6.4. 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

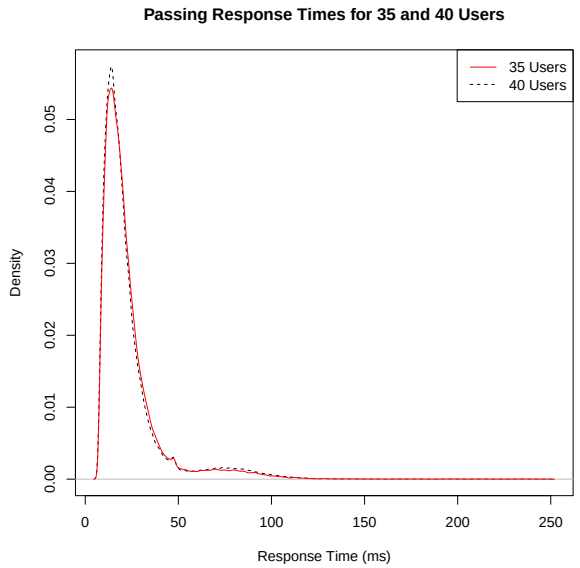

FIGURE 6.1. Density functions of passing response times in milliseconds for 35 and 40 users on a single node (ms)

Figure 6.1 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. Essentially, data is arriving simultaneously, meaning computations are queuing up to be activated to process their data. While most computations finish processing between 10 and 50ms, other computations need to wait for this initial processing to conclude before starting, leading to a little bump of straggler computations which finish 50 to 100ms after the data was sent.

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. As we attempt to support more users on a single resource, we are exacerbating interference problems between these collocated computations.

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 one avenue of future work is to explore methods of decreasing the footprint of R. Alternatively, we are also interested in exploring the memory overheads when using different implementations, such as in C or Java. Any implementation outside of Java, however, has the potential to suffer from the problem of not knowing the difference between the primary and backup replicas.

6.2.1.3. *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 chapter 5, 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.

The results of this experiment are displayed in Table 6.5. 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 that are very high.

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 well over 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. The results from a fully active cluster can be seen alongside the passive results in Table 6.5.

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 can be limiting, as we begin to hinder our scaling capabilities.

TABLE 6.5. 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

6.2.1.4. *Re-Replication of BCI Computations.* For this experiment, we worked with a fully loaded small cluster of 4 machines. After all computations had been loaded, we then killed one resource, and recorded the amount of time it took the backup replica to launch a new computation and send it all needed state information (for the BCI example, this includes the replica ID number and neural network for processing). In this process, we are looking to isolate the time it takes from the "Launch a new computation on machine X" message to be sent out in the system until the new computation is capable of processing data packets from the client. We further break down this period into 3 distinct times: *(1)* computation

instantiation – the time it took the computation to start up and request state information; *(2)* basic state transfer – the time it takes to receive basic computation state; and *(3)* neural network transfer – the time it takes to compress, send, and decompress the neural network.

As we can see from the results in Table 6.6, the re-replication process can be quite long, with an average overhead of 23.5 seconds. Given that EEG data is streamed every 250ms, this means that approximately 94 data packets have been sent during this time period. Looking at a breakdown of the times, we saw that the amount of time it took to actually instantiate a replica and get it ready for the neural network transfer was negligible – the vast majority of our overheads are compressing and sending the neural network to the new replica. We are hitting a bottleneck in our network and processing capacities, where attempting to get better times would begin to interfere with other computations. However, since this is a re-replication task and outside the path of critical processing for incoming data, it should pose a minimal risk to user safety.

TABLE 6.6. Re-Replication Overheads

	Mean (ms)	**Min (ms)**	**Max (ms)**	**SD (ms)**
Initialized	108.28	0.15	891.48	276.73
Basic State	628.65	501.32	925.24	153.96
Compress NN	11808.06	8746.83	22697.82	4021.34
Transmit NN	10024.36	8922.81	12818.59	1170.94
Decompress NN	1309.56	1261.52	1593.22	100.190
Overall	23578.92	19158.78	35513.15	5184.55

6.2.2. FULL SCALE CLUSTER STRESS TESTS. Previously, we found our network to be an effective bottleneck at 150 concurrent users. In Section 6.2.1, we found that memory started to became the bottleneck on a single node without replication at 35 users. In this subsection we explore the maximum number of concurrent users we can 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.

6.2.2.1. *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 streamed 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 as buffers overflow. In such situations, we are unable to properly asses worst case scenarios.

6.2.2.2. *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 BCI 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 due to congestion and spikes) 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 6.2.1, 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 6.2, 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.

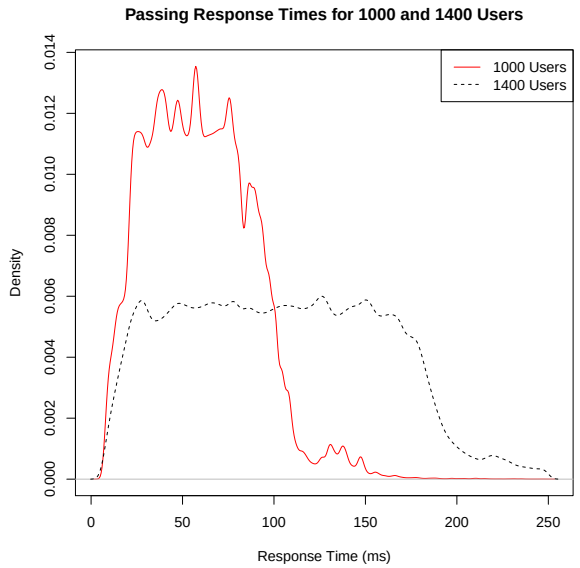

FIGURE 6.2. Probability density functions of passing response times in milliseconds for 1000 and 1400 users on a cluster of 40 nodes (ms)

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 6.2, 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, such as one that relies on the use of minimum spanning trees to disseminate content while making efficient use of the underlying bandwidth.

6.2.2.3. *Replication in a Large Cluster.* For the last set of experiments in this chapter, we decided to analyze how our replication approach scales. We hosted 450 distinct BCI 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 computations running in Java can recognize the difference between a primary and a replica, the R-based components of a computation 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 M each containing 9 machines. Keeping a T of 2 seconds and a S of 2, this leaves our best (TSM) and worst ($(M\text{-}1)T + TSM$) 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 uncommon ([11, 18, 45, 12]),and is a key functionality that we have built into the Granules framework.

In the context of BCI, 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.

6.3. Leveraging Replicas for BCI

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.

6.3.1. 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.

6.3.2. Dual Processing of Inputs. In stream processing systems some research has gone into exploring how replicas can reduce latency [28], this approach could also be adapted to our process of classifying 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 timely or delayed 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 wheelchair application 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 data is occurring.

6.3.3. 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 the particular 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 the 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.

6.3.4. 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 confirm that a failure has actually occurred. While we have been focusing on the task of detecting failure more quickly, a related issue is 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.

6.4. Summary

In this chapter we have described our approach to fault-tolerance through replication, as well as the costs seen for re-replication. While replication is a well-understood solution to fault-tolerance, we have developed a novel, distributed solution to detecting failures. This is a core component of our framework, and fully tunable by users in order to meet the needs of their clients. While our system has been designed specifically for the purpose of health stream processing, there is nothing to preclude its use in any other system.

Our experiments with the thorax computation have shown that our approach not only reliably outperforms theoretical average, but often approaches the theoretical best performance for failure detection and the amount of time needed to instantiate failover maneuvers.

With our EEG experiments, we explored the overheads associated with our failure detection system, and how these affect our overall system throughput. While the failure detection system does reduce the number of concurrent computations a single machine can hold, the ability to continue processing in the face of failures outweighs this decrease.

We have further benchmarked the re-replication process with the BCI computation, which is the most costly of our motivating applications. Given the overheads associated with re-replication, we have determined that: *(1)* re-replication tasks need to be taken outside the path of critical processing; and *(2)* average time to failure in a given resource pool will dictate the amount of replicas hosted computations require. Should the average time to

failure be significantly less than the re-replication timescale, we need a greater replication level to ensure computations are not lost entirely.

CHAPTER 7

Detecting Interference

Assigning a single computation to a single machine would avoid interference entirely, but would also mean that resources are being vastly underutilized. In order to prevent underutilization, we take advantage of Granules' ability to interleave computations on a single machine. This means that instead of needing a server farm for only a few patients, we would be able to support a hospital or nursing home with a small- to medium-sized cluster.

7.1. Defining Interference

Interference occurs when multiple computations are vying for the same set of limited resources. Such resource contention can lead to increased context switching, queue lengths, and computation times. As a cluster is expected to support more and more computations, interference levels will rise, meaning that interference detection and avoidance become more and more necessary. A related issue here is of resource imbalances – a small number of resources may be responsible for a disproportionate share of the processing.

For our purposes, we define interference as occurring when an individual computations' throughput suffers a decrease while a collocated computations' throughput is increased. There are several caveats to this case, however, the most prominent being that different computations will have different data arrival rates, leading to different maximum possible throughput.

The ultimate goal of our system is to ensure that no one computation prevents the processing of another computation. This would both be unfair, and potentially create an unsafe environment. In order to prevent this from happening, we need to take several precautions. First, we need to quickly detect interference as it happens. The loss of a

small number of packets may be acceptable, depending on the application, but the loss of a large number is not. So the ability to reactively respond to interference as it occurs is necessary. Second, we need to take proactive steps to prevent interference from causing problems. Relying on solely a reactive approach is far more likely to result in problems in the future. For interference which we cannot detect, we can then fall back on our reactive approach to resolve these problems as they occur.

7.2. Clustering Interference

When using a purely reactive approach, it is possible to detect interference by simply observing degradation in performance, but there is still the problem of trying to figure out how to reduce interference on the machine. We solve both the detection and placement problems using the same approach: clustering on computation state.

We use information on the current state of each computation to detect interference. Computations are expected to interfere if they have similar processing requirements, and similar activation arrivals. Essentially, we expect to see interference between computations that need the same resources and activate at the same time. Due to the fact that similarities signal interference, we decided to use a clustering algorithm to detect interference. This information is collected regularly from each computation as its `ComputationStatus`. Each status is projected into multi-dimensional space. Similar computations will be physically nearer to each other in this space, making a clustering algorithm a logical choice. This also allows us to quantify the degree of interference. We rely on Euclidean distance measures in this multi-dimensional space. Computations that are further apart are less likely to interfere with each other.

Previous work in mitigating interference has largely been reactive, so this is a novel approach. To our knowledge, no other system uses clustering on predicted interference to decide computation placement or migrations.

7.2.1. Clustering approach. Our approach uses k-means clustering with Euclidean distance measures. We have `k` set to 5, and cluster across all the information we collect from each computation: Data arrivals, amount of data and internal communications packets, and average size of data and internal communications packets. `k` was determined after running several experiments and observing cluster centroid distances. For the current set of computations that we are hosting, we found 5 to generally work well. One avenue of future work would be to allow `k` to change over time as needed.

For our Kmeans implementation, we are utilizing the publicly available kmeansclustering package: `https://code.google.com/p/kmeansclustering/`. Our data is relatively sparse, which can cause difficulties in clustering without the proper implementation. We use the default clustering parameters, which runs for X iterations, or until cluster centroids have shifted less than Y. One avenue for future work is to explore the effects of different Kmeans settings, as well as other clustering functions entirely.

7.2.2. Using Clustering. Clustering returns an assigned cluster ID as well as its distance from the assigned cluster centroid. We use the cluster ID information to determine which computations interfere with each other. Computations which are in the same cluster have similar activation times and resource requirements, so should be placed apart from each other if possible.

Given our desire to host hundreds of computations on a small to medium-sized cluster, completely avoiding all interference is impossible – such a situation would require close to

the single computation to a single machine idea. As eliminating all interference is infeasible, we need to focus on minimizing interference. We focus on ensuring that all machines in the system share an equivalent amount of interference. This means that each machine will share an equal load, and all computations are given an equivalent share of resources.

No system can handle processing requirements that outpace what is available within a collection of resources. In order to prevent the system from slowly sinking under more computations than it can accommodate, it becomes the responsibility of the `Coordinator Node` to alert system administrators when the load is greater than the current resource pool can handle.

We can also use the information on distance from the cluster centroid to detect the severity of interference. Computations which are further from the centroid are less likely to interfere with computations which are closer to the centroid, and vice versa. On the other hand, computations which are similar distances away are more likely to share the same plane of multi-dimensional space – meaning that their statuses are more similar, and more likely to interfere with each other.

7.3. Clustering Overheads

For clustering to be an effective measure of interference, we need to first make sure that the overheads induced by clustering do not preclude use in a real-time system. For this experiment, we explore the clustering overheads for converting computation statuses to clusterable points, the amount of time spent running clustering, and the amount of time looking for potential migrations. This information lets us know how often we can run interference detection without impacting overall cluster performance.

For these experiments, we ran both small- and large-scale tests, from 2 to 40 machines. From previous experiments, we know the maximum number of thorax computations we can reliably place on each machine to be around 100. For these experiments, we are only using thorax computations, as they are most light-weight, and we can fit more thorax computations on a single machine than any other type of computation. The clustering system is not impacted by different types of computations, only the number of computations.

Based on the results of this experiment, we can determine both how often we can run clustering, as well as how many computations a single `Coordinator Node` can reliably handle. This value would be different for different clustering methods, but will tell us at what point we need to develop a federated system to handle new computations. This would allow us to determine the number of `Coordinator Nodes` that must be present as the scale of the system increases.

For our small-scale tests, we deployed an increasing number of computations on 2 machines. We found the clustering time to increase linearly as more computations are added, as shown in Figure 7.1. While our overheads are relatively noisy, this is still a promising sign that we can achieve our goal of supporting a hospital or other institution on a medium-sized cluster. We have further supplied the actual values for mean, minimum, maximum and standard deviation of our benchmarks in Table 7.1. While it is important to ensure that the mean clustering overheads remain manageable, it is also important to take into account the frequency of significantly larger outliers. As with other aspects of our system, allowing this process to consume too much time could ultimately result in lost patient data.

Following our small-scale experiments, we next moved on to a much larger scale, using 40 machines. The results of these experiments can be seen in figure 7.2. These experiments show a drastic increase in the clustering overheads, with a sharp increase just above 1000

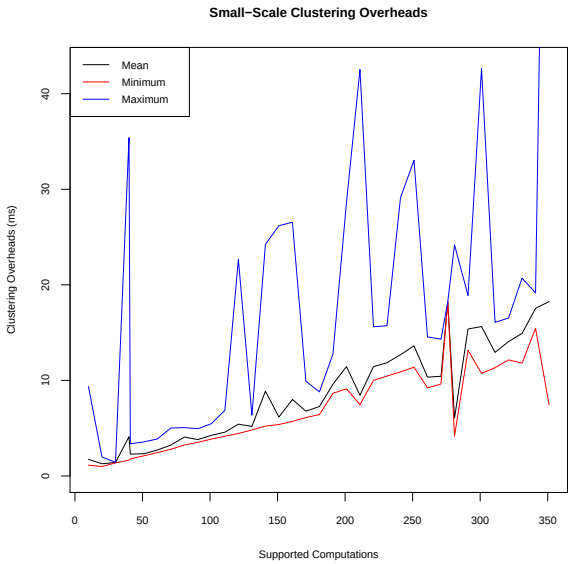

FIGURE 7.1. Small-scale clustering overheads across 2 machines with increasing computation loads (ms).

TABLE 7.1. Small-Scale Clustering as Hosted Computations Increase (ms)

Computations	**Mean (ms)**	**Min (ms)**	**Max (ms)**	**SD (ms)**
10	1.744	1.132	9.358	1.747
20	1.278	0.990	1.983	0.390
30	1.408	1.372	1.435	0.017
40	4.114	1.658	35.401	8.663
101	4.258	3.862	5.472	0.504
201	11.446	9.119	28.598	4.677
301	15.636	10.736	42.634	10.959
311	12.931	11.326	16.075	2.231
321	14.062	12.150	16.527	1.727
331	14.915	11.808	20.701	3.168
341	17.543	15.445	19.156	1.305
351	18.247	7.450	108.119	3.764

computations, and another, less drastic one at just over 3000 computations. At over 5000 supported computations, our interference detection system was able to cluster all computations in under 500ms in the worst case. We have also provided actual mean, minimum, maximum, and standard deviation values in Table 7.2.

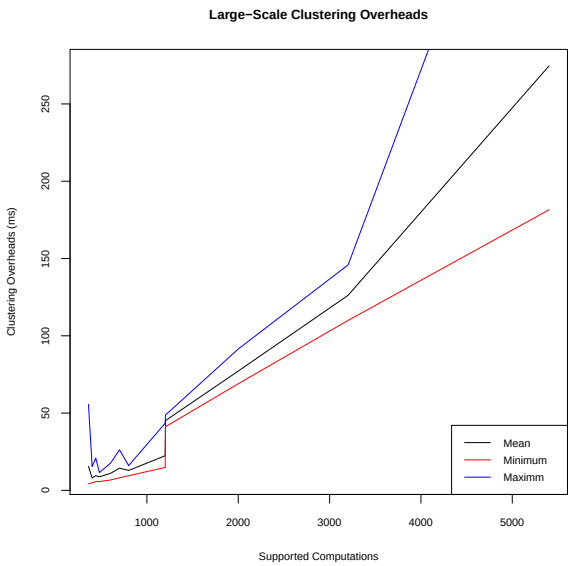

FIGURE 7.2. Small-scale clustering overheads across 2 machines with increasing computation loads (ms).

TABLE 7.2. Large-Scale Interference Clustering as Hosted Computations Increases (ms)

Computations	**Mean (ms)**	**Min (ms)**	**Max (ms)**	**SD (ms)**
360	15.507	4.259	55.638	16.619
400	8.028	4.867	15.462	2.552
440	9.502	5.671	20.856	3.9723
480	8.765	5.699	11.560	1.767
600	11.003	6.722	17.535	2.234
700	14.341	8.125	26.207	5.084
800	12.909	9.479	16.040	2.007
1200	22.417	14.849	43.491	7.864
1203	45.125	41.209	48.751	2.503
2003	77.295	69.097	91.621	7.154
3203	126.217	109.986	145.891	17.324
5403	274.611	181.518	493.842	27.201

7.4. INTELLIGENT PLACEMENT

We have developed our interference detection system to respond to interference both reactively and proactively. We can also utilize the interference information we are generating to aid in the initial placement of computations.

Previously, we have exclusively used a round robin approach to computation placement. Assuming that each computation has a relatively equivalent load, such an approach guarantees a balanced load. One thing that it cannot take into account, however, is computation activation times. Should all collocated computations receive data at the same time, they will all be competing for limited resources at the same time.

CHAPTER 8

Migration

Migrations serve many purposes. They can be used to alleviate current load on a machine, improve a computations throughput, or even avoid machines that are scheduled for maintenance.

We have developed a migration system that responds to both detected and predicted interference. To solve both problems effectively, we have developed two different migration schemes: *soft migration* and *hard migration*

8.1. Soft Migration

Soft migration takes advantage of our hybrid replication scheme. This is suitable as a quick response to either imminently impending interference, or currently detected interference. If the machine which is hosting the primary computation is experiencing slowdowns, it is possible to switch the replica that is acting as primary. While it will not completely remove all load from the original primary's machine, due to the computations update schedule, it does reduce the current load, and increase throughput for the migrated computation.

One prerequisite for a soft migration is that one of the machines hosting a replica can actually support the load of hosting the primary. If performing a soft migration will result in another machine becoming overloaded, the system will consider a *hard migration*. Additionally, hard migrations are a good choice when interference is detected further into the future. A hard migration carries more overheads than a soft migration, meaning that it is not a good choice when an immediate response is needed.

8.1.1. Small Scale Soft Migration. In our stress tests we overloaded the cluster with identical BCI 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 that 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, shifting 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.

In this experiment we deployed 24 BCI 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 soft migrations. We are currently only looking at machine load and queue delays. On the active replica, we gather information about the time that data packets spend waiting in a queue for processing. All replicas have the ability to query their host machine about the 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: *(1)* if the problem is because the system is overloaded, in which case shifting processing to another machine would be beneficial; or *(2)* if the problem is due to 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 have all replicas query for the status of their host machines, which includes the amount of memory allocated as well as how much of it is used, the number of currently hosted computations, and the number of data packets currently queued for processing. The user then chooses the replica with the lowest overheads to host the primary computation.

In order to limit *flapping*, where the primary role is constantly being switched between replicas, we have built in a 5 second "cooldown" period. After switching to a new primary, the system is required to wait 5 seconds (for our BCI application, this means 20 data packets will have been sent) before starting to look for a new primary. This approach prevented flapping in our experiments, but a more detailed look into these settings is a subject of future work.

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 classifications. 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 mitigation 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 to achieve greater improvements more efficiently.

8.2. Hard Migration

Hard migrations require much more overhead than a soft migration, as they involve deploying a new computation and bringing it up to the current state. Hard migrations are needed for situations where a soft migration will not solve imbalance problems. Hard migrations are also useful in situations where a machine is going down for scheduled maintenance.

Instead of waiting for a computation to failover, computations can be proactively migrated away from the machine which is about to reboot.

8.2.1. Small Scale Hard Migrations. For these sets of experiments, we explore how hard migrations behave on a small scale. By using a smaller number of machines for a testbed, it is easier to control when migrations are triggered, and where computations will migrate to. In order to further simplify matters, we used computations with a single replication level. While this ensures migrations occur, and that only hard migrations are possible, it also means that we are reducing communications overheads as there are no replicas which need to be informed of state updates.

8.2.1.1. *Homogeneous BCI Migrations.* Due to the nature of the state needed for the BCI application – the ANN trained to that specific user – BCI computation migrations are expected to have the highest overheads. For a computation to successfully migrate, the current ANN needs to be transferred to the new computation. These are on the order of 60MB. The size is determined based on both the complexity of the ANN itself, the number of inputs, and the outputs. It will vary depending on the type of amplifier being used, and number of units in the hidden layer of the NN, and the number of classifications it is expected to handle.

In order to fully capture the breakdown of migration overheads for the BCI application, we gather information for multiple steps in the process: time from initial migration decision to new computation instantiation, time to compress the current ANN, time to transfer the ANN, time to decompress the ANN, and then the complete overhead from the decision to migrate the computation until the NN is fully transferred, and the computation is ready to handle the processing of new data.

As we can see from the results in Table 8.1, the migration process can be quite long, with an average overhead of 23.5 seconds. Given that EEG data is pushed out every 250ms, this means that approximately 94 data packets have been sent during this time period. Looking at a breakdown of the times, we saw that the amount of time it took to actually instantiate a replica and get it ready for the neural network transfer was negligible – the vast majority of our overheads are compressing and sending the neural network to the new replica. We are hitting a bottleneck in our network and processing capacities, but attempting to get better times would begin to interfere with other computations. From this we can see that a hard migration task needs to be moved outside the path of critical processing for incoming data, in order to reduce risk to user safety.

Table 8.1. Small Scale BCI Migration Overheads

	Mean (ms)	**Min (ms)**	**Max (ms)**	**SD (ms)**
Initialized	108.28	0.15	891.48	276.73
Basic State	628.65	501.32	925.24	153.96
Compress NN	11808.06	8746.83	22697.82	4021.34
Transmit NN	10024.36	8922.81	12818.59	1170.94
Decompress NN	1309.56	1261.52	1593.22	100.190
Overall	23578.92	19158.78	35513.15	5184.55

8.3. Summary

In this chapter, we have presented our approach to migrations. Our work stands out in that we differentiate between soft and hard migrations. Soft migrations are only available in systems with either passive or hybrid replication schemes – in a fully active environment, there is no passive replica to switch processing to. The closest parallel to our soft migrations is seen in the concept of 'speculative tasks', where the same task is run in parallel and the results from the first returned task are accepted. If a passive replica is located on a machine with lower load, it is promoted to primary status.

Our hard migrations are more typical of the migrations discussed in other works, with the caveat that our system is designed for a streaming environment. This adds extra complications to the process, since we cannot simply freeze a computation throughout the migration process, but must continue to process data in real time. Hard migrations typically have overheads an order of magnitude higher than soft migrations, yet can offer a more permanent solution to imbalance load.

Migration schemes are an important tool for load balancing. A good migration can lead to a better balanced load, allowing the resource pool to safely support more patient computations. Poor migration choices can exacerbate already existent problems in a cluster, in some cases resulting in flip-flopping, or oscillating migrations where computations never settle and are constantly migrating.

Our benchmarks of migration overheads tells us how often we can effectively use each type of migration. Scheduling multiple, simultaneous migrations is more likely to lead to difficulties in determining current load, possibly leading to an unstable situation where cascading migrations are launched erroneously. It is important to find a balance between limiting how often we launch new migrations with the systems ability to respond to changes in system state.

CHAPTER 9

Contributions and Future Work

9.1. Conclusions

In this dissertation, we have presented a framework to support the robust processing of streaming data in real time. Our motivating focus has been on health stream processing, which requires strict constraints on response times, as well as very strong fault-tolerance guarantees. While designed for health stream processing, there is nothing to preclude the use of this framework for any other sort of real-time sensor processing.

Our framework is built upon the Granules stream processing runtime, and takes advantage of its underlying communications and scheduling tools. To achieve fault-tolerance, we have developed a distributed `HeartBeat` system which can be fine-tuned in order to meet an institution's requirements with respect to time to failure detection, robustness to false negatives, and overall response time to failures.

In order to limit the amount of data lost in failure scenarios, we have also implemented replication to work within our fault-tolerance approach. Replication is a well-understood response to the possibility of failures, as has been thoroughly explored in Chapter 2. We believe that we have expanded upon this basic concept to provide a more flexible framework than previous works have provided. We support traditional active and passive approaches to replication, as well as a hybrid approach which allows developers to specify how many computations should be actively processing data, as well as how often state information should be transferred to passive replicas. To the best of our knowledge, no other system allows for this much flexibility. This provides the opportunity to developers to balance resource utilization with the number of supportable users, while tweaking robustness on a

per-computation basis. This is an ideal environment for a patient care scenario, allowing individualized guarantees not only on a per-patient level, but on a data stream level – a single patient can have a different policy for every sensor.

While replication adds to the robustness of the system and allows us to handle failures without affecting the user, this also makes the system more complicated. We now have X times as many computations and communications streams to handle, where X is the current replication level of a computation, as well as any additional state transfers which arise from a hybrid scheme. As discussed in Chapter 7, in order to handle this more complex situation, we have developed a novel system for interference detection and avoidance. Our system takes into account previous computation activation schedules in order to predict future activation times. A computation does not access system resources when dormant, meaning we now have a load profile. This lets us not only react to current imbalances in system load, but take proactive steps to limit the affects of future interference.

This interference system can then be leveraged to inform the placement of new computations in the system as well as prompt migrations of existing computations. To take advantage of the variety of replication schemes we support, we have developed two distinct migration operations: *soft migrations* and *hard migrations*, both of which have been described in Chapter 8.

Our system builds upon previous works in the field of distributed stream processing, using well understood techniques heartbeat systems to detect failures, and replication to reduce the impacts of failures. Our system fills a niche by providing a distributed, fully tunable fault tolerance system, and is unique in its ability to support a wide variety of replication schemes on a per-computation basis, even allowing the ability to switch replication scheme after the computation has been launched without impacting the user. We have further introduced

the novel blending of machine learning concepts with load balancing. To the best of our knowledge, no other system has attempted to run clustering on computation resource usage and activation in order to detect interference. We have further developed hooks allowing us to use various learning algorithms in order to predict future loads and activation times on a per-computation basis.

9.2. Contributions

This dissertation contributes to both the fields of stream processing and health stream processing. It also has the potential for societal impact.

9.2.1. Stream Processing. This dissertation works with the Granules stream processing framework to add fault-tolerance. While other stream processing frameworks have implemented fault-tolerance, our work offers several unique benefits. First, we have developed a novel heartbeat system. Our approach has been designed to scale dynamically, while limiting the amount of bandwidth consumed as the resource pool. This allows much larger scales than can be feasibly supported by a more traditional gossip-based approach.

We have also developed a framework which allows replication levels to be set on a per-computation basis. While this is found in both distributed file systems, such as HDFS and in distributed hash tables, such as Dynamo it is usually not supported in distributed stream processing systems, such as Borealis and Aurora. This means that a single resource pool can support a wide variety of computations, where other approaches would need to have separate resource pools for computations with different replication needs. This is ideal in a hospital environment, where computations monitoring patients in the ICU would need a higher replication level than for those monitoring patients who are staying for an overnight study.

We have further improved upon existing replication schemes by allowing a mixture of replication schemes within a single resource pool. To the best of our knowledge, previous work only allows for a single replication scheme in a deployment – active, passive, or a hybrid scheme. Our system allows for all types of replication within the same cluster. It is even possible for computations to switch between replication schemes over time. This is again particularly suited to a hospital environment where a patient may move from critical care to recovery and thus require lower replication guarantees.

Our work has also extended the Granules framework by developing reactive interference detection and resolution through soft migrations. The system is able to detect interference and take steps to mitigate it autonomously. This is handled entirely in the back end, and requires minimal code to ensure stateful computations properly handle state for this transition. While other systems have developed their own migration schemes, our work leverages passive replicas for quick responses to detected interference.

We have expanded on our interference detection to not only react to currently detected interference, but also predict interference in the future, and take measures to ensure that this interference does not come to pass. This is a highly desirable trait in any stream processing framework, as it means that data is far less likely to be dropped and lost forever. To the best of our knowledge, we are the first group to explore active interference detection and mitigation. Instead of simply waiting for interference to appear and affect computations, we can prevent the interference from occurring in the first place.

9.2.2. Health Stream Processing. Health stream processing has been gaining traction in recent years, with several studies showing not only its potential worth, but also

exploring what is required of such a framework. The big gap in current health stream processing research is simply the fact that no work has attempted to process health data both at scale and in real time.

Our work is unique in that we test our framework at extremely large scales, with hundreds of unique computations, while also enforcing strict processing constraints to ensure data is processed in real-time. This dissertation represents a large step in the field of health stream processing, as we have not only developed a framework which can support further research, but also developed benchmarks by which to compare future frameworks.

Society. This framework has the potential to change the face of healthcare. Hospitals would be able to provide care to larger numbers of patients with reduced direct monitoring by professionals. This would free doctors for emergent cases, providing one-on-one care for those needing it most. It also has the chance to provide better care for individuals with "personalized healthcare". Each patient would have a fuller, more complete history than can often be obtained through questioning the patient alone – this could lead to better diagnoses faster.

Our framework could also be applied to full-time care facilities. In many cases, it could be used to better care for those staying at the facility, providing information about patient status that the patient may not think to bring up, or isn't able to discuss with care providers. As in a hospital situation, it would allow healthcare professionals to provide better care to patients, without increasing workload.

Some patients would be able to avoid entering full time care facilities and instead stay in their own homes. In the case of convalescing patients, staying in a familiar environment can reduce recovery time, as well as reducing the chance of an opportunistic infection contracted at a hospital. It could also be used in situations such as when the elderly prefer to age at

home – patients get to stay in familiar surroundings, don't need to relocate, and can avoid the high costs of a full-time care facility.

In general, we are looking at the ability to provide better, more complete care with much lower human overheads and, hopefully, costs. These savings have the potential to greatly reduce healthcare costs for those who would otherwise need monitoring, a problem which will become increasingly prevalent as the baby boomer generation approaches retirement.

9.3. Future Work

Throughout this work, we have already mentioned several avenues of future work. In this section we both expand on these ideas, as well as present new avenues of exploration.

One big avenue of future work is to help improve the performance of bridged computations. While Granules is aware of which computations are dormant and can reduce their processing footprint, computations which bridge to applications not contained within Java are not able to take advantage of this information. We found this to be a problem with both the ECG and BCI computations we are supporting. These computations both rely on R-backed classification algorithms, and one bottleneck we have run into with respect to the number of supportable users occurs with memory usage. The R instances of dormant computations still take up as much memory as those which are actively being used. If we can find a way to communicate which computations should be dormant to other frameworks, we would be able to better support more users and with a wider variety of computation types.

We would also like to explore several aspects of our interference detection scheme. First of all, we would like to explore allowing the `k` in our Kmeans function to vary over time. Through our experiments, we have discovered 5 to be a good setting, given our current list of supported computations. As we support new computations, and explore different settings

of other parameters, allowing k to vary could provide better interference predictions. We would like to additionally explore other Kmeans settings, such as how we measure or scale distances between computations. Should memory usage be weighed the same as disk I/O? How should network I/O scale? Such explorations are beyond the scope of our current work, and may warrant a dissertation in their own right.

Another modification to interference detection would be the incorporation of other clustering approaches. Kmeans is well understood, but has certain limitations, such as a set `K`. Other clustering algorithms, such as Dirichlet will determine how many clusters should be formed while it is running. This value is then expected to change across iterations. This seems like a particularly ideal approach for a cluster which experiences relatively large churn over computation membership, hosting many short-lived computations.

We would also like to explore the application of other methods of predicting future interference. While we have developed our system with the intent to allow more complex predictors, in our current approach we simply assume that computations will continue their current usage pattern. This is a far from ideal situation, particularly as we expand to work with sensors which have different sampling rates available, and longer-running computations which collect data over a period of weeks or months instead of hours.

In our experiments, we found a minimal amount of damping was needed to prevent oscillations in migrations, where computations are periodically migrated between two machines and never enter a balanced state. As we work with longer and longer running applications, the possibility of such behavior increases. Our current approach of locking migrations from a machine until all previous migrations have completed has prevented such behavior so far, but it comes at the cost of limiting the speed with which we can react to current or predicted interference. One avenue of future work is exploring more fine-grained approaches to this

damping behavior, which would lock down individual computations from migrating instead of the hosting machine.

While we have conducted experiments on a much larger scale than previous forays into health stream processing, our ultimate goal is to support whole hospitals, or even a town, on a small to medium sized cluster. While such a grand scale is still a long ways off, working with larger amounts of users would allow us to discover what new challenges appear as we approach this goal.

All our current work has utilized publicly available health stream data sets. In order for our work to grow into an application which is a public utility alternative to full-time health care, we need to take the next step of working with live, streaming data from patients. If we can maintain our processing guarantees that we have held up so far with simulated data sets, we will be able to begin providing support for situations where full-time monitoring would otherwise be needed. This would involve not only working closely with patients, but their doctors and those who develop current monitoring applications as well.

Another avenue of future work involves making our system more robust to fluctuations in resource pool churn. Our `HeartBeat` system currently relies on settings made when it is initially launched to set up the `HeartBeat Groups`. Our framework could only be improved by adding the ability to modify the heartbeat settings at runtime in response to changes in both the resource pool membership and the hosted computations.

Our current implementation relies on previously benchmarked data about the hosted computations to make decisions about not only the current load of each machine, but also to predict how migrations will affect resource utilization. There are two possible directions we can move from this point. First, we can continue to use our current implementation, and focus on developing an automated benchmark tool. This would allow us to add new

types of computations to a resource pool which is already running. Our second option is to revamp the entire load measurement and interference detection system to use raw resource data about how much memory is being used, disk and network accesses, etc. While this would require a large investment, it is most likely to have the biggest payoff since such a system would not only be able to keep track of the computations we launch in it, but it would also be able to monitor machine usage outside the scope of Java – it will capture the impacts of background processes and tasks. Such a system would be able to migrate primary computations away from machines which are running updates, possibly causing a slowdown in response times.

Bibliography

[1] 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*, pp. 20–28, 2009.

[2] 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*, pp. 153–156, 2008.

[3] 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*, pp. 791–794, 2004.

[4] 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.

[5] 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*, pp. 248–251, 2008.

[6] 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.

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

[8] 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.

[9] 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), pp. 412–413, 2008.

[10] S. Pallickara, J. Ekanayake, and G. Fox, "Granules: A lightweight, streaming runtime for cloud computing with support for map-reduce," in *IEEE International Conference on Cluster Computing*, (New Orleans, LA), pp. 1–10, 2009.

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

[12] 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), pp. 59–72, 2007.

[13] 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.*, pp. 464 –474, 2000.

[14] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, pp. 51–59, June 2002.

[15] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," *SIGMOD Rec.*, vol. 25, pp. 173–182, June 1996.

[16] D. J. Abadi, "Data management in the cloud: Limitations and opportunities," in *Bulletin of the Technical Committee on Data Engineering*, pp. 3–12, IEEE Computer Society.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 29–43, October 2003.

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

[19] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, vol. 11, p. 21, 2007.

[20] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, pp. 50–58, April 2010.

[21] E. Amazon, "Amazon elastic compute cloud (amazon ec2)," *Amazon Elastic Compute Cloud (Amazon EC2)*, 2010.

[22] D. Chappel, "Introducing windows azure," tech. rep., Microsoft Corporation, 2009.

[23] D. Chappel, "Introducing the windows azure platform: An early look at windows azure, sql azure and net services," tech. rep., Microsoft Corporation, 2009.

[24] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, (Berkeley, CA, USA), pp. 29–42, USENIX Association, 2008.

[25] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, (Washington, DC, USA), pp. 4–10, IEEE Computer Society, 2004.

[26] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[27] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 205–220, ACM, 2007.

[28] 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), pp. 827–838, ACM, 2004.

[29] T. Repantis, X. Gu, and V. Kalogeraki, "Synergy: sharing-aware component composition for distributed stream processing systems," in *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, Middleware '06, (New York, NY, USA), pp. 322–341, Springer-Verlag New York, Inc., 2006.

[30] 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), pp. 277–289, 2005.

[31] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani, "Design, implementation, and evaluation of the linear road benchmark on the stream processing core," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, (New York, NY, USA), pp. 431–442, ACM, 2006.

[32] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pp. 779–790, IEEE, 2005.

[33] T. Repantis and V. Kalogeraki, "Replica placement for high availability in distributed stream processing systems," in *Proceedings of the second international conference on Distributed event-based systems*, DEBS '08, (New York, NY, USA), pp. 181–192, ACM, 2008.

[34] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," *SIGCOMM Comput. Commun. Rev.*, vol. 33, pp. 3–12, July 2003.

[35] J.-H. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik, "A cooperative, self-configuring high-availability solution for stream processing," in *IEEE 23rd International Conference on Data Engineering, 2007. ICDE 2007.*, pp. 176 –185, April 2007.

[36] J.-H. Hwang, U. Cetintemel, and S. Zdonik, "Fast and highly-available stream processing over wide area networks," *International Conference on Data Engineering*, vol. 0, pp. 804–813, 2008.

[37] X. Gu, S. Papadimitriou, P. Yu, and S.-P. Chang, "Toward predictive failure management for distributed stream processing systems," in *The 28th International Conference on Distributed Computing Systems, 2008. ICDCS '08.*, pp. 825 –832, june 2008.

[38] K. Ericson, S. Pallickara, and C. W. Anderson, "Failure-resilient real-time processing of health streams," *Concurrency and Computation: Practice and Experience*, 2014.

[39] M. GB and M. RG, "The impact of the mit-bih arrhythmia database," *IEEE Eng in Med and Biol*, vol. 20, no. 3, pp. 45–50, 2001.

[40] A. L. Goldberger, L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley, "Physiobank, physiotoolkit, and physionet : Components of a new research resource for complex physiologic signals," *Circulation*, vol. 101, no. 23, pp. e215–e220, 2000.

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

[42] 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), pp. 185–192, 2010.

[43] 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.

[44] 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*, pp. 271–280, 2010.

[45] 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.

[46] 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*, pp. 323–329, 1999.

[47] 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), pp. 257–268, 2003.

[48] K. Eamonn, "Hot sax: Efficiently finding the most unusual time series subsequence," vol. 0, pp. 226–233, 2003.

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